Meltdown

Reading Kernel Memory from User Space

Moritz Lipp | Michael Swartz | Daniel Gruss | Thomas Prescher | Werner Haas | Anders Fogh | Jann Horn | Stefan Mangard | Paul Kocher | Daniel Genkin | Yuval Yarom | Mike Hamburg

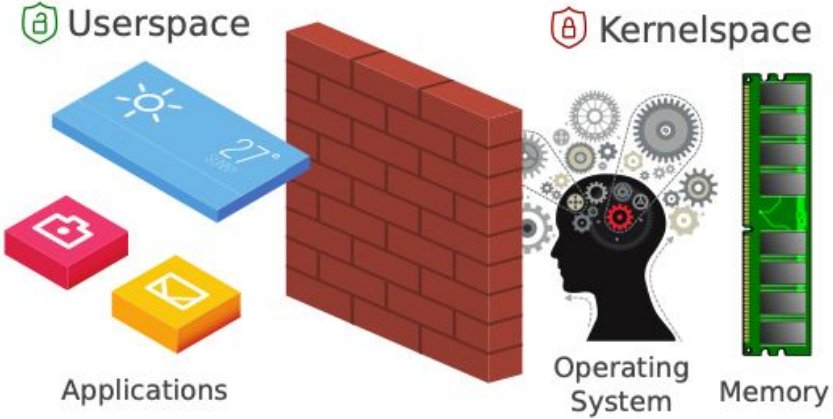
Presented by Christos Komis
sdi2200073@di.uoa.gr

What is Meltdown?

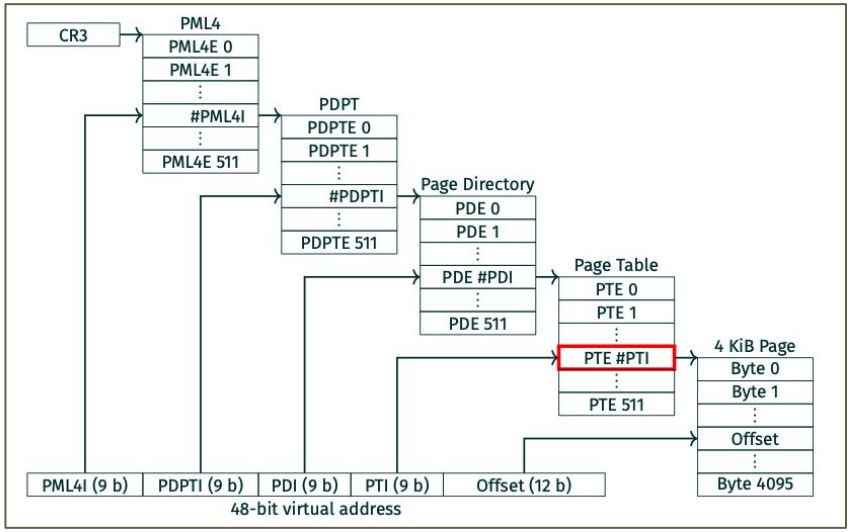
- Meltdown is a critical security flaw found in modern microprocessors (primarily Intel)
- It destroys the fundamental security boundary between user applications and the kernel
- Exploiting out-of-order execution, security checks can be bypassed
- Consequently, an unprivileged malicious program can read arbitrary kernel memory, exposing secrets like passwords and encryption keys

Background: Memory Isolation

- Kernel is isolated from user space
- User applications cannot access anything from the kernel
- This isolation is a combination of hardware and software
- CPUs support virtual address spaces to isolate processes
- Physical memory is organized in page frames
- Virtual memory pages are mapped to page frames using page tables



