Moritz Lipp (@mlqxyz)

Michael Schwarz (@misc0110)

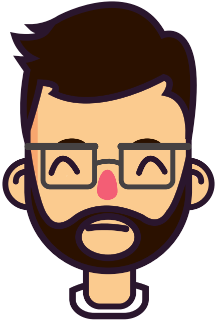
Daniel Gruss (@lavados)

Meltdown

Basics, Details, Consequences

Black Hat USA 2018
9 August, 2018 - Las Vegas, NV, USA

Who am I?



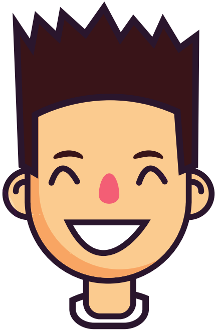
Moritz Lipp

PhD student @ Graz University of Technology

🐦 @mlqxyz

✉ moritz.lipp@iaik.tugraz.at

Who am I?



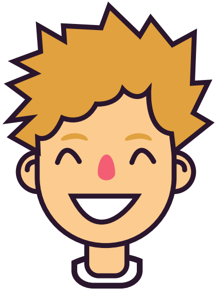
Michael Schwarz

PhD student @ Graz University of Technology

🐦 @misc0110

✉ michael.schwarz@iaik.tugraz.at

Who am I?



Daniel Gruss

PostDoc @ Graz University of Technology

🐦 @lavados

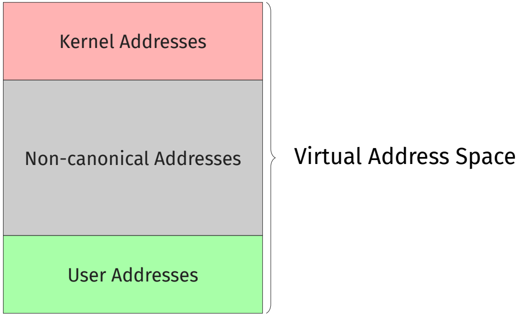
✉ daniel.gruss@iaik.tugraz.at

Team

- Anders Fogh
- Daniel Genkin
- Werner Haas
- Mike Hamburg
- Jann Horn
- Paul Kocher
- Stefan Mangard
- Thomas Prescher
- Yuval Yarom

Let’s Read Kernel Memory from User Space!

