Meltdown

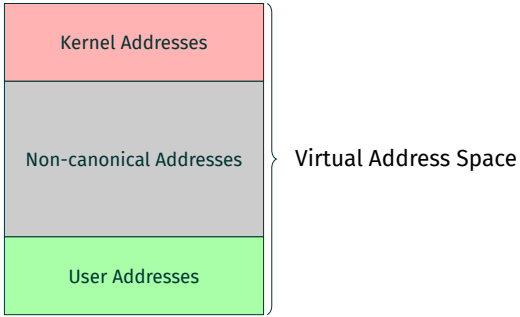
Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1], Paul Kocher[5], Daniel Genkin[6], Yuval Yarom[7], Mike Hamburg[8]

27th USENIX Security Symposium - August 16, 2018

[1]Graz University of Technology, [2]Cyberus Technology GmbH, [3]G-Data Advanced Analytics, [4]Google Project Zero, [5]Independent (www.paulkocher.com), [6]University of Michigan, [7]University of Adelaide & Data61, [8]Rambus, Cryptography Research Division

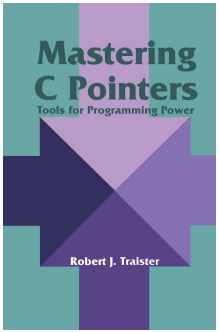
Virtual Memory



Building the Code

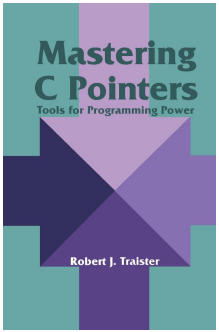
- Find something human readable, e.g., the Linux version

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



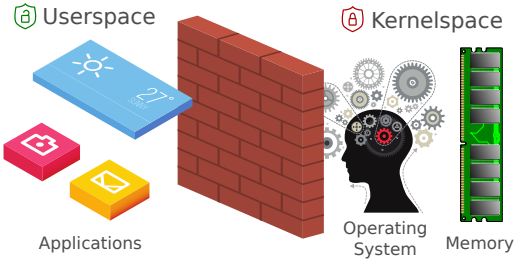

```
char data = *(char*) 0xffffffff81a000e0;
printf("%c\n", data);
```

Building the Code



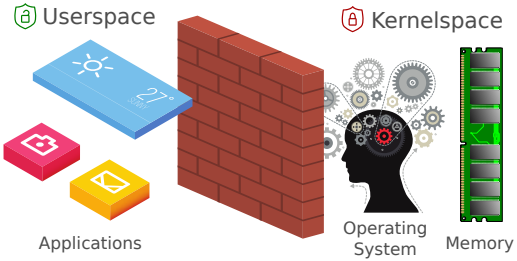

```
char data = *(char*) 0xffffffff81a000e0;
printf("%c\n", data);
```

- Any invalid access throws an exception → segmentation fault



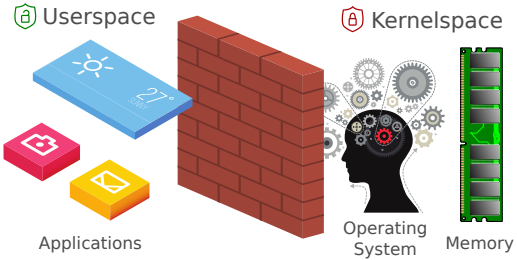
- Kernel is isolated from user space

Memory Isolation



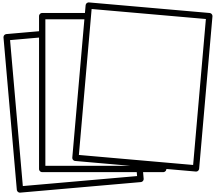
- Kernel is isolated from user space
- This isolation is a combination of hardware and software

Memory Isolation



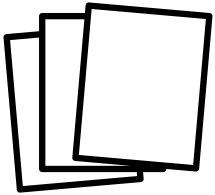
- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel

Paging



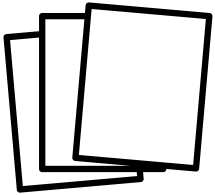
- CPU support virtual address spaces to isolate processes

Paging



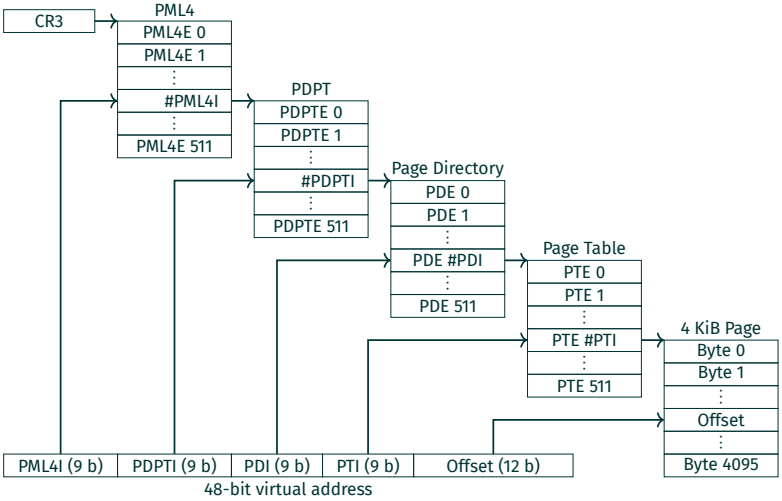
- CPU support virtual address spaces to isolate processes
- Physical memory is organized in page frames

Paging

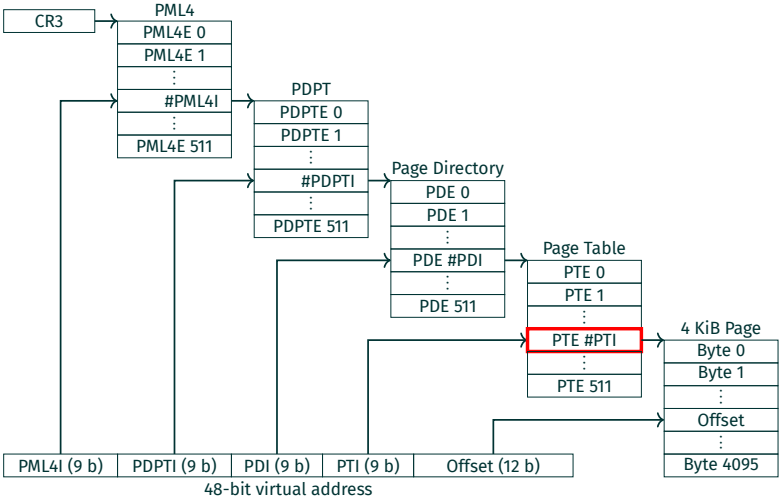


- CPU support virtual address spaces to isolate processes
- Physical memory is organized in page frames
- Virtual memory pages are mapped to page frames using page tables