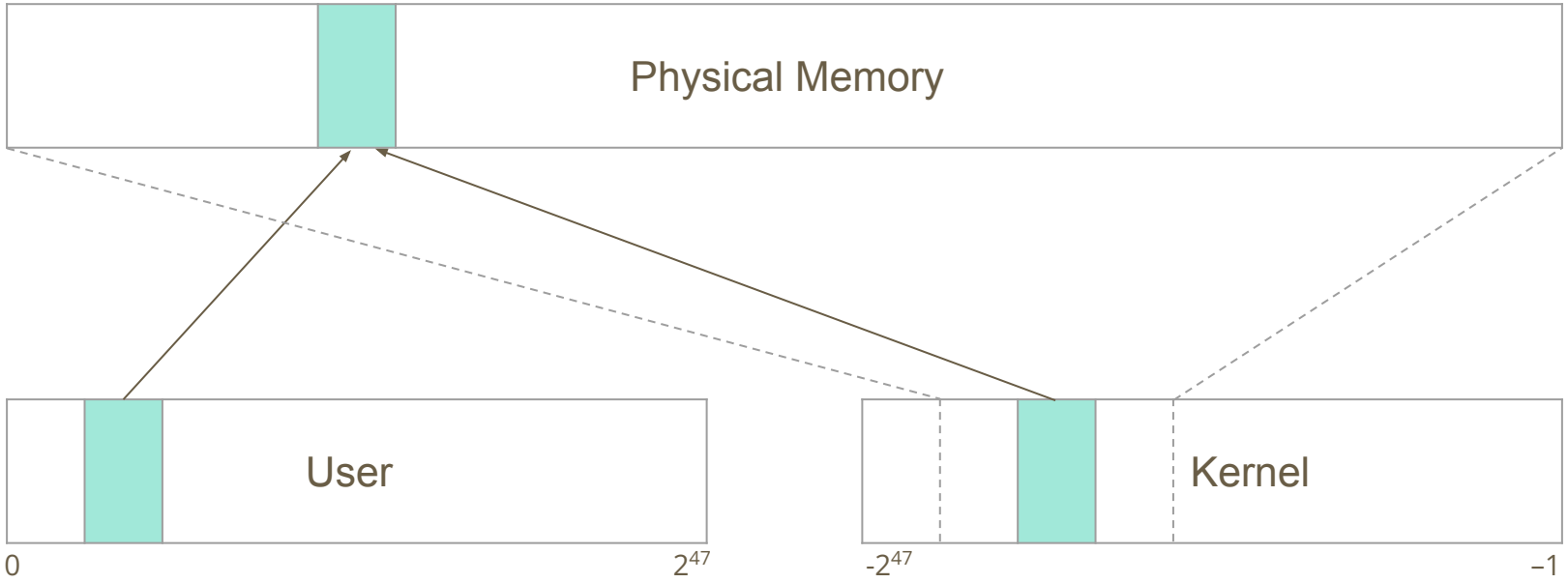
Background: Address Translation on x86-64



P	RW	US	WT	UC	R	D	S	G	IG
Physical Page Number								IG	EX

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

Background: Direct-physical map

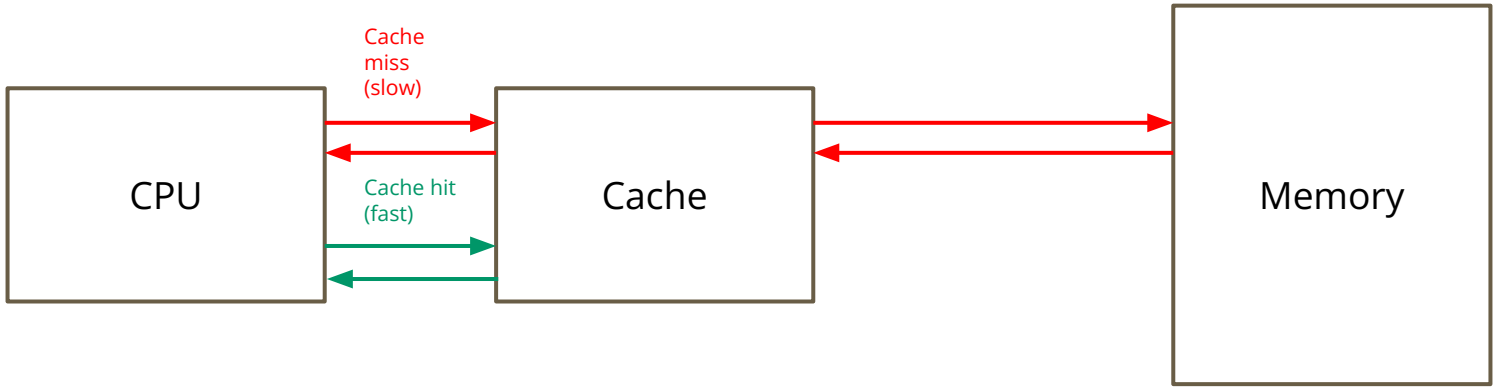


- Kernel is typically mapped into every address space
- Entire physical memory is mapped in the kernel

Background: CPU Cache

Purpose: Bridge the massive speed gap between the lightning-fast CPU and slow Main Memory (RAM) by storing recently used data nearby.