Virtual Memory

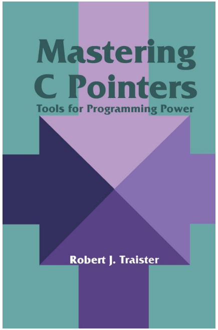


Building the Code

- 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);
```

Building the Code

- Compile and run

Building the Code

- Compile and run

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

Building the Code

- Compile and run

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

- Kernel addresses are of course not accessible

Building the Code

- 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

Building the Code

- Just catch the segmentation fault!

Building the Code

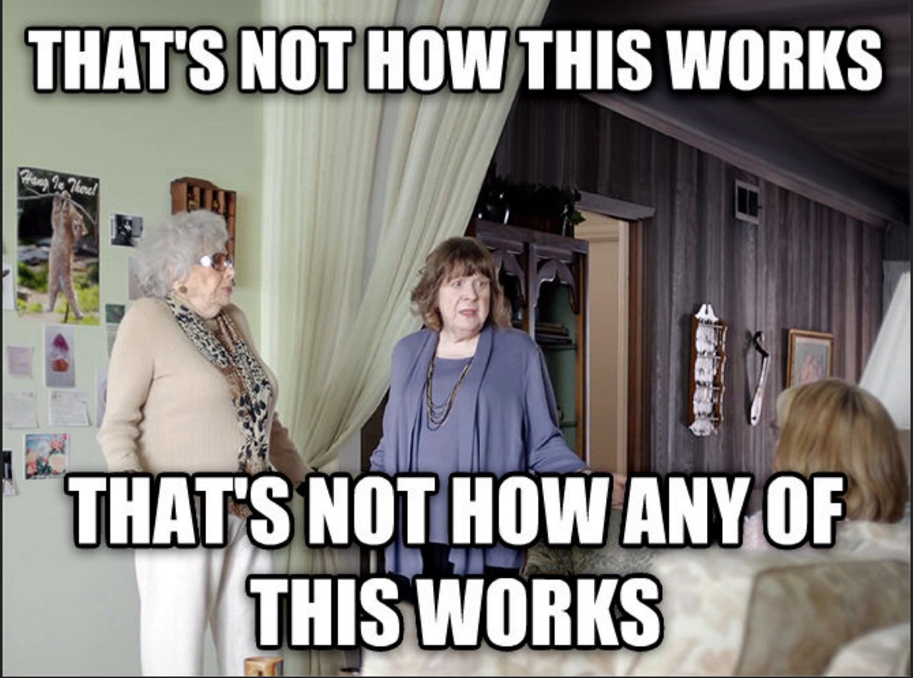
- Just catch the segmentation fault!
- We can simply install a signal handler

Building the Code

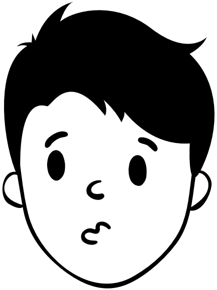
- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue

Building the Code

- 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

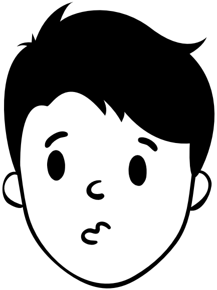

THAT'S NOT HOW THIS WORKS
Hang In There!
THAT'S NOT HOW ANY OF THIS WORKS

Building the Code



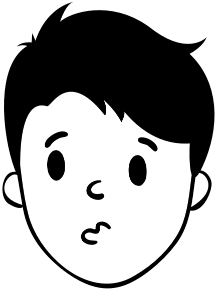
- Still no kernel memory

Building the Code



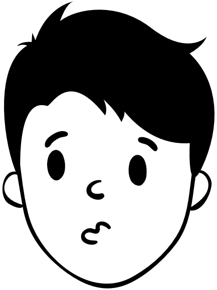
- Still no kernel memory
- Privilege checks seem to work

Building the Code



- Still no kernel memory
- Privilege checks seem to work
- Maybe it is not that straight forward

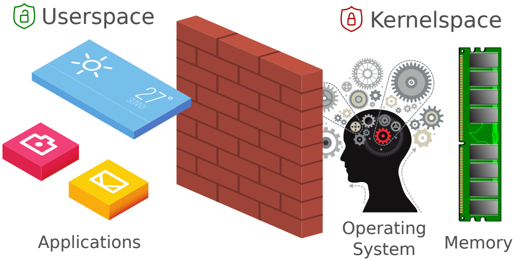
Building the Code



- Still no kernel memory
- Privilege checks seem to work
- Maybe it is not that straight forward
- Back to the drawing board

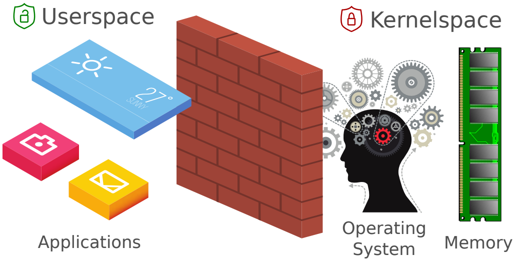
Operating Systems 101

Memory Isolation



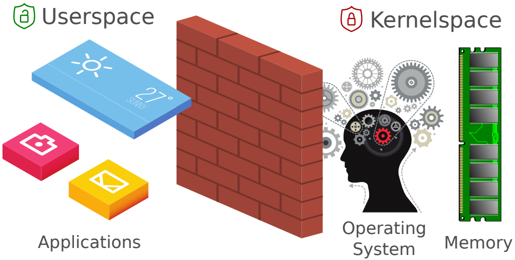
- Kernel is isolated from user space

Memory Isolation



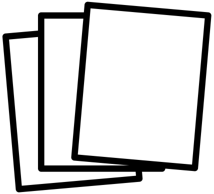
- Kernel is isolated from user space
- This isolation is a combination of hardware and software

Memory Isolation



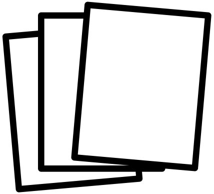
- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel

Paging



- CPU support virtual address spaces to isolate processes

Paging

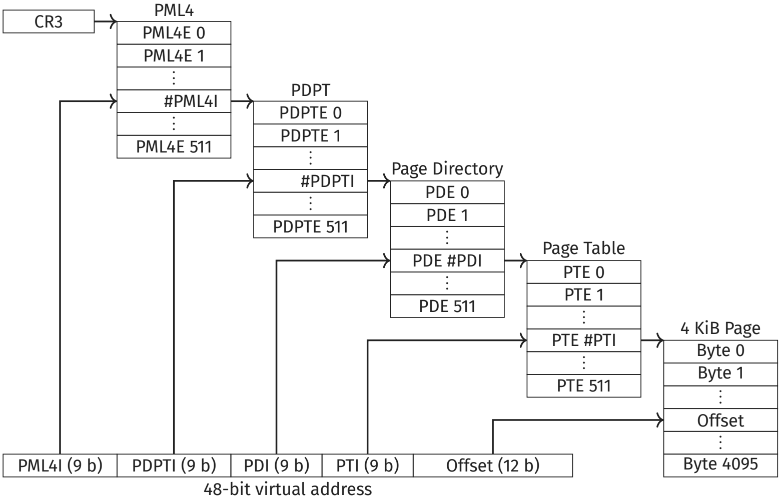


- CPU support virtual address spaces to isolate processes
- Physical memory is organized in page frames

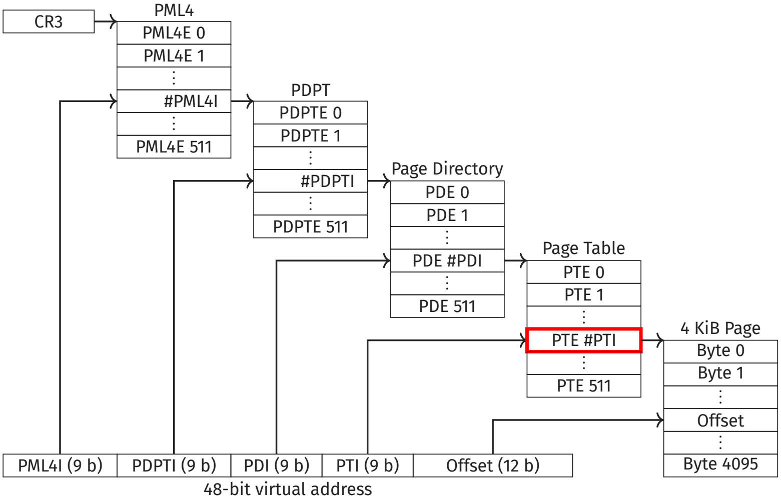
Paging

- 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

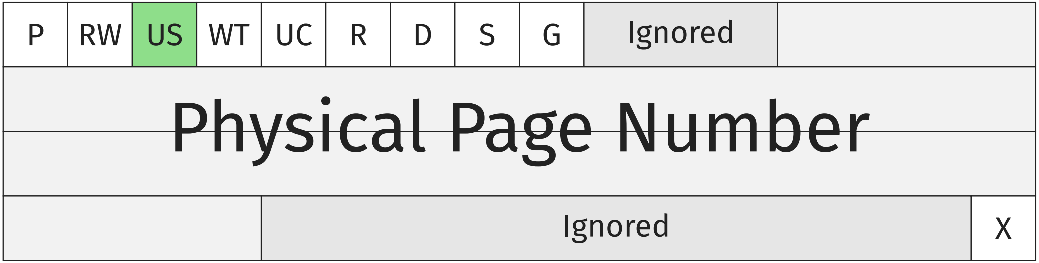
Address Translation on x86-64



Address Translation on x86-64



Page Table Entry



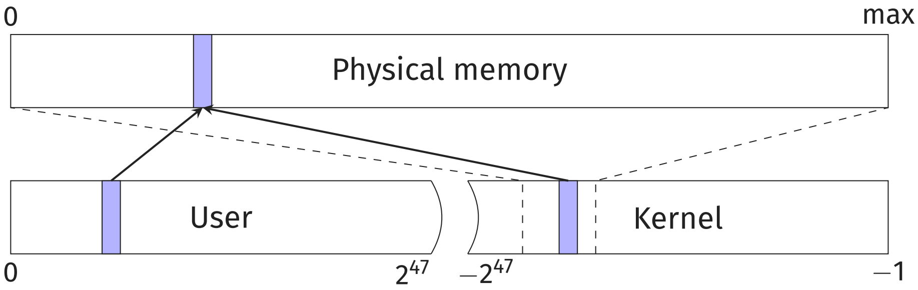
P | RW | US | WT | UC | R | D | S | G | Ignored