Address Translation on x86-64

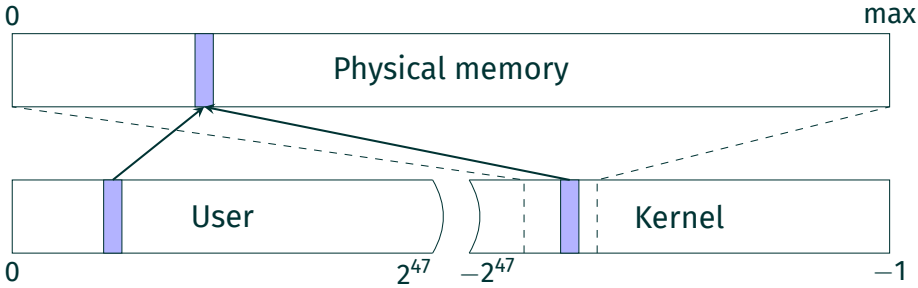


Address Translation on x86-64



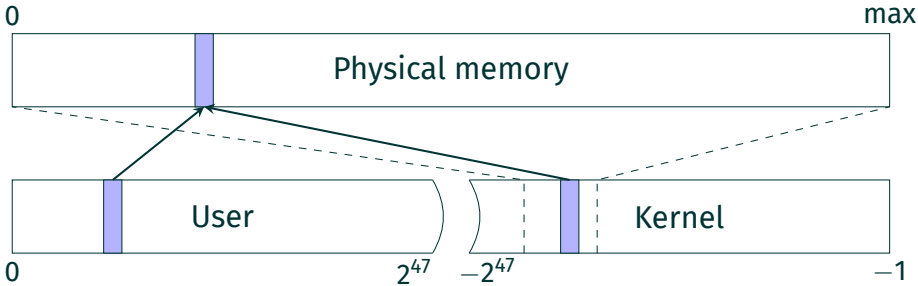
- User/Supervisor bit defines in which privilege level the page can be accessed

Direct-physical map



- Kernel is typically mapped into every address space

Direct-physical map

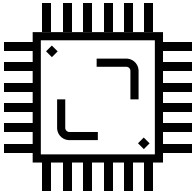


- Kernel is typically mapped into every address space
- Entire physical memory is mapped in the kernel

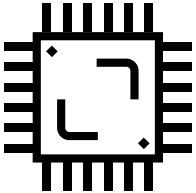
Loading an address

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

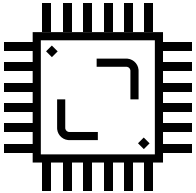
- We try to load an inaccessible address
- Permission is checked



- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)

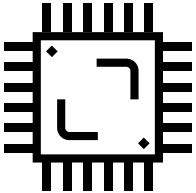


- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
- Serves as the interface between hardware and software



- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

Architecture and Microarchitecture

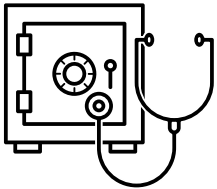


- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

●●●

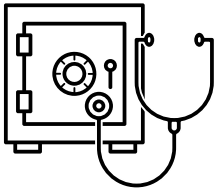
Side-channel Attacks

- Safe software infrastructure does not mean safe execution

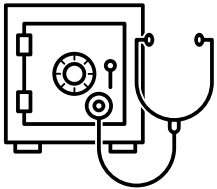


Side-channel Attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware

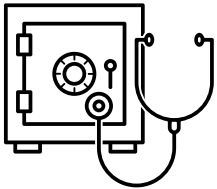


Side-channel Attacks



- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects

Side-channel Attacks



- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Exploit unintentional information leakage by side-effects

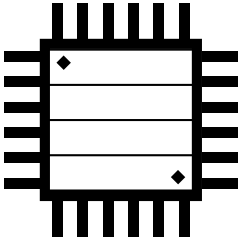
Power consumption

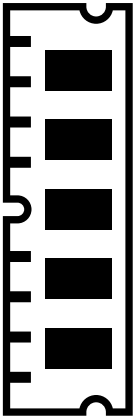
Execution time

CPU caches

CPU Cache

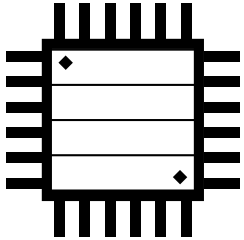
```
printf("%d", i);
printf("%d", i);
```