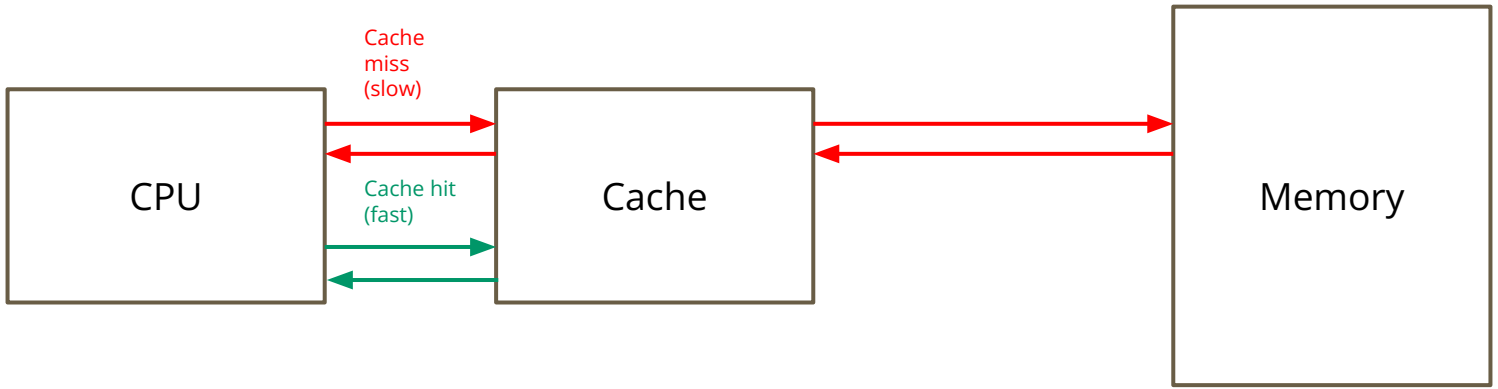
- Load Instructions that are served by the cache are faster
- Load Instructions that are served by the RAM are slower.