Physical Page Number

Ignored | X

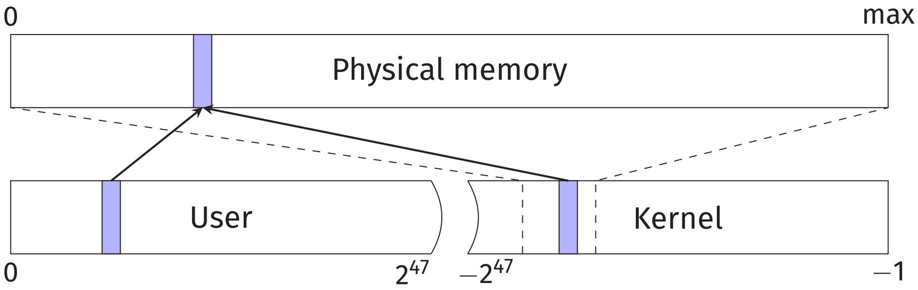
- User/Supervisor bit defines in which privilege level the page can be accessed

Direct-physical map



- Kernel is typically mapped into every address space

Direct-physical map

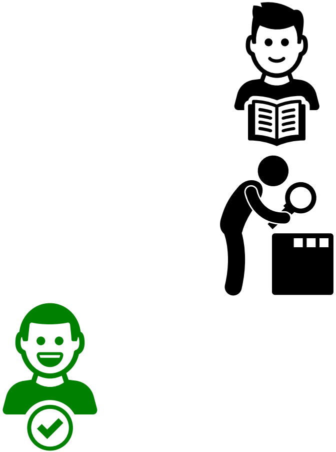


- Kernel is typically mapped into every address space
- Entire physical memory is mapped in the kernel

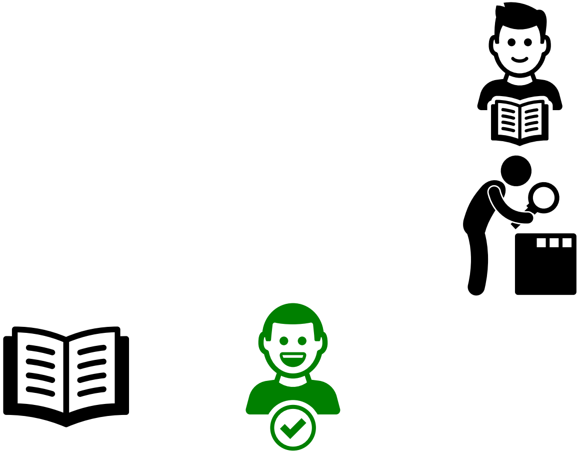
Loading an address

Loading an address

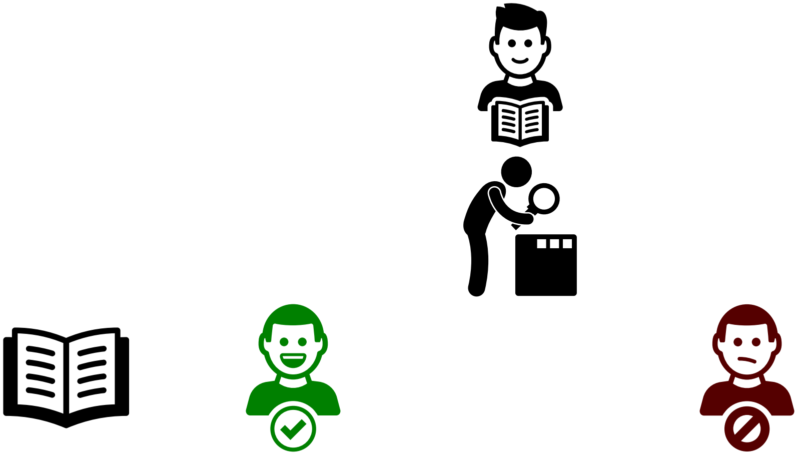
Loading an address



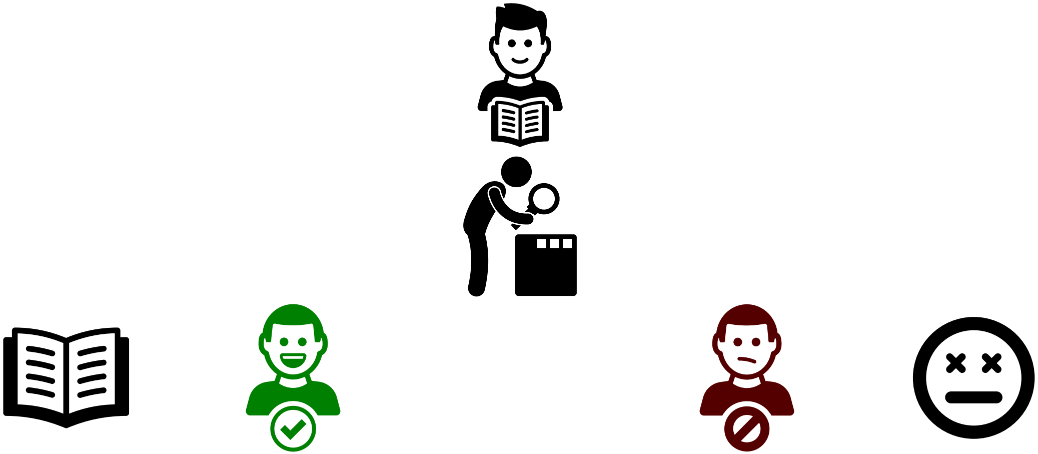
Loading an address



Loading an address



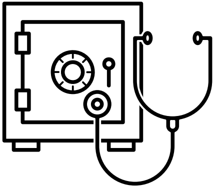
Loading an address



Side-channel Attacks

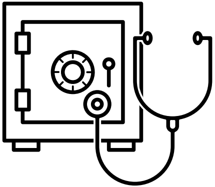
Side-channel Attacks

- Safe software infrastructure does not mean safe execution

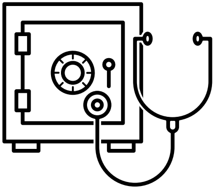


Side-channel Attacks

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

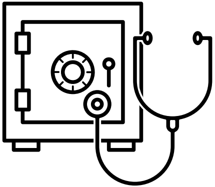


Side-channel Attacks



- 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

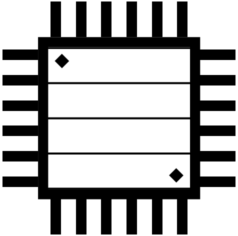
CPU caches

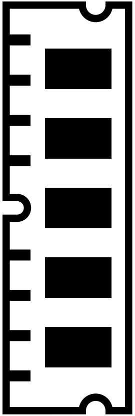
●●●

Caches and Cache Attacks

CPU Cache

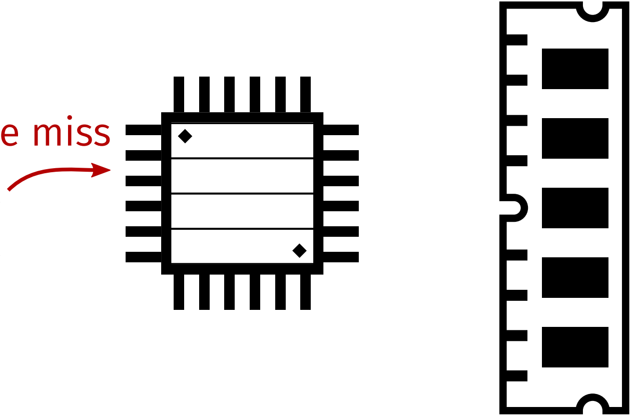
```
printf("%d", i);
printf("%d", i);
```





CPU Cache

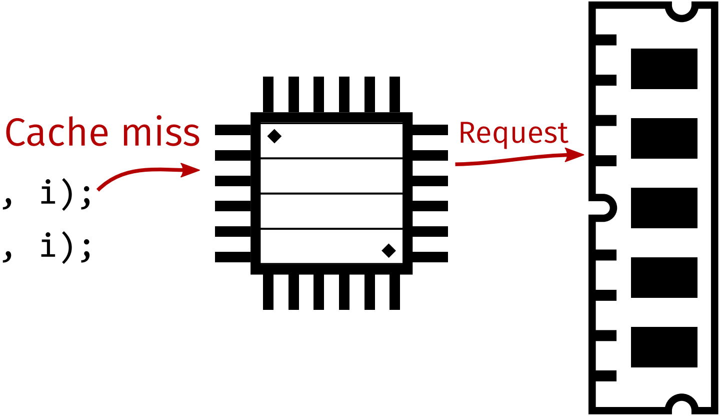
```
printf("%d", i);
printf("%d", i);
```



Cache miss

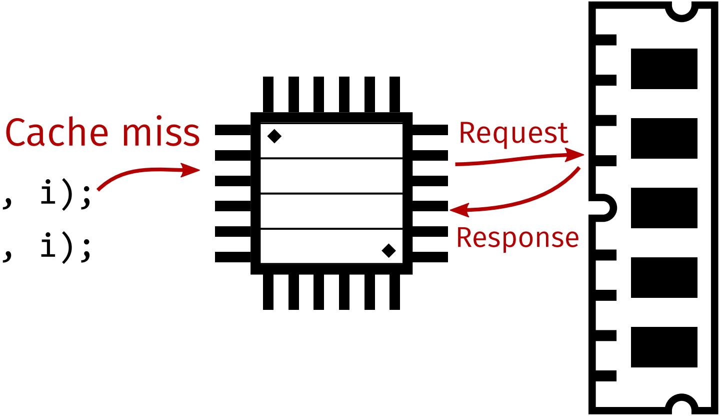
CPU Cache

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



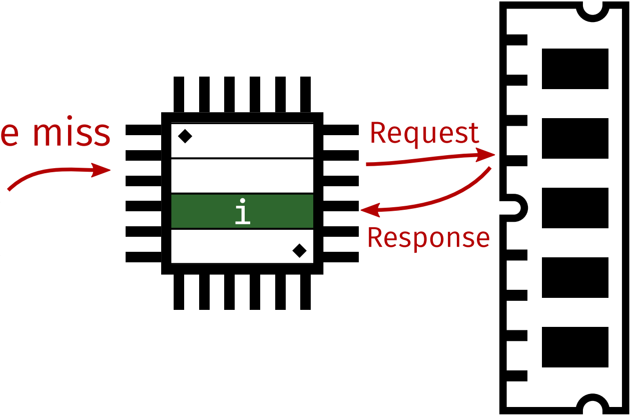
CPU Cache

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



CPU Cache

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



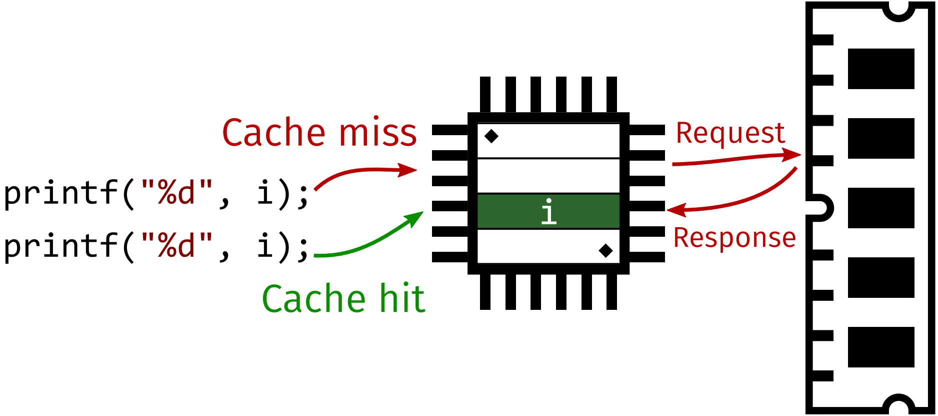
Cache miss

Request

Response

i

CPU Cache



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

Cache miss

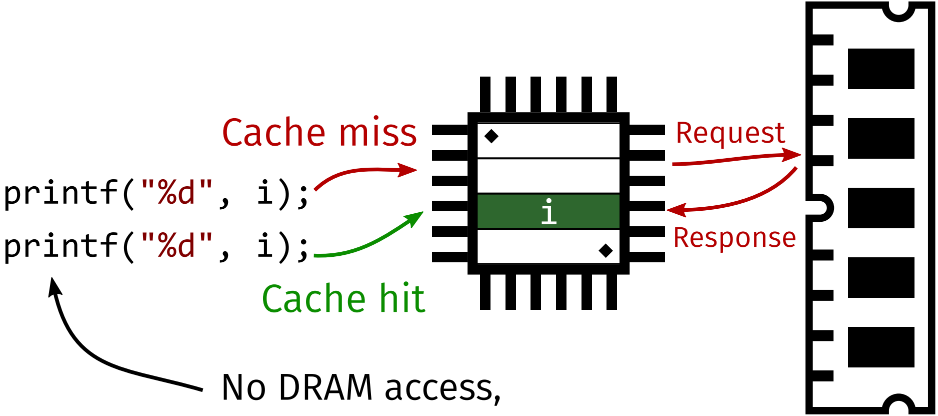
Cache hit

i

Request

Response

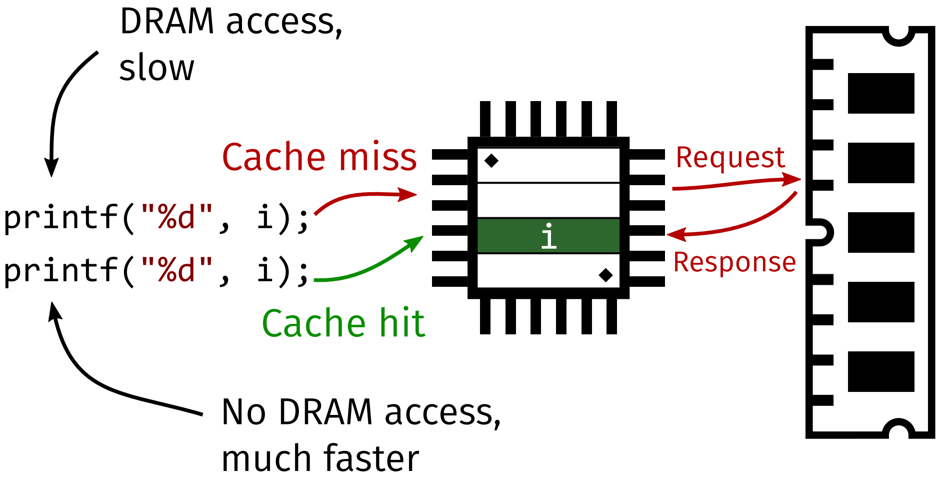
CPU Cache



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

No DRAM access,
much faster

CPU Cache



DRAM access, slow

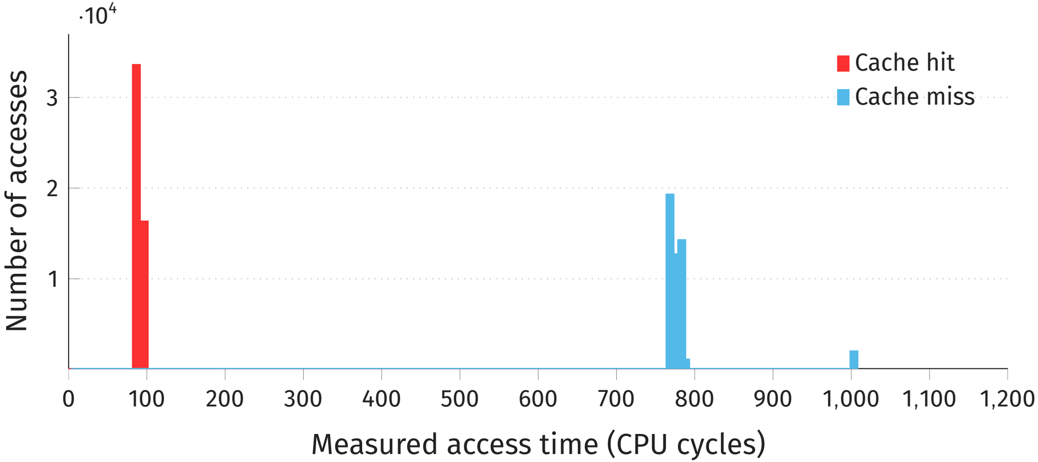
```
printf("%d", i);
printf("%d", i);
```

Cache miss

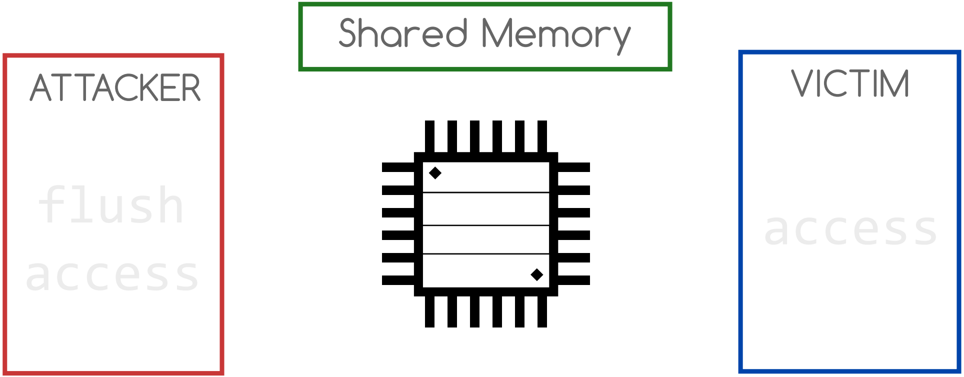
Cache hit

No DRAM access, much faster

Memory Access Latency



Flush+Reload



Shared Memory

ATTACKER

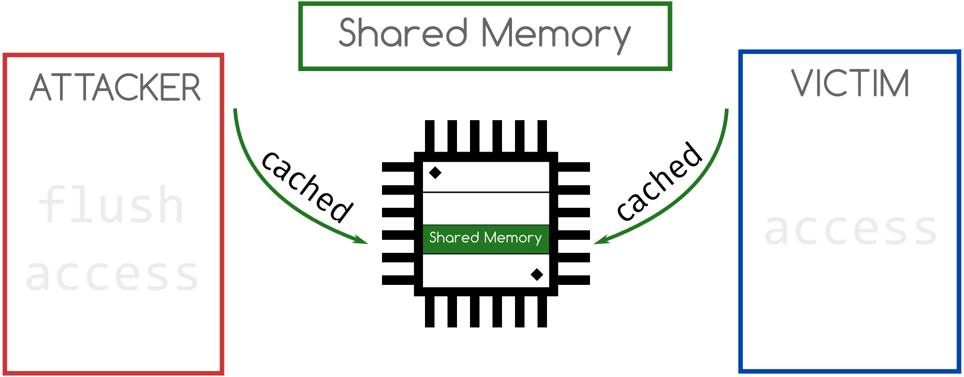
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

access

cached

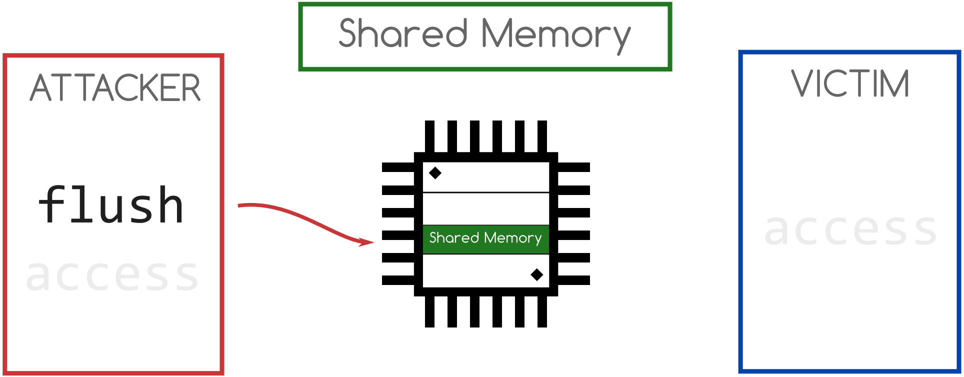
Shared Memory