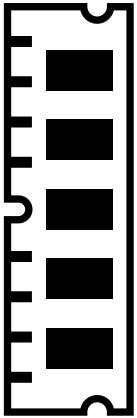


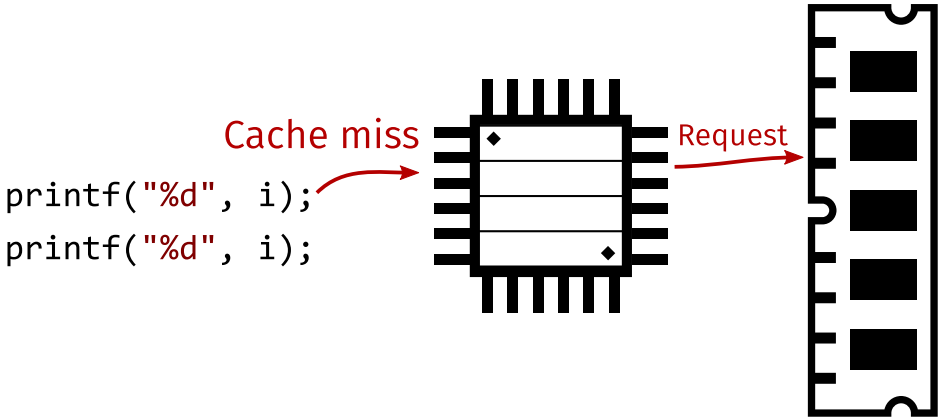


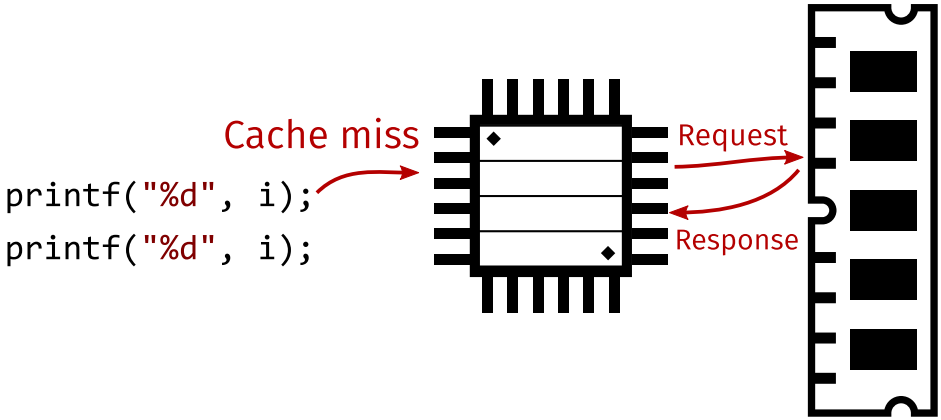
Cache miss

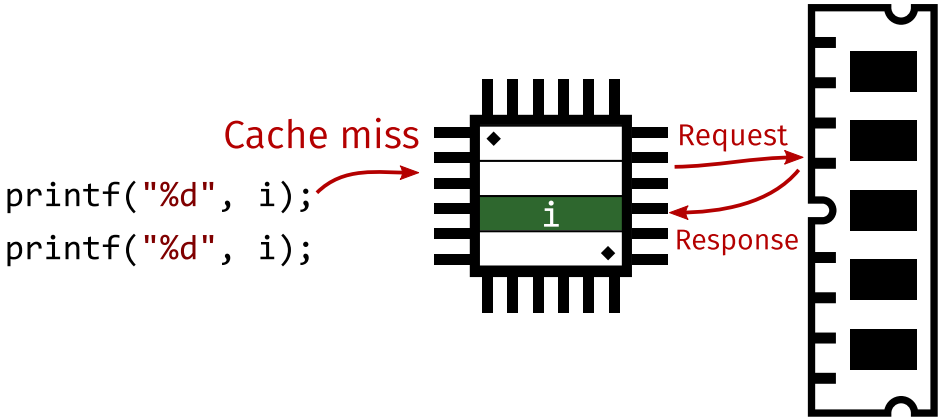
```
printf("%d", i);
printf("%d", i);
```

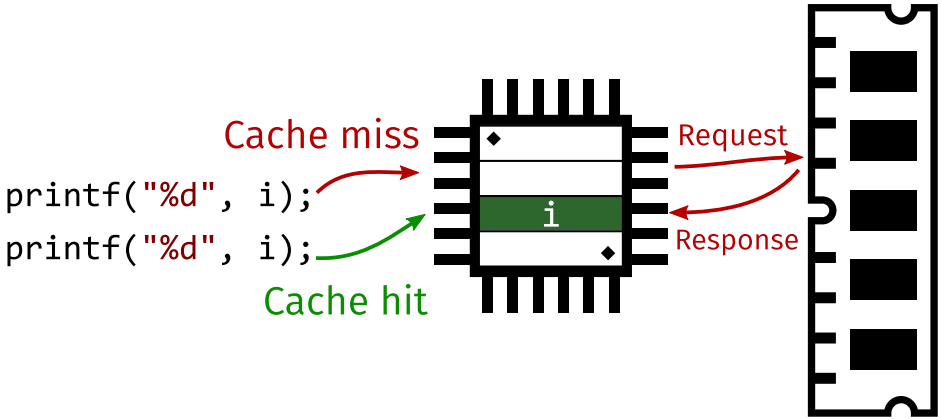


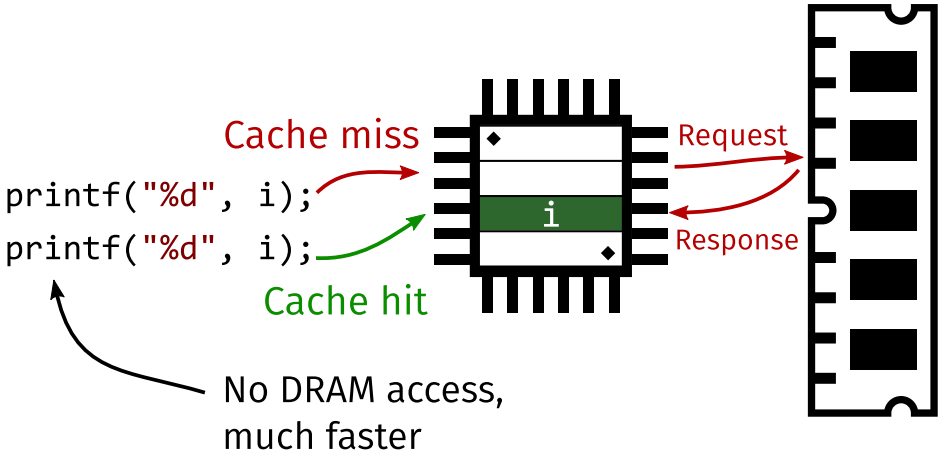


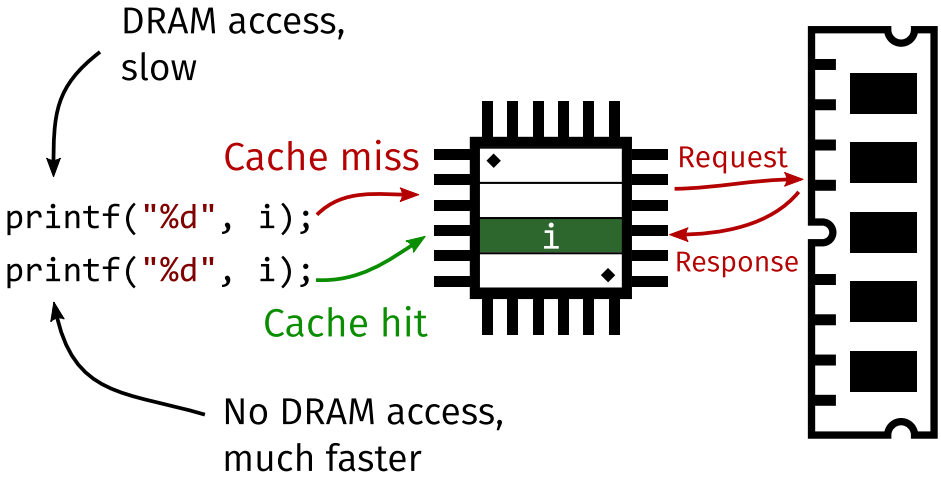

Cache miss
Request
printf("%d", i);
printf("%d", i);


