Side-Channel Lab I

Michael Schwarz

Security Week Graz 2019

Scope

Scope

Scope

Scope

- everyday hardware: servers, workstations, laptops, smartphones. . .

Scope

- everyday hardware: servers, workstations, laptops, smartphones. . .
- **remote side-channel attacks**

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information **leaks** because of the **hardware** it runs on

Side channels

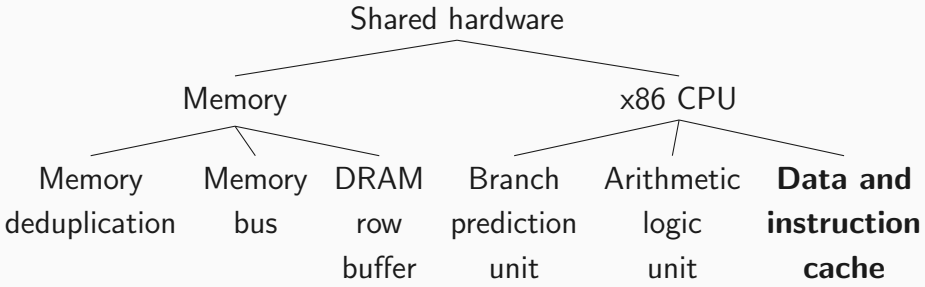
- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information **leaks** because of the **hardware** it runs on
- no "bug" in the sense of a mistake → lots of performance optimizations

Side channels

- **safe software** infrastructure → no bugs, e.g., Heartbleed
- does not mean safe execution
- information **leaks** because of the **hardware** it runs on
- no "bug" in the sense of a mistake → lots of performance optimizations

→ crypto and sensitive info., e.g., keystrokes and mouse movements

Shared hardware



Why targeting the cache?

- shared across cores

Why targeting the cache?

- shared across cores
- fast

Why targeting the cache?

- shared across cores
- fast

→ fast cross-core attacks!

Timing differences

- caches improve performance

Timing differences

- caches improve performance
- SRAM is expensive → small caches

Timing differences

- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses

Timing differences

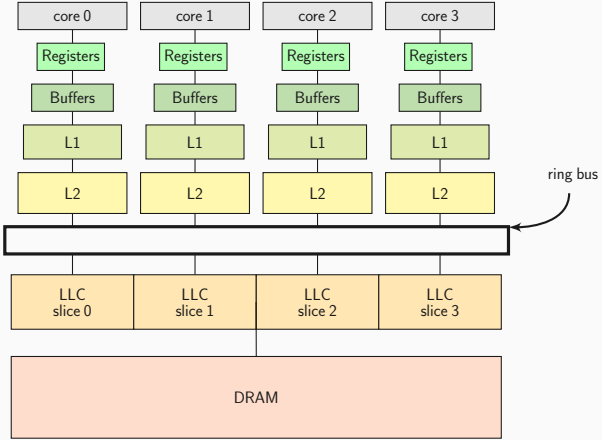
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
    - data is **cached** → cache hit → **fast**

Timing differences

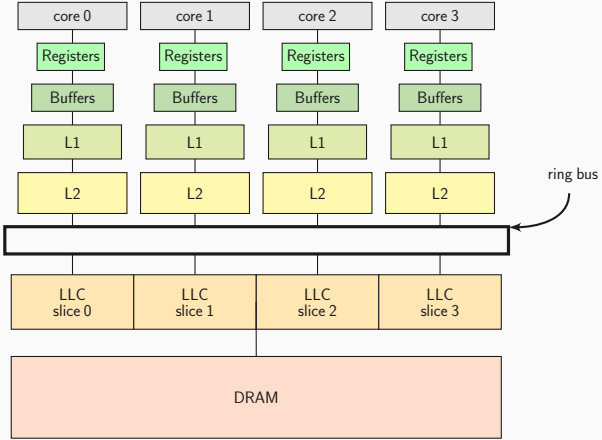
- caches improve performance
- SRAM is expensive → small caches
- different timings for memory accesses
  - data is **cached** → cache hit → **fast**
  - data is **not cached** → cache miss → **slow**