cached

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

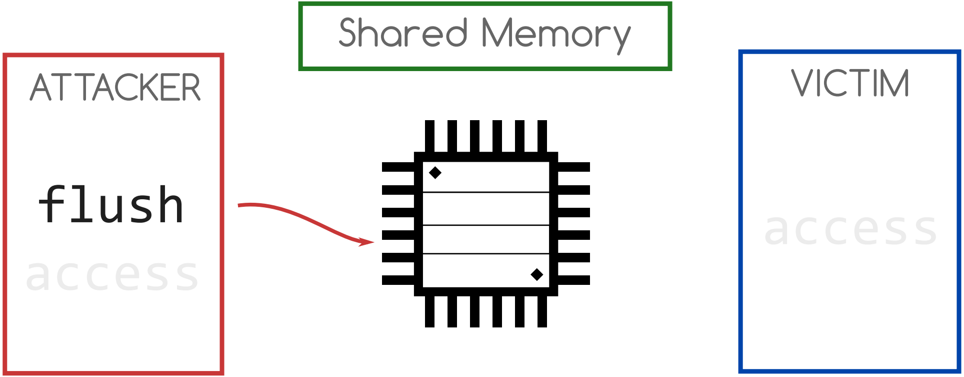
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

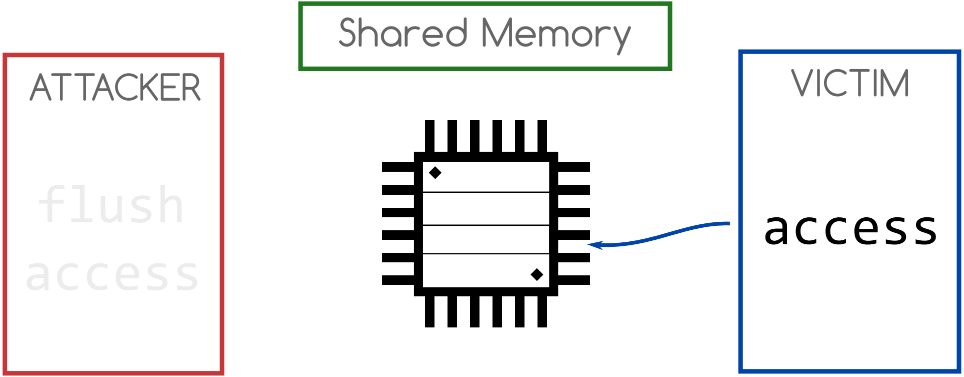
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

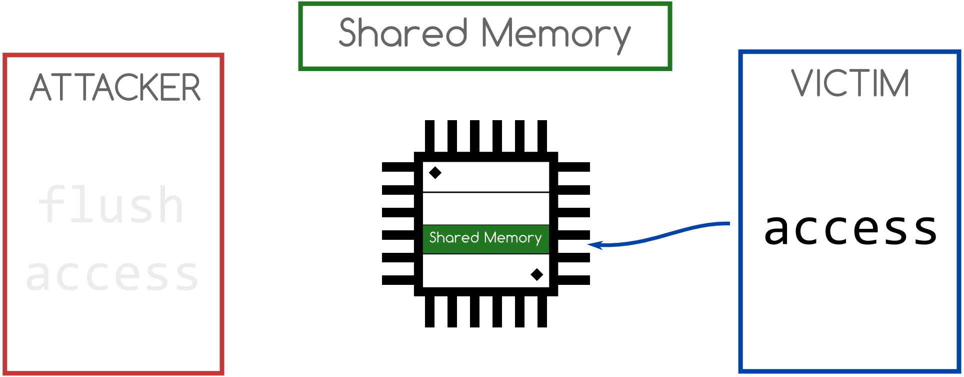
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

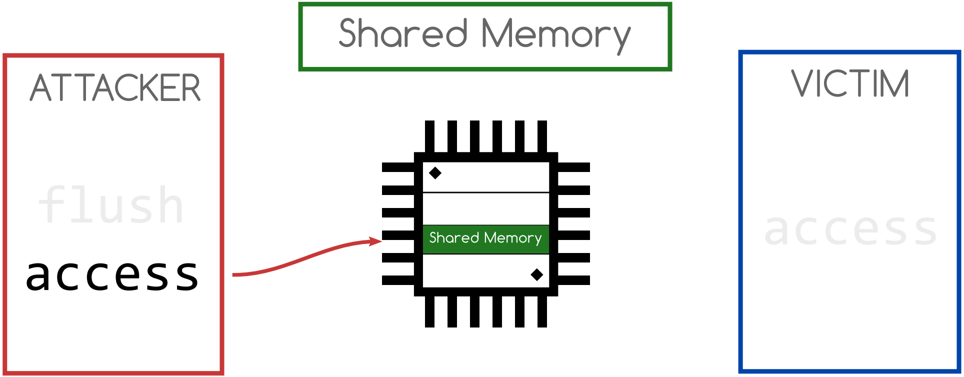
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

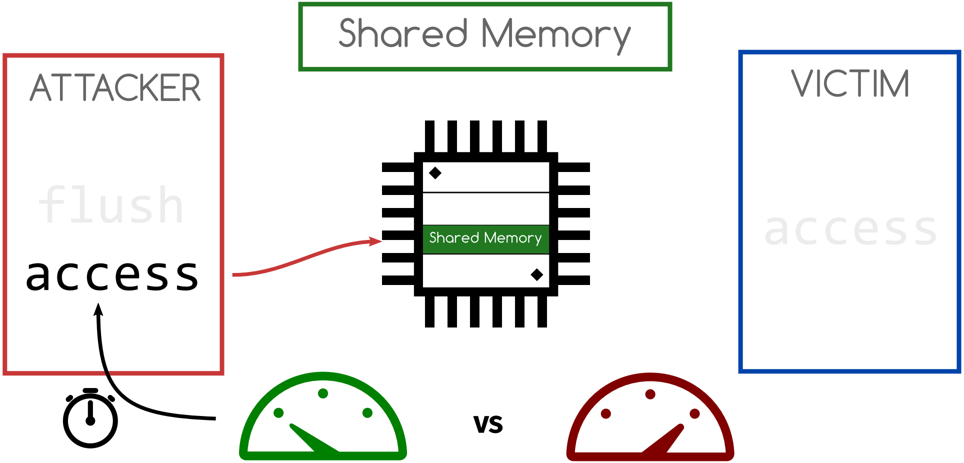
flush

access

VICTIM

access

Flush+Reload



Shared Memory

ATTACKER

flush

access

Shared Memory

VICTIM

access

vs

Victim accessed (fast)

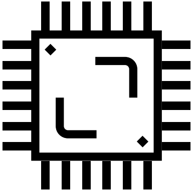
Victim did not access (slow)

Microarchitecture

Architecture and Microarchitecture

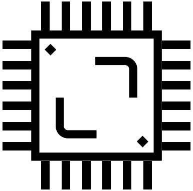
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)

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

Architecture and Microarchitecture



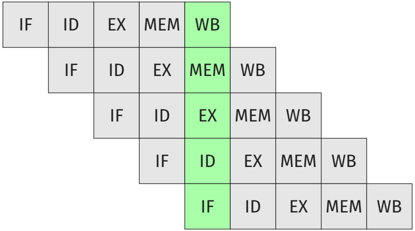
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, ...)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

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

●●●

In-Order Execution



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

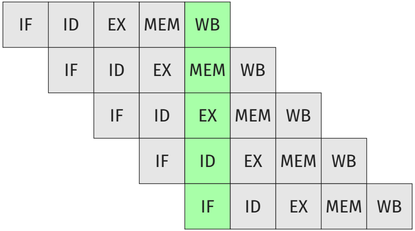
- Instructions are...
  - fetched (IF) from the L1 Instruction Cache

In-Order Execution

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache
  - decoded (`ID`)

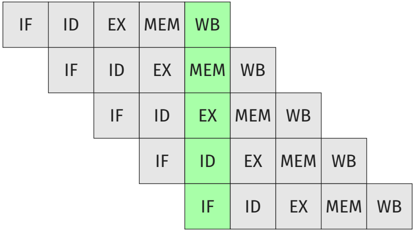
In-Order Execution



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- Instructions are...
  - fetched (IF) from the L1 Instruction Cache
  - decoded (ID)
  - executed (EX) by execution units

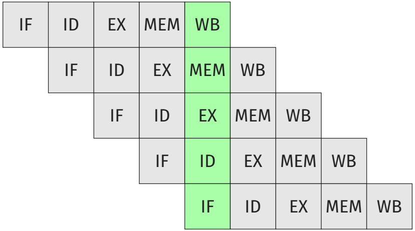
In-Order Execution



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache
  - decoded (`ID`)
  - executed (`EX`) by execution units
- Memory access is performed (`MEM`)

In-Order Execution



IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache
  - decoded (`ID`)
  - executed (`EX`) by execution units
- Memory access is performed (`MEM`)
- Architectural register file is updated (`WB`)

In-Order Execution

- Instructions are executed in-order

In-Order Execution

- Instructions are executed in-order
- Pipeline stalls when stages are not ready

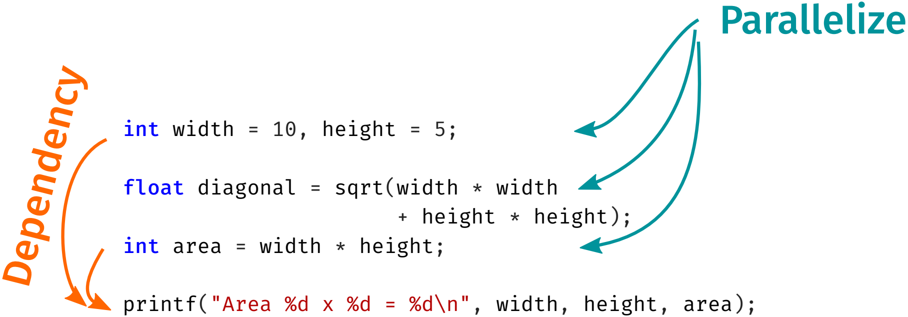
In-Order Execution

- Instructions are executed in-order
- Pipeline stalls when stages are not ready
- If data is not cached, we need to wait

Out-of-order Execution