Cache miss
printf("%d", i);
printf("%d", i);
Request
Response

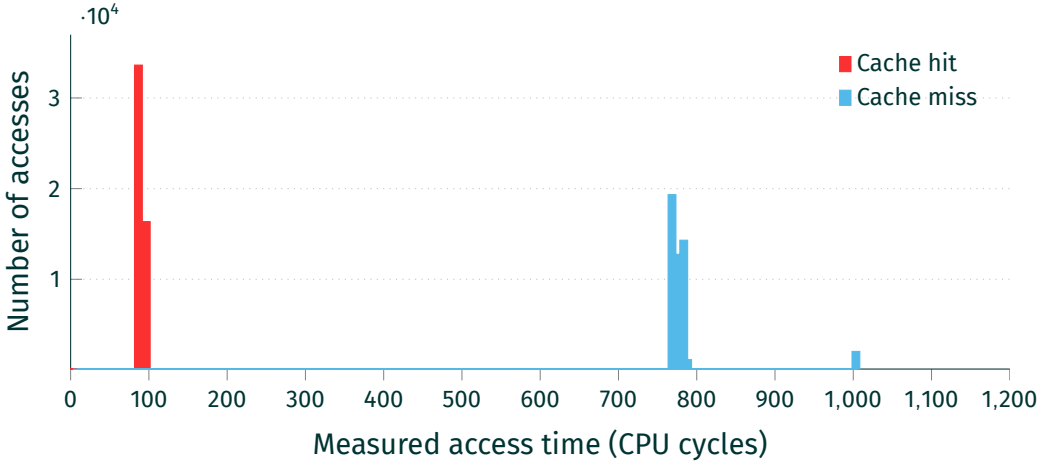

Cache miss
printf("%d", i);
printf("%d", i);
i
Request
Response


Cache miss
printf("%d", i);
printf("%d", i);
Cache hit
i
Request
Response


Cache miss
printf("%d", i);
printf("%d", i);
Cache hit
i
Request
Response
No DRAM access,
much faster


DRAM access,
slow
Cache miss
printf("%d", i);
printf("%d", i);
Cache hit
No DRAM access,
much faster
i
Request
Response