Memory Hierarchy

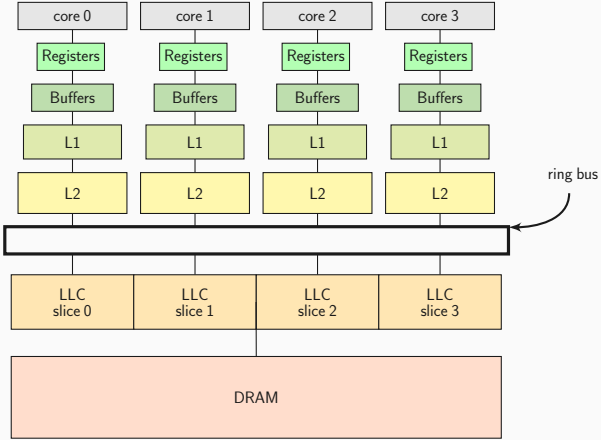
Memory Hierarchy



Memory Hierarchy

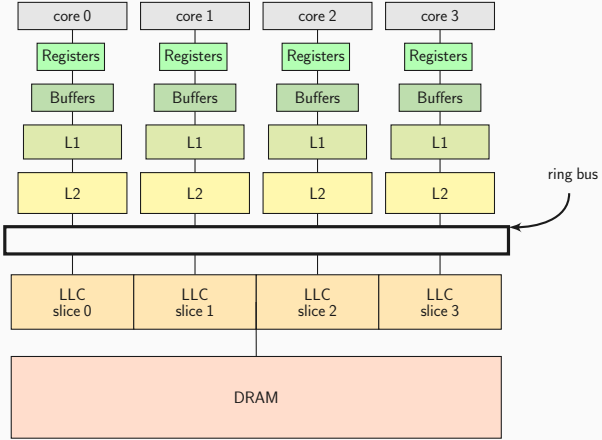


Memory Hierarchy



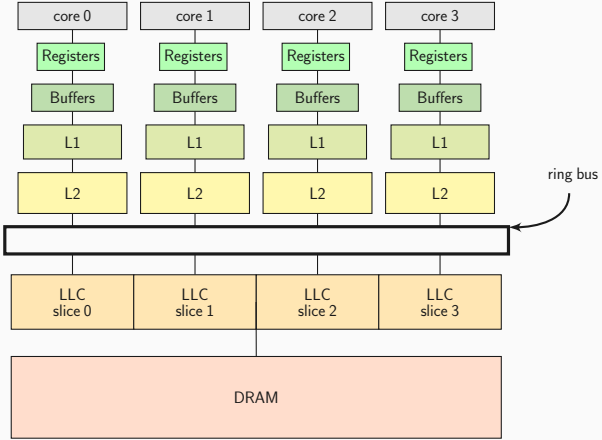
- L1 and L2 are private

Memory Hierarchy



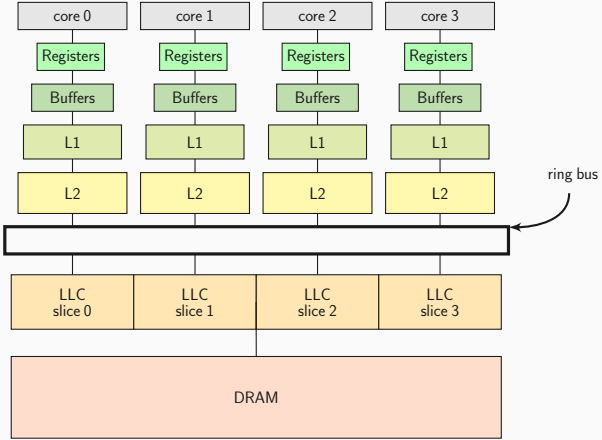
- L1 and L2 are private
- last-level cache:

Memory Hierarchy



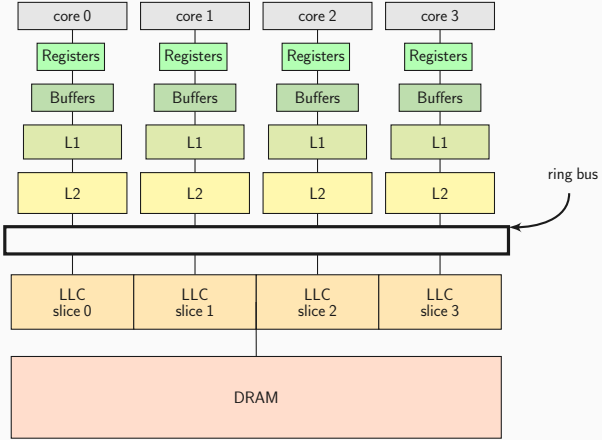
- L1 and L2 are private
- last-level cache:
  - divided in **slices**