```c
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

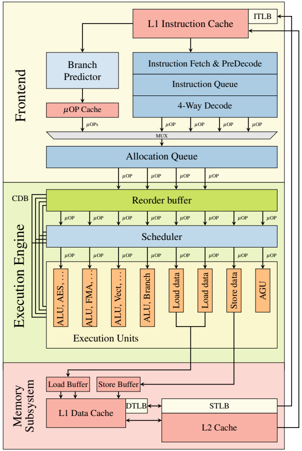


Parallelize

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);
```

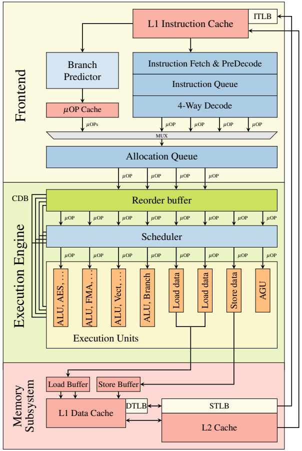
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end

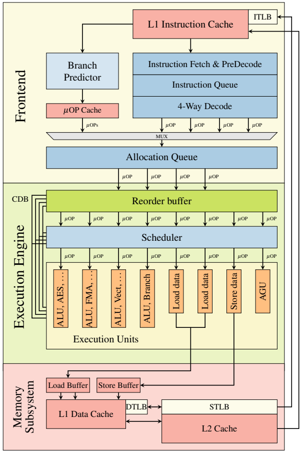
Out-of-Order Execution



Instructions are

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

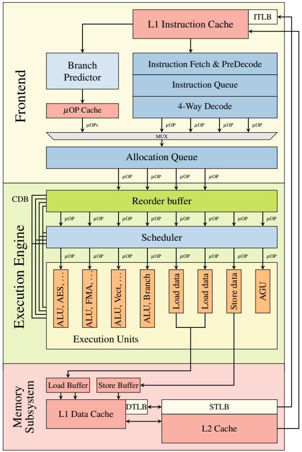
Out-of-Order Execution



Instructions are

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

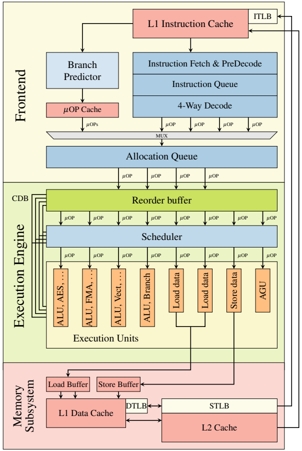
Out-of-Order Execution



Instructions

- are executed out-of-order

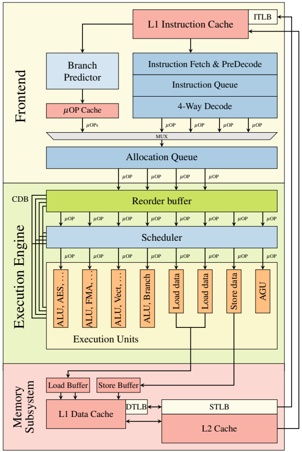
Out-of-Order Execution



Instructions

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

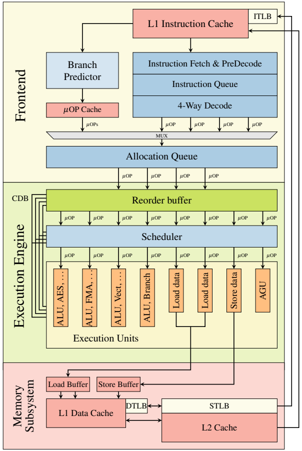
Out-of-Order Execution



Instructions

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

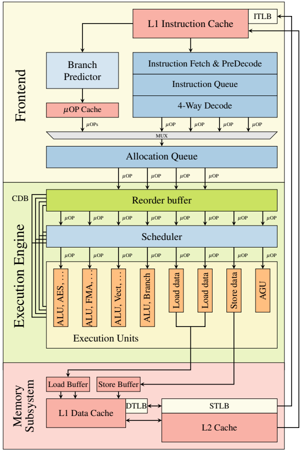
Out-of-Order Execution



Instructions

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

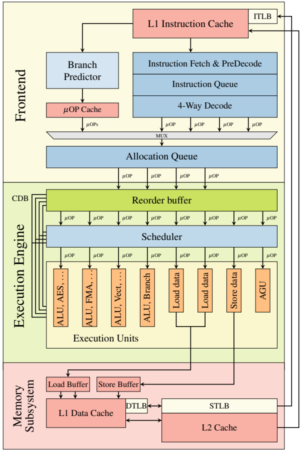
Out-of-Order Execution



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

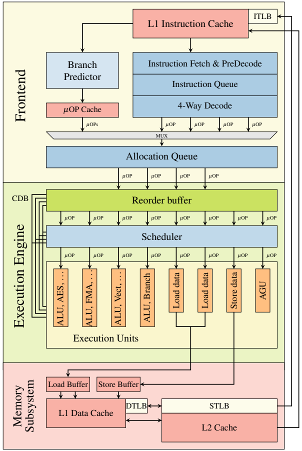
Out-of-Order Execution



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

The state does not become architecturally visible but ...

The state does not become architecturally visible but ...

Building the Code

- New code

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

Building the Code

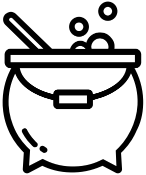
- New code

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

- `volatile` because compiler was not happy

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

Building the Code



- New code

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

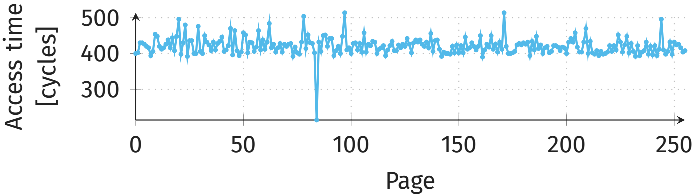
- `volatile` because compiler was not happy

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

- Static code analyzer is still not happy

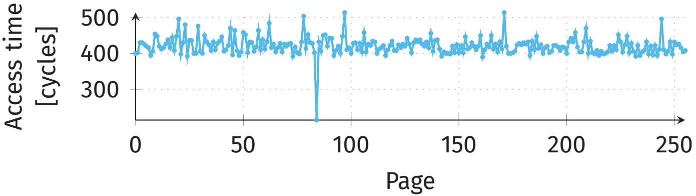
```
warning: Dereference of null pointer
        *(volatile char*)0;
```

- Flush+Reload over all pages of the array



Building the Code

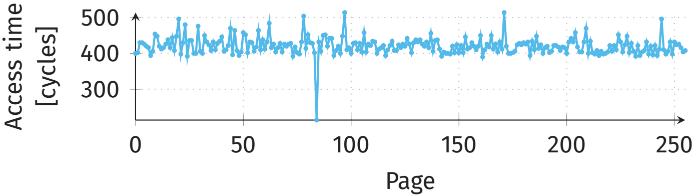
- Flush+Reload over all pages of the array



- “Unreachable” code line was actually executed

Building the Code

- 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

Building the Code

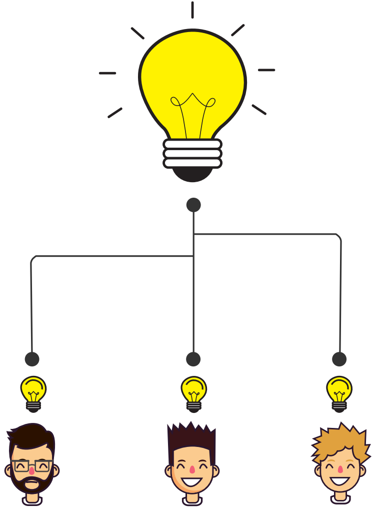
- Out-of-order instructions leave microarchitectural traces
  - We can see them for example in the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions

Building the Code

December

3

Building the Code



Building the Code

- Add another layer of indirection to test

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

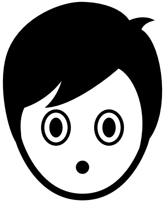
Building the Code

- Add another layer of indirection to test

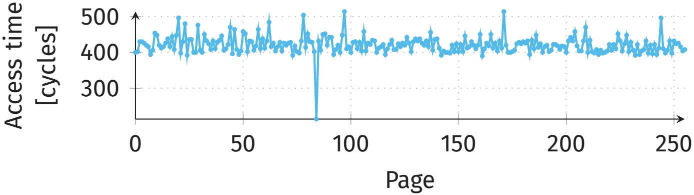
```
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```