Memory Access Latency

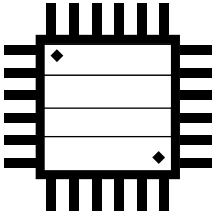


Shared Memory

ATTACKER

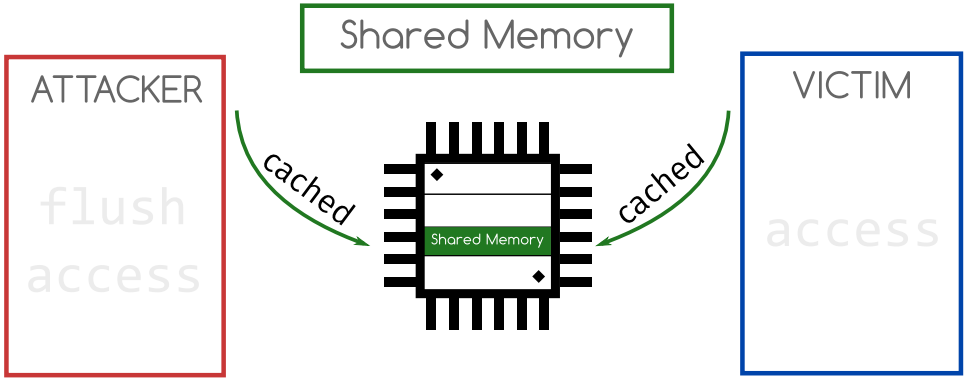
flush

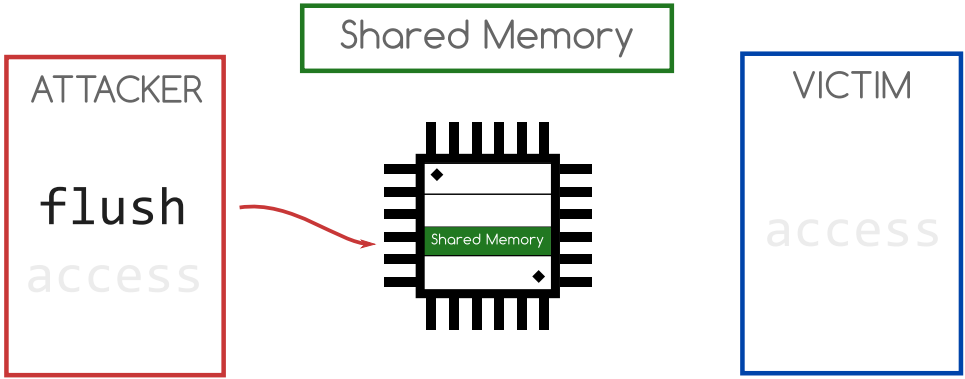
access

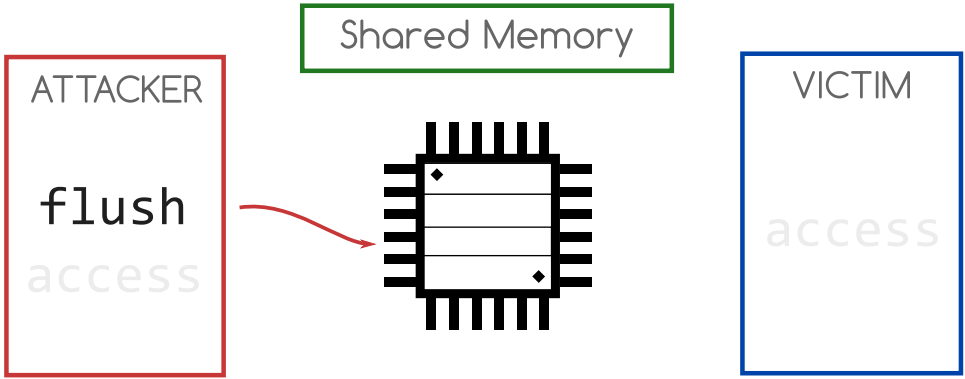


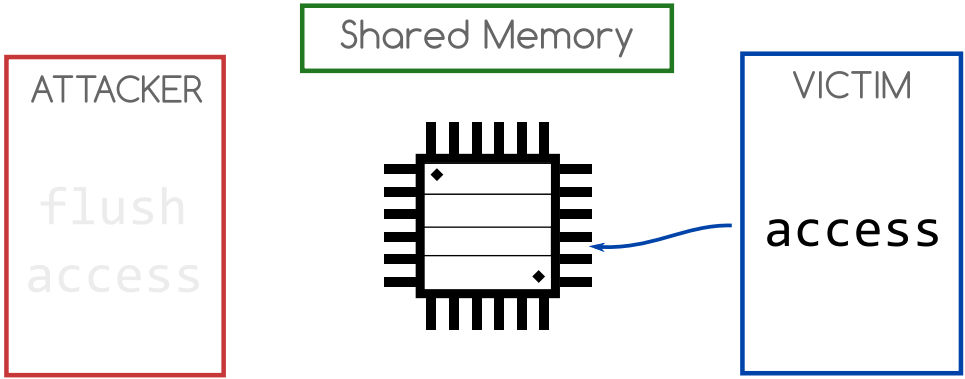
VICTIM

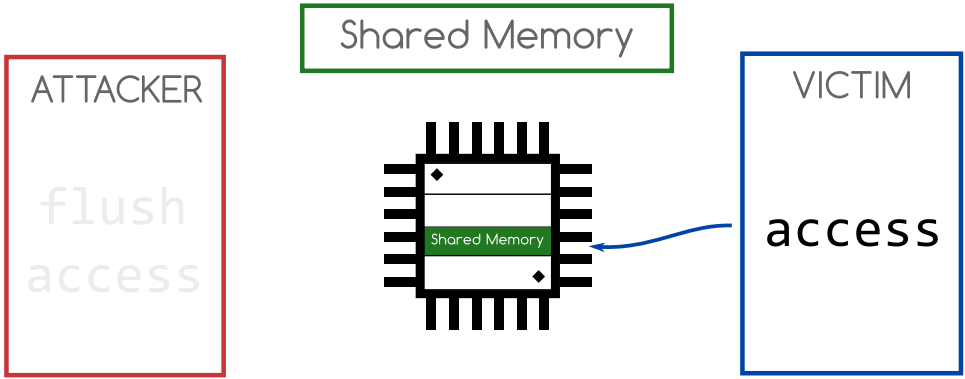
access

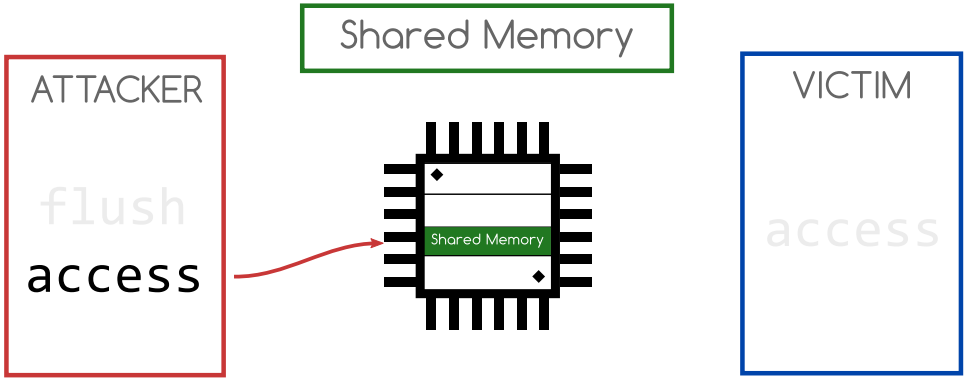

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

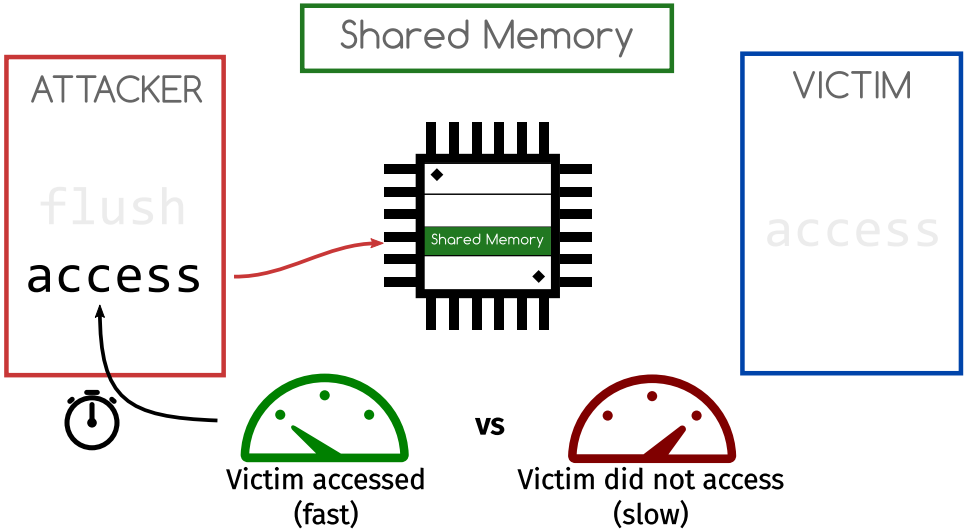

Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
VICTIM
access

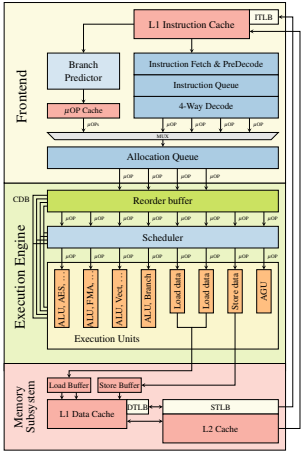

Shared Memory
ATTACKER
flush
access
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access


Shared Memory
ATTACKER
flush
access
Shared Memory
VICTIM
access
vs
Victim accessed
(fast)
Victim did not access
(slow)

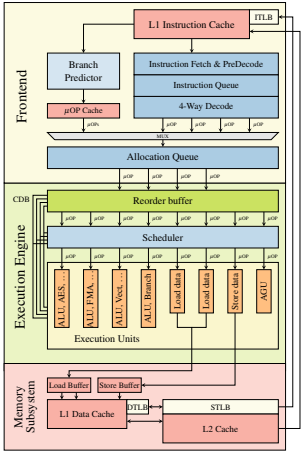
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end

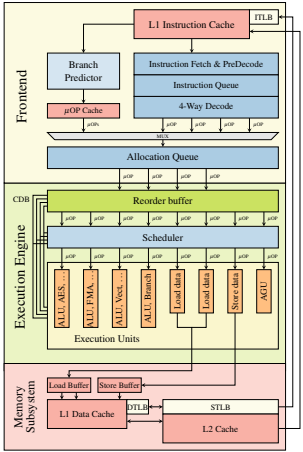
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end
- dispatched to the backend