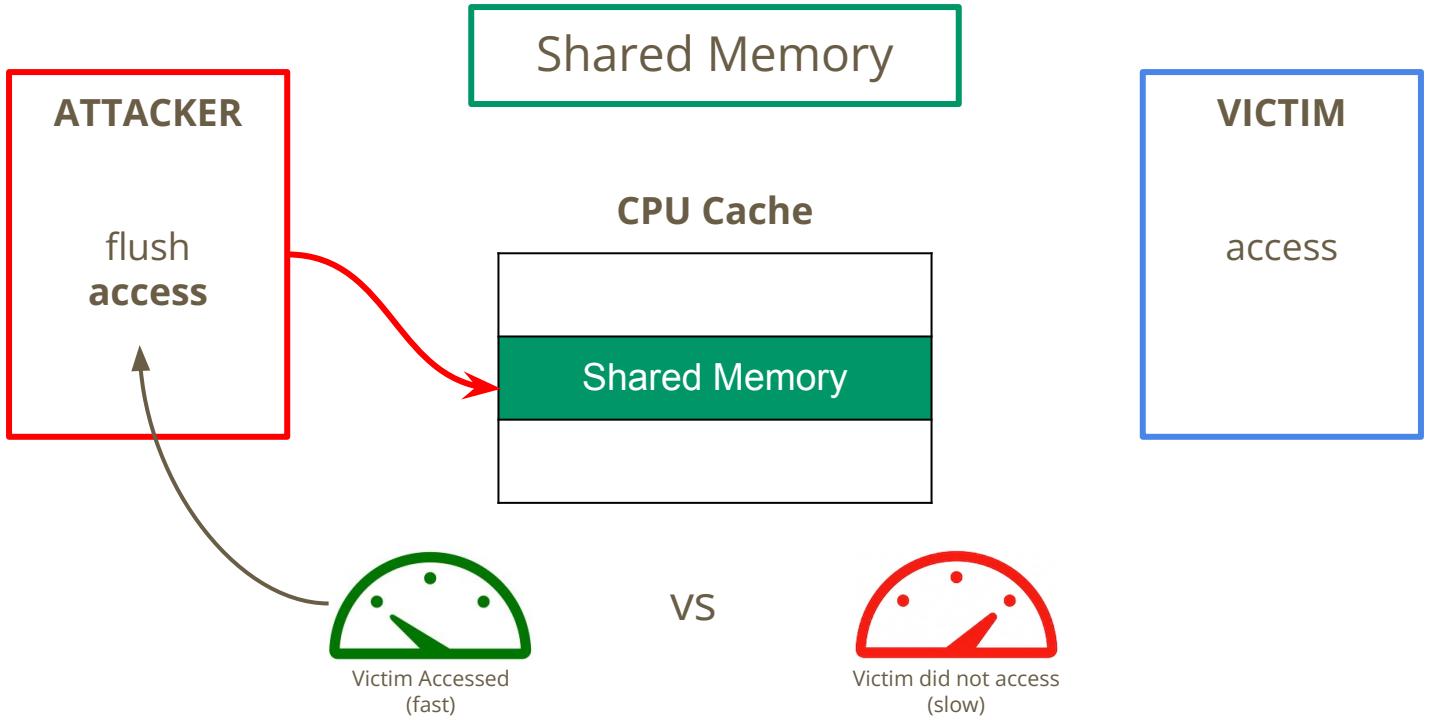
Background: CPU Cache

Side Effect: An attacker can time load instruction to determine whether a process accessed a specific address.

- **Flush+Reload:** A common cache side-channel attack on x86 that exploits this vulnerability.



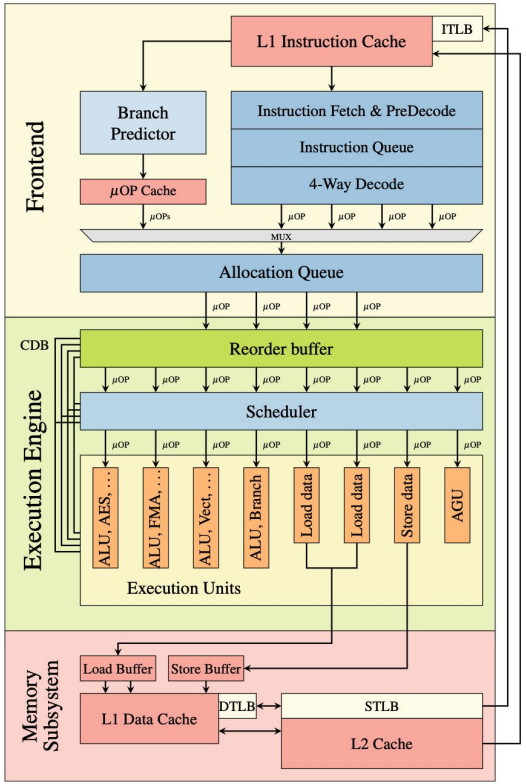
Background: Flush+Reload



Background: Out-of-Order Execution

Instructions:

- Are executed out-of-order
- Wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- Retire in-order
  - State becomes architecturally visible
- Exceptions are checked during retirement
  - Flush pipeline and recover state

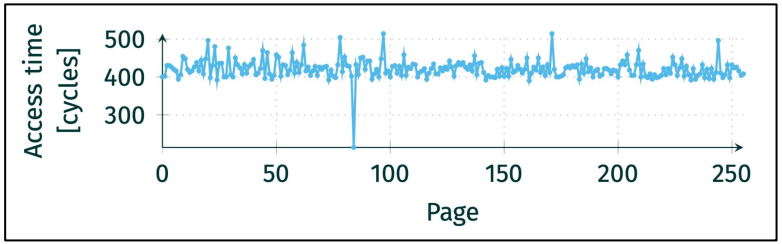


Meltdown: Toy Example

- “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

```
*(volatile char *) 0; // raise_exception()
array[84 * 4096] = 0;
```

Flush+Reload over all pages of the array



Executing Transient Instructions

- Transient instructions are executed all the time
- Loading inaccessible addresses leads to a crash (segfault)
- How to prevent the crash?

Exception Handling

Exception Suppression

Exception Prevention

Crash Prevention Approaches

Exception Handling

- *fork()* and let the child process crash
- Signal handler

Exception Suppression

- Transactional Memory

Exception Prevention

- Speculative Execution

Building a Covert Channel

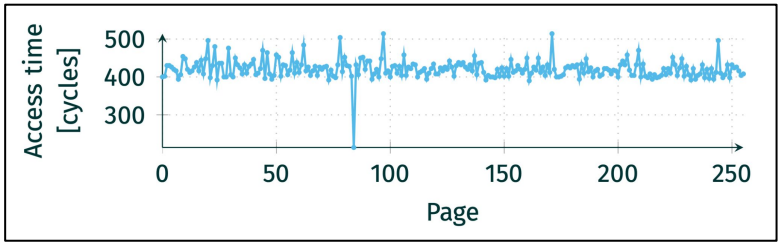
- Transient instruction sequence is the sender
- Receiver receives the microarchitectural state change and deduces the secret from the state
- Leverage techniques from cache attacks: Flush+Reload
- Transmit multiple bits at once
  - 256 different byte values → access different cache line
- Not limited to the cache