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

Building the Code

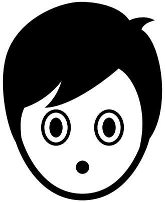


- Flush+Reload over all pages of the array

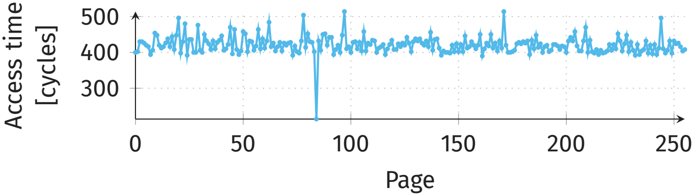


- Index of cache hit reveals data

Building the Code



- Flush+Reload over all pages of the array



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

MELTDOWN

Meltdown

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

Meltdown

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

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

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

I SHIT YOU NOT
THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL
STRONGBOW
memegenerator.net

Demo

Details: Exception Handling

- Basic Meltdown code leads to a crash (segfault)

Details: Exception Handling

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

Details: Exception Handling

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

Fault Handling

Fault Suppression

Fault Prevention

Meltdown with Fault Suppression

- Intel TSX to suppress exceptions instead of signal handler

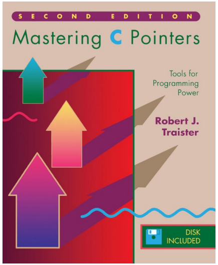
```c
if(xbegin() == XBEGIN_STARTED) {
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}

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

Meltdown with Fault Prevention

- Speculative execution to prevent exceptions

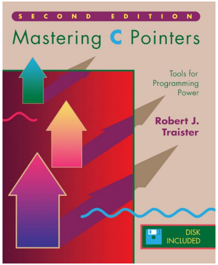
```
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
                 ((size_t)&zero * (1 - speculate));
if(!speculate) {
  char secret = *(char*) address;
  array[secret * 4096] = 0;
}

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

Make it faster



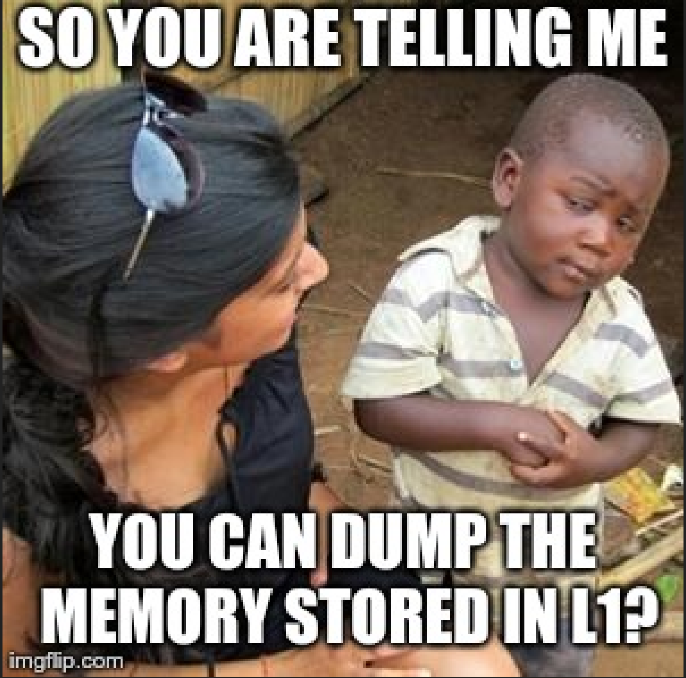
- Improve the performance with a NULL pointer dereference

Make it faster



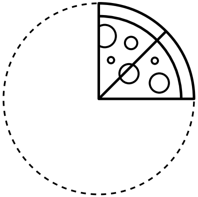
- Improve the performance with a NULL pointer dereference

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


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

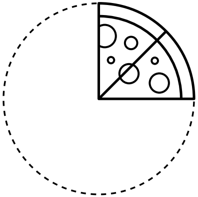
WHAT IF I TOLD YOU
YOU CAN LEAK THE ENTIRE MEMORY
imgflip.com

Uncached memory



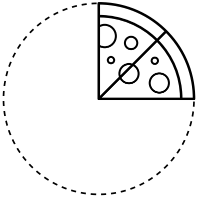
- Assumed that one can only read data stored in the L1 with Meltdown

Uncached memory



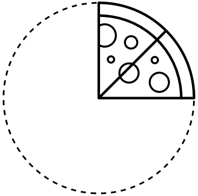
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value

Uncached memory



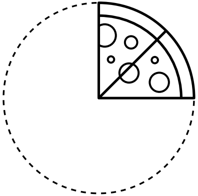
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core

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

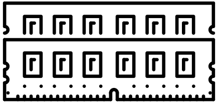


- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core
- We can still leak the data at a lower reading rate
- Meltdown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

Uncachable memory

- Mark pages in page tables as UC (uncachable)

Uncachable memory

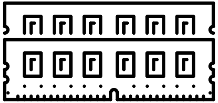


- Mark pages in page tables as UC (uncachable)
  - Every read or write operation will go to main memory

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

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
- Meltdown might read the data from one of the fill buffers

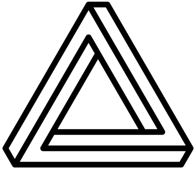
Uncachable memory

- 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
- Meltdown might read the data from one of the fill buffers
  - as they are shared between threads running on the same core

So you can dump the entire memory.

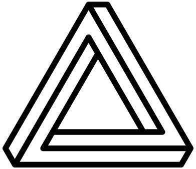
So you can dump the entire memory. But it takes ages?

Practical attacks



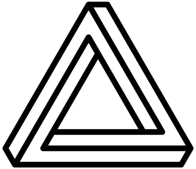
- Dumping the entire physical memory takes some time

Practical attacks



- Dumping the entire physical memory takes some time
  - Not very practical in most scenarios

Practical attacks



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

VeraCrypt

- Open-source utility for disk encryption

VeraCrypt

- Open-source utility for disk encryption
- Fork of TrueCrypt

VeraCrypt

- Open-source utility for disk encryption
- Fork of TrueCrypt
- Cryptographic keys are stored in RAM

VeraCrypt

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

Demo

Breaking KASLR

- De-randomize KASLR to access internal kernel structures

Breaking KASLR

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

Breaking KASLR

- 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

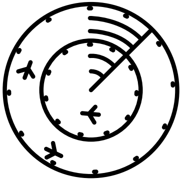
Breaking KASLR

- 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 ⇒ only 64 possible randomization offsets

Breaking KASLR

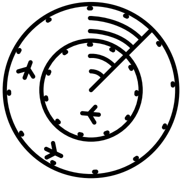
- De-randomize KASLR to access internal kernel structures
- Locate a known value inside the kernel, e.g., Linux banner
  - Start at the default address according to the symbol table of the running kernel
  - Linux KASLR has an entropy of 6 bits ⇒ only 64 possible randomization offsets
- Difference between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

Locating the victim process



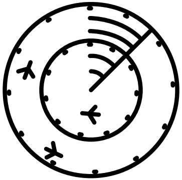
- Linux manages all processes in a linked list

Locating the victim process



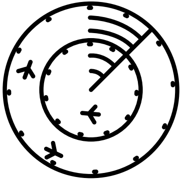
- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure

Locating the victim process



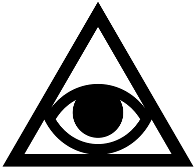
- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure
  - At a fixed offset that varies only among kernel builds

Locating the victim process



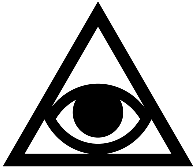
- Linux manages all processes in a linked list
- Head of the list is stored at `init_task` structure
    - At a fixed offset that varies only among kernel builds
- Each task list structure contains a pointer to the next task and
    - PID of the task
    - name of the task
    - Root of the multi-level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

Dumping memory content



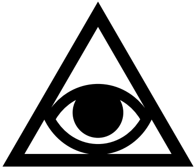
- Resolve physical address using paging structures

Dumping memory content



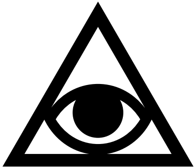
- Resolve physical address using paging structures
- Read the content using the direct-physical map

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

Final steps

- `aeskeyfind` to extract AES keys from the memory dump

Final steps

- `aeskeyfind` to extract AES keys from the memory dump
- `pytruecrypt` to decrypt disk image using the extracted key

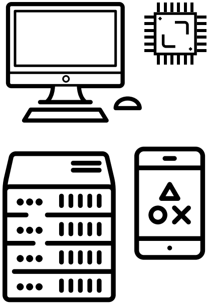
Final steps

- `aeskeyfind` to extract AES keys from the memory dump
- `pytruecrypt` to decrypt disk image using the extracted key

- Affects every application that stores its secret in DRAM

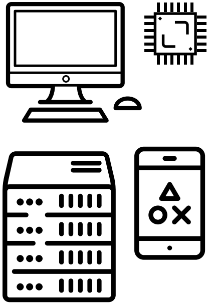
Who is affected?