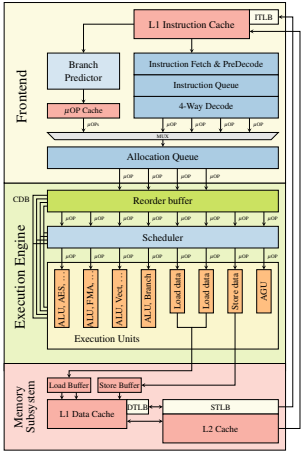
Out-of-Order Execution



Instructions are

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

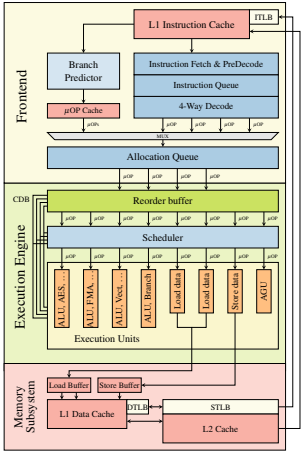
Out-of-Order Execution



Instructions

- are executed out-of-order

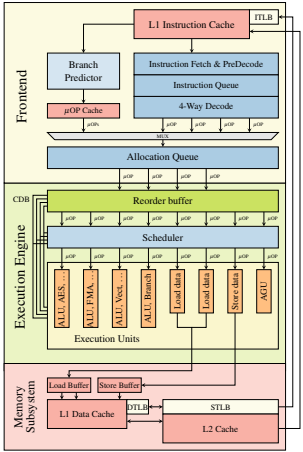
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready

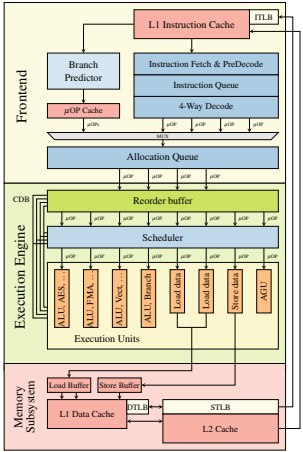
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions

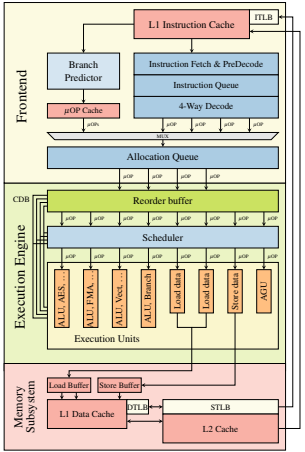
Out-of-Order Execution



Instructions

- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order

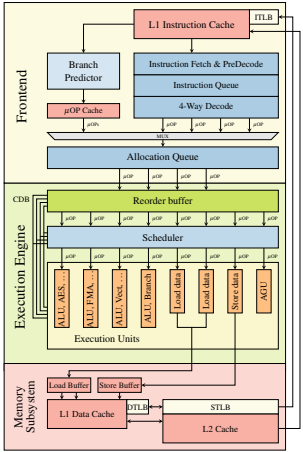
Out-of-Order Execution



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

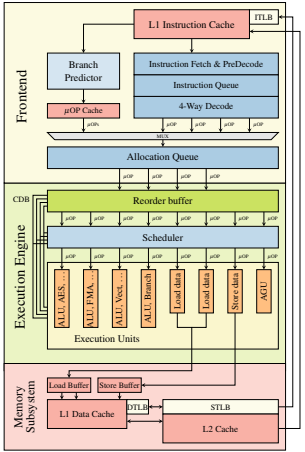
Out-of-Order Execution



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

Out-of-Order Execution

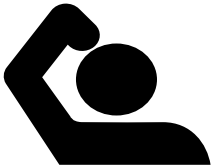


Instructions

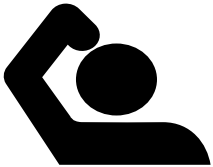
- are executed out-of-order
- wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- retire in-order
  - State becomes architecturally visible
- Exceptions are checked during retirement
  - Flush pipeline and recover state

Toy example

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

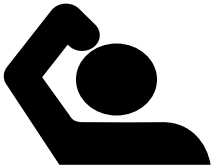


- Flush+Reload over all pages of the array



- Flush+Reload over all pages of the array

- “Unreachable” code line was actually executed



- Flush+Reload over all pages of the array

- “Unreachable” code line was actually executed
- Exception was only thrown afterwards

Building the Code

- Out-of-order instructions leave microarchitectural traces

Building the Code

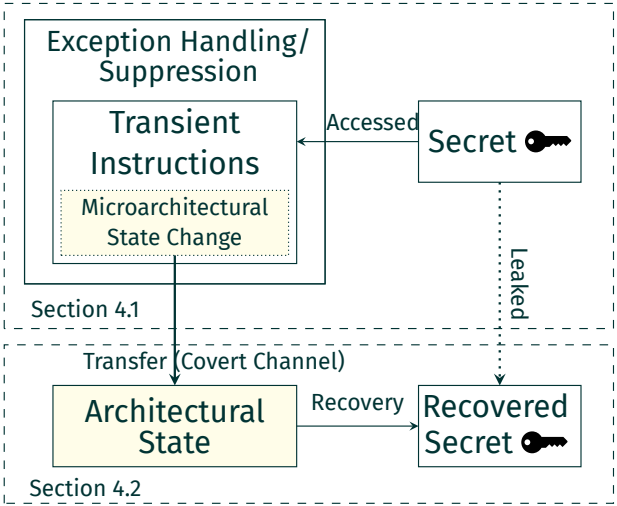
- Out-of-order instructions leave microarchitectural traces
  - We can see them for example in the cache

Building the Code

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

- 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

Building blocks



Executing transient instructions

- Transient instructions are executed all the time

Executing transient instructions

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

Executing transient instructions

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

Executing transient instructions

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

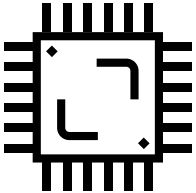
Fault Handling

Fault Suppression

Fault Prevention

Building a Covert Channel

- Transfer of the microarchitectural state into an architectural state
- 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 $\Rightarrow$ access different cache line
- Not limited to the cache