Building the Code

- Add another layer of indirection to test
- Check whether any part of the array is cached
- Index of cache hit reveals data
- Permission check is in some cases not fast enough

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

Flush+Reload over all pages of the array



Meltdown

- Using out-of-order execution, we can read data at any address
- Index of cache hit reveals data
- Permission check is in some cases not fast enough
- Entire physical memory is typically accessible through kernel space

Attack Description

Meltdown consists of 3 steps:

1. Content of inaccessible, attacker-chosen memory location is loaded into a register
2. A transient instruction accesses a cache line based on the secret value
3. Using *Flush+Reload*, attacker determines the accessed cache line.

Q: Why loop on 0?

```
; rcx = kernel address, rbx = probe array
xor rax, rax
retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
mov rbx, qword [rbx + rax]
```

```
register uint64_t secret = 0;
do {
      secret = *(uint8_t *)kernel_address;
      secret *= 4096;
} while (secret == 0);
p_arr = *(uint64_t *)(p_arr + secret);
```

Optimizations and Limitations

Inherent bias towards *0*

- CPUs usually stall if a value is not available during an out-of-order load operation
- However, CPUs may continue, assuming a value for the load
- Bias towards *0* may be due to:
  - Speculated value of stalled load
  - Memory load is masked out by a failed permission check

Optimizing the *0* case

- Place ‘retry’ loop in case of loading *0*
- Loop stops either on non-zero value or on exception

0 Occurrence Rate

Unoptimized	5.25% (σ = 4.15)
Optimized	0.67% (σ = 1.47)

Optimizations and Limitations

Single-bit transmission

- Trade-off between running more transient instruction sequences and performing more *Flush+Reload* measurements
- *Flush+Reload* measurements seem to be the performance bottleneck of the attack
- Solution: Transmit 1 bit at a time
  - Transmitted Bit 0 → Cache Miss on Index 1
  - Transmitted Bit 1 → Cache Hit on Index 1

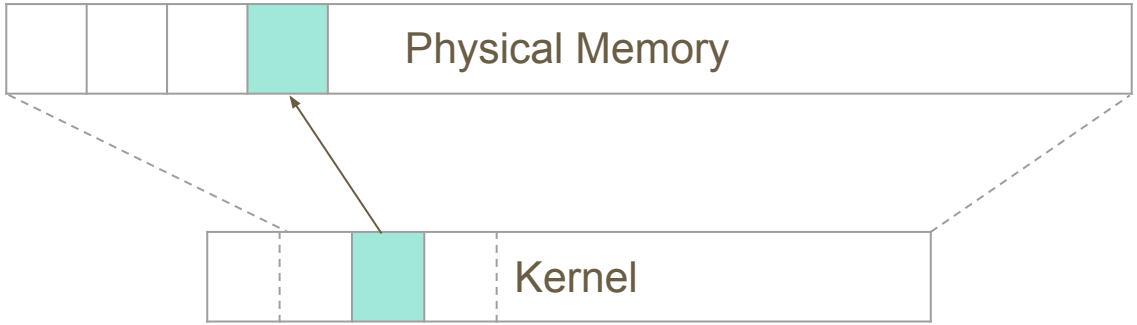
Exception Suppression using Intel TSX

- One instruction fails → already executed instructions are reverted and no exception is raised
- Suppressing the exception is significantly faster than trapping it into the kernel

Optimizations and Limitations

Dealing with KASLR

- Randomizes across a 40-bit space (~1 TB)
- Physical memory is mapped fully onto kernel space
- Assuming a setup of 8GB of RAM, attackers can step through memory in 8GB strides
- KASLR defeat in ≤ 128 tests



Meltdown in Practice

- Dumping the entire physical memory takes some time
  - L1: 582 KB/s
  - L3: 12.4 KB/s
  - Uncached: 10 B/s (improved: 3.2 KB/s)
- Not very practical in most scenarios

Affected by Meltdown

- **Intel:** Almost every CPU
- **AMD:** Seems not to be affected
- **ARM:** Only the Cortex-A75
- **Apple:** All Mac and iOS devices

Mitigations

- **Linux:** Kernel Page-Table Isolation (KPTI)
- **Apple:** Double Map
- **Windows:** Kernel Virtual Address (KVA) Shadow

Conclusion

- Microarchitectural attacks were underestimated for a long time
- Meltdown allows to read arbitrary kernel memory from user space
- Affecting millions of devices of various CPU manufacturers
- Countermeasures come with a performance impact