Affected by Meltdown



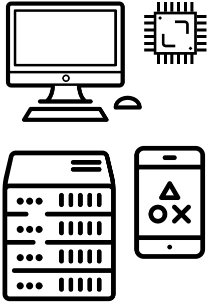
- **Intel**: Almost every CPU

Affected by Meltdown



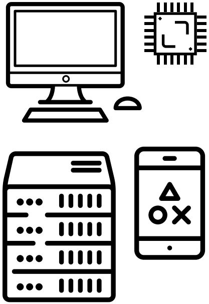
- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected

Affected by Meltdown



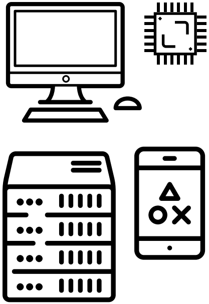
- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75

Affected by Meltdown



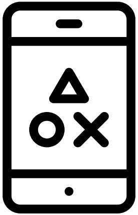
- **Intel**: Almost every CPU
- **AMD**: Seems not to be affected
- **ARM**: Only the Cortex-A75
- **IBM**: System Z, Power Architecture, POWER8 and POWER9

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

Affected by Meltdown



- But there are other CPU manufacturers as well ...

Samsung Galaxy S7

Samsung Galaxy S7

- Exynos Mongoose M1 CPU Architecture

Samsung Galaxy S7

SAMSUNG GALAXY S7

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

Samsung Galaxy S7

Samsung Galaxy S7

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

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

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

Demo

Samsung Galaxy S7

SAMSUNG GALAXY S7

- Luckily they already fixed it

Samsung Galaxy S7

Samsung Galaxy S7

- Luckily they already fixed it
- With their latest update

Samsung Galaxy S7

Samsung Galaxy S7

- Luckily they already fixed it
- With their latest update on July 10, 2018

Affected by Meltdown

- But there are other CPU manufacturers as well ...
- ...which are affected
- Need to evaluate the attack on other CPUs as well
- Notify the users ...
- ...and custom ROM developers, e.g., LineageOS

But wait, what about privileged registers?

Variant 3a

- ARM found a closely related Meltdown variant

Variant 3a

- ARM found a closely related Meltdown variant
- Read of system registers that are not accessible from current exception level

Variant 3a

- ARM found a closely related Meltdown variant
- Read of system registers that are not accessible from current exception level
- ARM Cortex-A15, Cortex-A57 and Cortex-A72 are vulnerable

Variant 3a

- ARM found a closely related Meltdown variant
- Read of system registers that are not accessible from current exception level
- ARM Cortex-A15, Cortex-A57 and Cortex-A72 are vulnerable
- Impact: breaking KASLR and pointer authentication

Demo

Variant 3a

- Intel is affected too (May 21, 2018)

Variant 3a

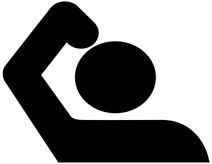
- Intel is affected too (May 21, 2018)
  - Almost every CPU (Core i3/i5/i7, 2nd-8th Intel Core, Xeon, Atom, Pentium, ...)

Variant 3a

- Intel is affected too (May 21, 2018)
  - Almost every CPU (Core i3/i5/i7, 2nd-8th Intel Core, Xeon, Atom, Pentium, ...)
- Rogue System Register Read (RSRE) (CVE-2018-3640)

Is Meltdown (or Spectre) a side-channel attack?

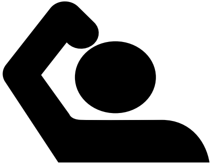
Is Meltdown (or Spectre) a side-channel attack?



Is Meltdown (or Spectre) a side-channel attack?

No.

Is Meltdown (or Spectre) a side-channel attack?



No.

- We read the data directly

Is Meltdown (or Spectre) a side-channel attack?

No.

- We read the data directly
- We use a side channel internally for transmission

Is Meltdown (or Spectre) a side-channel attack?

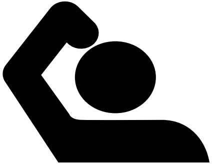
No.

- We read the data directly
- We use a side channel internally for transmission

→ does not make the entire thing a side-channel attack

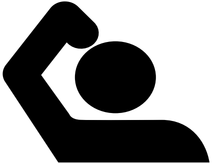
Is Meltdown a variant of Spectre? Is it speculative execution?

Is Meltdown a variant of Spectre? Is it speculative execution?

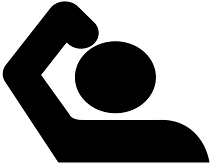


Is Meltdown a variant of Spectre? Is it speculative execution?

No.



Is Meltdown a variant of Spectre? Is it speculative execution?



No.

- Often heard: “Meltdown is speculating beyond faulting instructions”

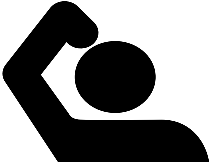
Is Meltdown a variant of Spectre? Is it speculative execution?

No.

- Often heard: “Meltdown is speculating beyond faulting instructions”

→ That’s not speculative execution

Is Meltdown a variant of Spectre? Is it speculative execution?



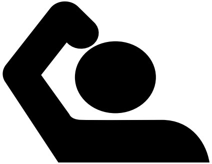
No.

- Often heard: “Meltdown is speculating beyond faulting instructions”

→ That’s not speculative execution

- “Speculating beyond faulting instructions” - not even the actual problem

Is Meltdown a variant of Spectre? Is it speculative execution?



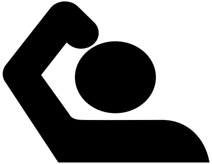
No.

- Often heard: "Meltdown is speculating beyond faulting instructions"

→ That's not speculative execution

- "Speculating beyond faulting instructions" - not even the actual problem
- AMD does that - but is not affected!

Is Meltdown a variant of Spectre? Is it speculative execution?



No.

- Often heard: “Meltdown is speculating beyond faulting instructions”

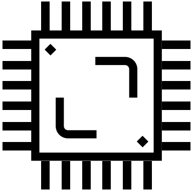
→ That’s not speculative execution

- “Speculating beyond faulting instructions” - not even the actual problem
- AMD does that - but is not affected!

→ Actual problem: fetching & using real values for instructions after faulting ones

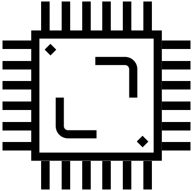
How can this all be fixed?

Meltdown Mitigation



- Problem is rooted in hardware

Meltdown Mitigation

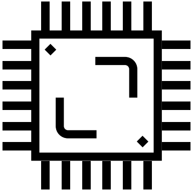


- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them

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

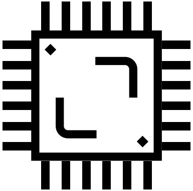
Meltdown Mitigation



- 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

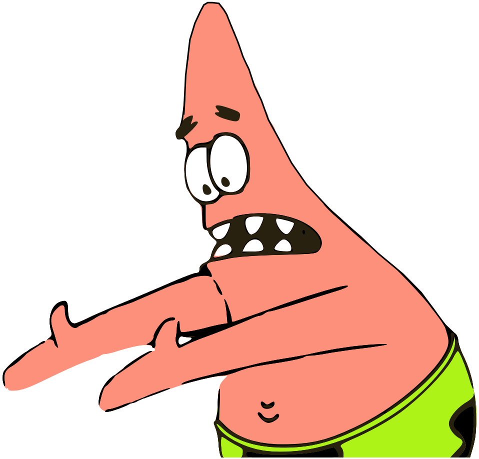


- Problem is rooted in hardware
- Race condition between the memory fetch and corresponding permission check
  - Serialize both of them
- Hard split of user space and kernel space
  - New bit in control register

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

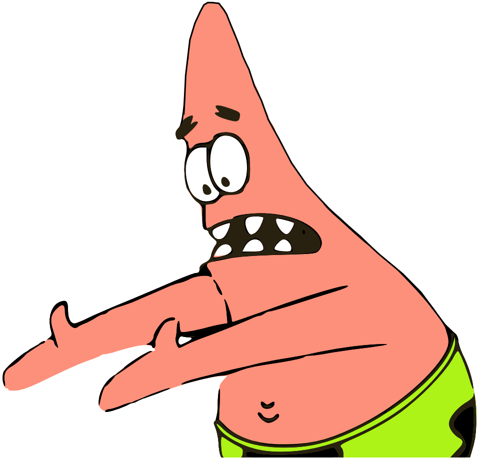
Meltdown Mitigation

- Kernel addresses in user space are a problem

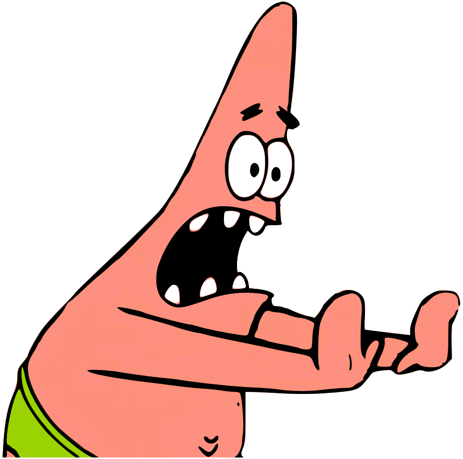


Meltdown Mitigation

- Kernel addresses in user space are a problem
- Why don't we take the kernel addresses...

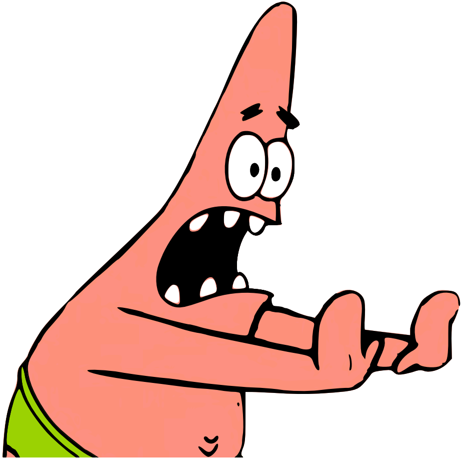


Meltdown Mitigation



- …and remove them if not needed?