Building the Code

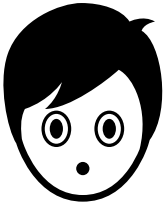
- Add another layer of indirection to test

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

- Add another layer of indirection to test

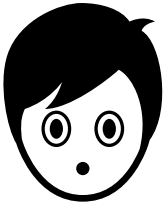
```
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```

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



- Flush+Reload over all pages of the array

- Index of cache hit reveals data



- Flush+Reload over all pages of the array

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

MELTDOWN

- Using out-of-order execution, we can read data at any address

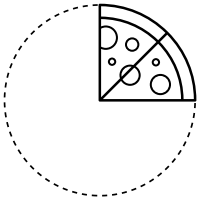
- Using out-of-order execution, we can read data at any address
- Index of cache hit reveals data

- 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

- 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

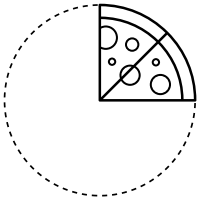
Demo

Uncached memory



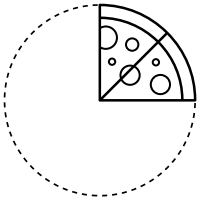
- Assumed that one can only read data stored in the L1 with Meltdown

Uncached memory



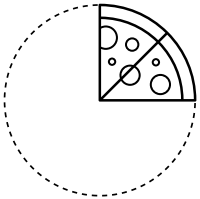
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value

Uncached memory



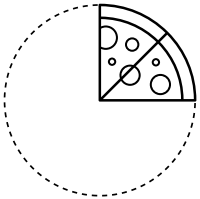
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core

Uncached memory



- Assumed that one can only read data stored in the L1 with Meltdown
- 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 can still leak the data at a lower reading rate

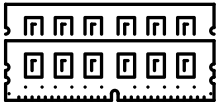
Uncached memory



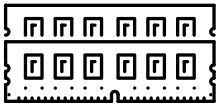
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate
- Meltdown might implicitly cache the data

Uncachable memory

- Mark pages in page tables as UC (uncachable)

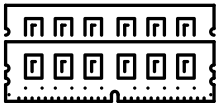


Uncachable memory

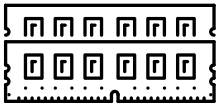


- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory

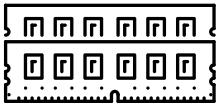
Uncachable memory



- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
- If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked

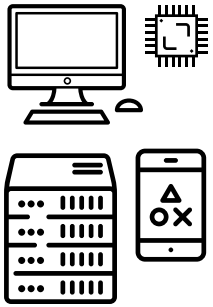


- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
- If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked
- Meltdown might read the data from one of the fill buffers



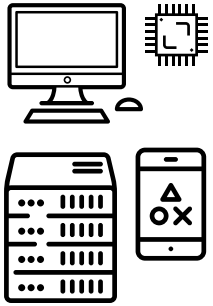
- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory
- If the attacker can trigger a legitimate load (system call, ...) on the same CPU core, the data still can be leaked
- Meltdown might read the data from one of the fill buffers
  - as they are shared between threads running on the same core

Affected by Meltdown



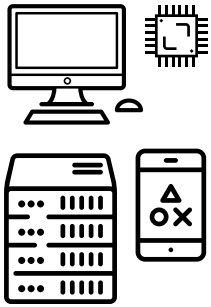
- **Intel**: Almost every CPU

Affected by Meltdown



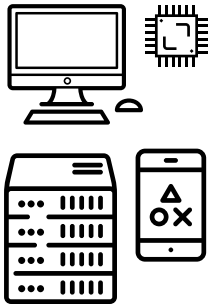
- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected

Affected by Meltdown



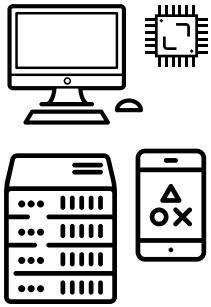
- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75

Affected by Meltdown



- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
- **IBM**: System Z, Power Architecture, POWER8 and POWER9

Affected by Meltdown



- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
- **IBM**: System Z, Power Architecture, POWER8 and POWER9
- **Apple**: All Mac and iOS devices

SAMSUNG GALAXY S7

- Exynos Mongoose M1 CPU Architecture

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction
  - Full Out-of-Order Instruction Execution

Samsung Galaxy S7

Samsung Galaxy S7

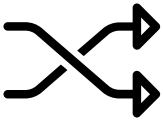
- Exynos Mongoose M1 CPU Architecture
  - Custom CPU core in the Exynos 8 Octa (8890)
  - Perceptron Branch Prediction
  - Full Out-of-Order Instruction Execution
    - Full Out-of-Order loads and stores

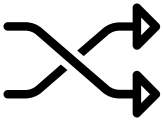
- Dumping the entire physical memory takes some time

- 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

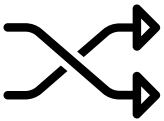
- Dumping the entire physical memory takes some time
  - L1: 582 KB/s
  - L3: 12.4 KB/s
  - Uncached: 10 B/s (improved: 3.2 KB/s)
- Not very practical in most scenarios

- De-randomize KASLR to access internal kernel structures

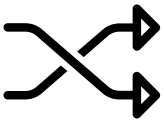




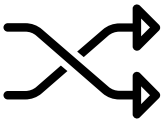
- De-randomize KASLR to access internal kernel structures
- Locate a known value inside the kernel, e.g., Linux banner



- De-randomize KASLR to access internal kernel structures
- Locate a known value inside the kernel, e.g., Linux banner
  - Start at the default address according to the symbol table of the running kernel

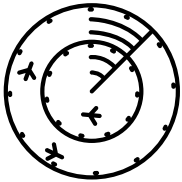


- De-randomize KASLR to access internal kernel structures