Meltdown Mitigation



- ...and remove them if not needed?
- User accessible check in hardware is not reliable

Meltdown Mitigation

- Unmap the kernel in user space

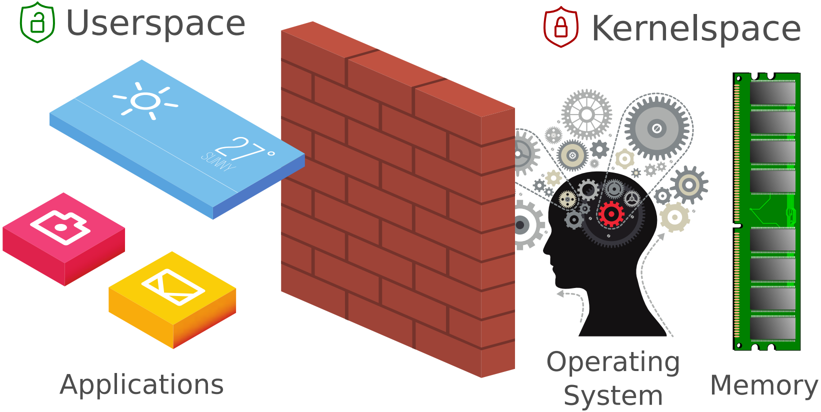
Meltdown Mitigation

- Unmap the kernel in user space
- Kernel addresses are then no longer present

Meltdown Mitigation

- Unmap the kernel in user space
- Kernel addresses are then no longer present
- Memory which is not mapped cannot be accessed at all

KAISER



Userspace

Kernelspace

Applications

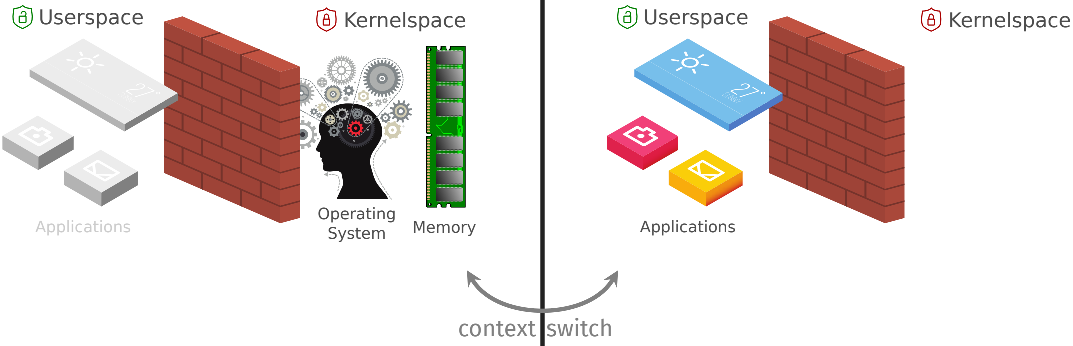
Operating System

Memory

KAISER

Kernel View

User View



Userspace

Kernelspace

Applications

Operating System

Memory

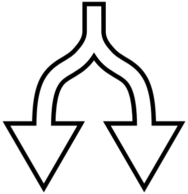
Userspace

Kernelspace

Applications

context switch

KAISER

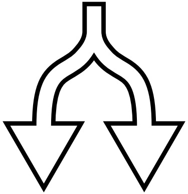


- We published KAISER in May 2017 ...

KAISER

- We published KAISER in May 2017 ...
- ...as a countermeasure against other side-channel attacks

KAISER



- We published KAISER in May 2017 ...
- ...as a countermeasure against other side-channel attacks
- Inadvertently defeats Meltdown as well

KAISER

- We published KAISER in May 2017 ...
- ...as a countermeasure against other side-channel attacks
- Inadvertently defeats Meltdown as well
- PoC implementation for the Linux kernel

KAISER

- Hardware interrupt while running in user mode
  - Kernel needs to deal with interrupt but does not exist anymore in address space
  - Traps, NMI, system calls, ...
- Must map some kernel code in user space

KAISER

- Need to update CR3 in order to switch to other address space

KAISER

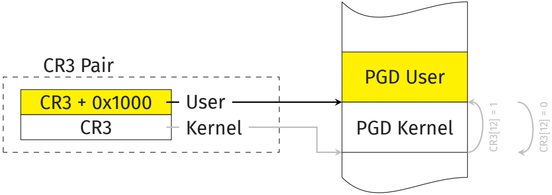
- Need to update CR3 in order to switch to other address space
- How can we do this efficiently?

KAISER

- Need to update CR3 in order to switch to other address space
- How can we do this efficiently?
- Instead of one PGD, two PGDs are allocated
  - 8k in size and 8k aligned

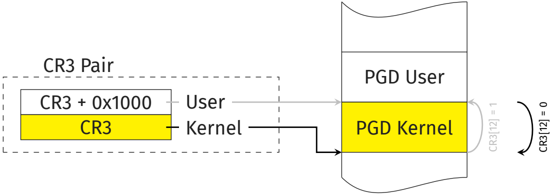
KAISER

- Need to update CR3 in order to switch to other address space
- How can we do this efficiently?
- Instead of one PGD, two PGDs are allocated
    - 8k in size and 8k aligned
- **Trick:** Just flip bit 12 in the pointer to swap between both halves



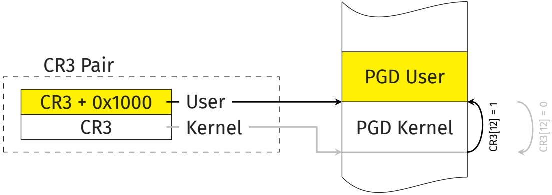
KAISER

- Need to update CR3 in order to switch to other address space
- How can we do this efficiently?
- Instead of one PGD, two PGDs are allocated
  - 8k in size and 8k aligned
- **Trick**: Just flip bit 12 in the pointer to swap between both halves

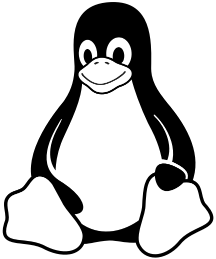


KAISER

- Need to update CR3 in order to switch to other address space
- How can we do this efficiently?
- Instead of one PGD, two PGDs are allocated
  - 8k in size and 8k aligned
- **Trick:** Just flip bit 12 in the pointer to swap between both halves

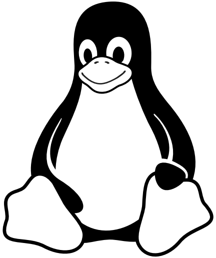


Kernel Page-table Isolation



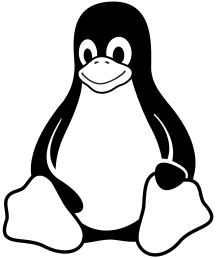
- Intel and others improved KAISER

Kernel Page-table Isolation



- Intel and others improved KAISER
- Merged it into upstream as KPTI (Kernel Page-table Isolation)

Kernel Page-table Isolation



- Intel and others improved KAISER
- Merged it into upstream as KPTI (Kernel Page-table Isolation)
- Kernel patches are available for arm64 as well

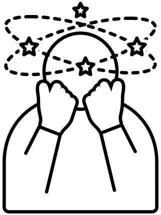
Apple

- Apple released updates in iOS 11.2, macOS 10.13.2 and tvOS 11.2 to mitigate Meltdown
- Boot option: `-no-shared-cr3`
  - Unmaps the user space while running in kernel mode
  - But not vice versa

KVA Shadow

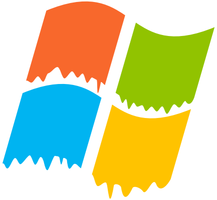
- Kernel Virtual Address (KVA) Shadow
- Meltdown Mitigation for Microsoft Windows

Implementing



- Introducing such a fundamental change to the operating system is extremely challenging
- Our PoC implementation contained many bugs as well

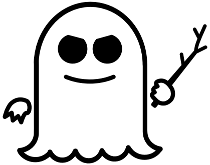
Total Meltdown



- Discovered by Ulf Frisk (@ulffrisk) in the 2018-02 security update
  - CVE-2018-1038
- Modified the PML4 entry of `0x1ed` to allow to access page from user-mode
- On Windows 7 and Server 2018 R2: Self-Referencing Entry
- Allows to read and modify entire physical memory

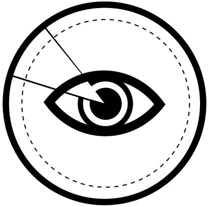
What now?

Future



- More attacks exploiting performance optimizations in hardware
  - New variants are disclosed frequently

A unique chance



A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)

Proof-of-Concept

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

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

Conclusion

- Underestimated microarchitectural attacks for a long time
- Meltdown exploit performance optimizations
  - Allow to leak arbitrary memory
- Countermeasures come with a performance impact
- Find trade-offs between security and performance

Moritz Lipp (@mlqxyz)

Michael Schwarz (@misc0110)

Daniel Gruss (@lavados)

Meltdown

Basics, Details, Consequences

Black Hat USA 2018
9 August, 2018 - Las Vegas, NV, USA

References

R. Grisenthwaite. Cache Speculation Side-channels. 2018.

D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard. KASLR is Dead: Long Live KASLR. In: ESSoS. 2017.

Intel. Intel Analysis of Speculative Execution Side Channels. Jan. 2018. URL: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

K. Johnson. KVA Shadow: Mitigating Meltdown on Windows. Mar. 2018. URL: https://blogs.technet.microsoft.com/srd/2018/03/23/kva-shadow-mitigating-meltdown-on-windows/.

M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In: USENIX Security Symposium. 2018.

LWN. The current state of kernel page-table isolation. 2017. URL: https://lwn.net/SubscriberLink/741878/eb6c9d3913d7cb2b/.

Y. Yarom and K. Falkner. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In: USENIX Security Symposium. 2014.