- Locate a known value inside the kernel, e.g., Linux banner
  - Start at the default address according to the symbol table of the running kernel
  - Linux KASLR has an entropy of 6 bits $\Rightarrow$ only 64 possible randomization offsets



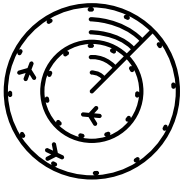
- De-randomize KASLR to access internal kernel structures
- Locate a known value inside the kernel, e.g., Linux banner
  - Start at the default address according to the symbol table of the running kernel
  - Linux KASLR has an entropy of 6 bits $\Rightarrow$ only 64 possible randomization offsets
- Difference between the found address and the non-randomized base address is the randomization offset

Locating the victim process



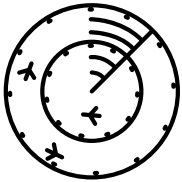
- Linux manages all processes in a linked list

Locating the victim process



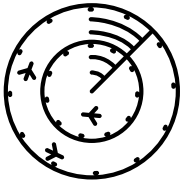
- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure

Locating the victim process



- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure
  - At a fixed offset that varies only among kernel builds

Locating the victim process



- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure
  - At a fixed offset that varies only among kernel builds
- Each task list structure contains a pointer to the next task and
  - PID of the task
  - name of the task
  - Root of the multi-level page table

Dumping memory content

- Resolve physical address using paging structures

Dumping memory content

- Resolve physical address using paging structures
- Read the content using the direct-physical map

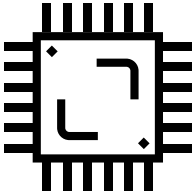
Dumping memory content

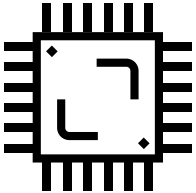
- Resolve physical address using paging structures
- Read the content using the direct-physical map
- Enumerate all mapped pages and dump entire process memory

Dumping memory content

- Resolve physical address using paging structures
- Read the content using the direct-physical map
- Enumerate all mapped pages and dump entire process memory
- Location of the key known, we can just dump the key directly

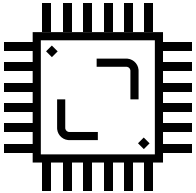
- Problem is rooted in hardware





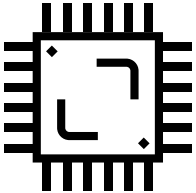
- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them

Meltdown Mitigation



- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them
- Hard split of user space and kernel space
  - New bit in control register

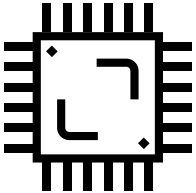
Meltdown Mitigation



- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them
- Hard split of user space and kernel space
  - New bit in control register

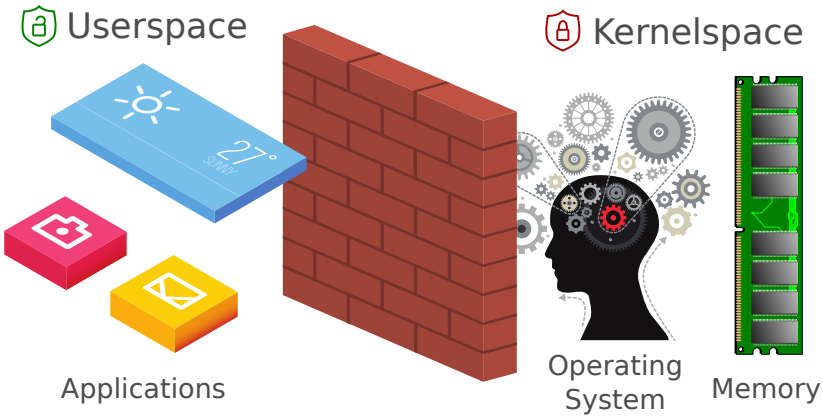
- Fix the hardware → long-term solution

Meltdown Mitigation

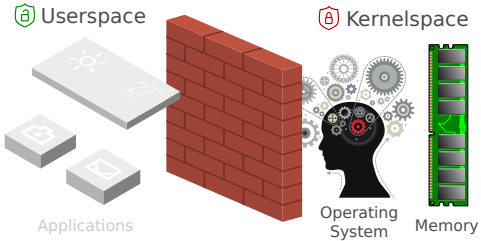


- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them
- Hard split of user space and kernel space
  - New bit in control register

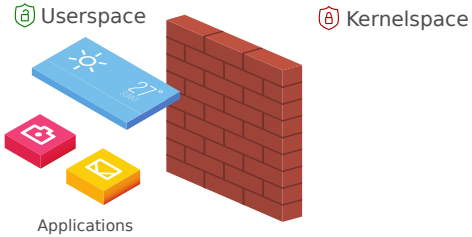
- Fix the hardware → long-term solution
- Can we fix it in software?

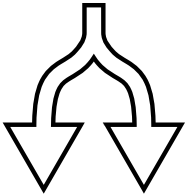

Userspace
Kernelspace
27°
SUNNY
Applications
Operating System
Memory

Kernel View

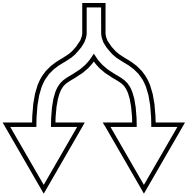


User View

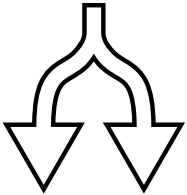




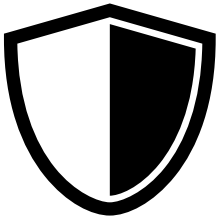
- KAISER [Gru+17] has been published in May 2017 ...



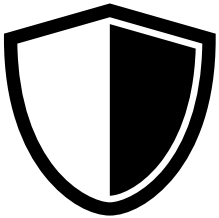
- KAISER [Gru+17] has been published in May 2017 ...
- ...as a countermeasure against other side-channel attacks



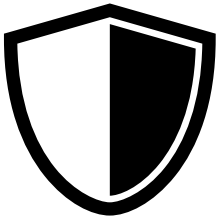
- KAISER [Gru+17] has been published in May 2017 ...
- ...as a countermeasure against other side-channel attacks
- Inadvertently defeats Meltdown as well



- **Linux**: Kernel Page-table Isolation (KPTI)



- **Linux**: Kernel Page-table Isolation (KPTI)
- **Apple**: Released updates



- **Linux**: Kernel Page-table Isolation (KPTI)
- **Apple**: Released updates
- **Windows**: Kernel Virtual Address (KVA) Shadow

You can find our proof-of-concept implementation on:

- `https://github.com/IAIK/meltdown`

Conclusion

- Underestimated microarchitectural attacks 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

Meltdown

Reading Kernel Memory from User Space

Moritz Lipp[1], Michael Schwarz[1], Daniel Gruss[1], Thomas Prescher[2], Werner Haas[2], Anders Fogh[3], Jann Horn[4], Stefan Mangard[1], Paul Kocher[5], Daniel Genkin[6], Yuval Yarom[7], Mike Hamburg[8]

27th USENIX Security Symposium - August 16, 2018

[1]Graz University of Technology, [2]Cyberus Technology GmbH, [3]G-Data Advanced Analytics, [4]Google Project Zero, [5]Independent (www.paulkocher.com), [6]University of Michigan, [7]University of Adelaide & Data61, [8]Rambus, Cryptography Research Division