Memory Hierarchy



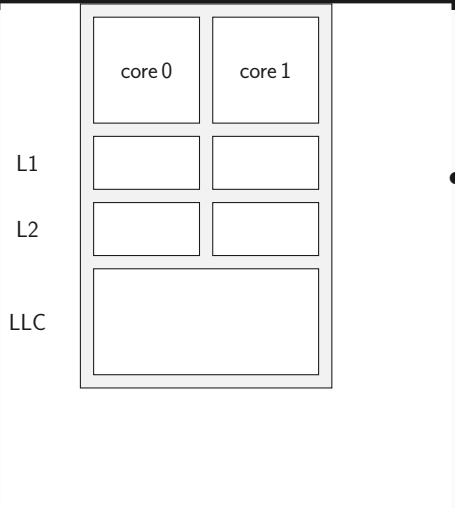
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores

Memory Hierarchy



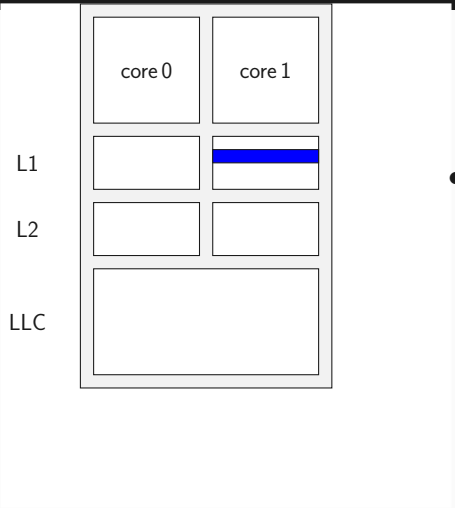
- L1 and L2 are private
- last-level cache:
  - divided in **slices**
  - **shared** across cores
  - **inclusive**

Inclusive property



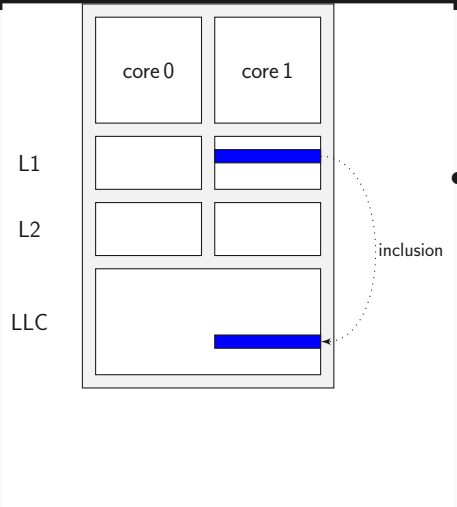
- **inclusive** LLC: superset of L1 and L2

Inclusive property



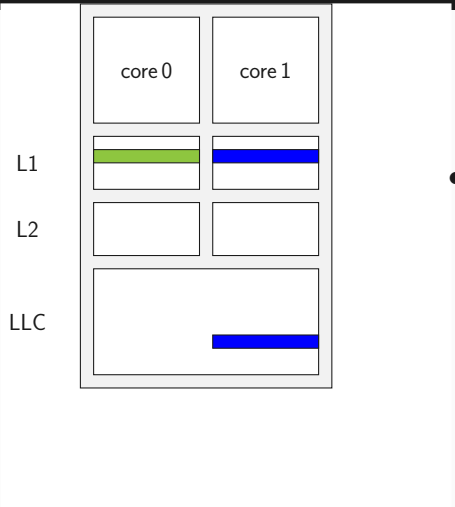
- **inclusive** LLC: superset of L1 and L2

Inclusive property



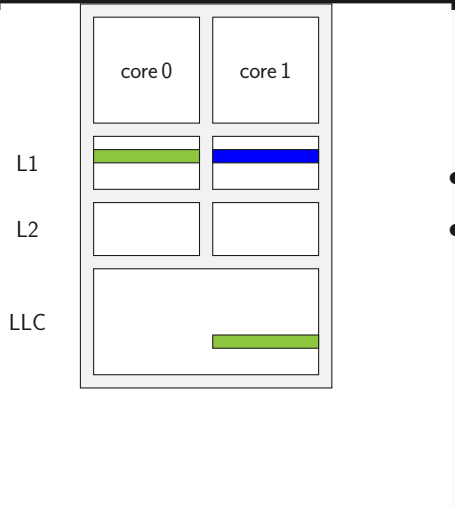
- **inclusive** LLC: superset of L1 and L2

Inclusive property



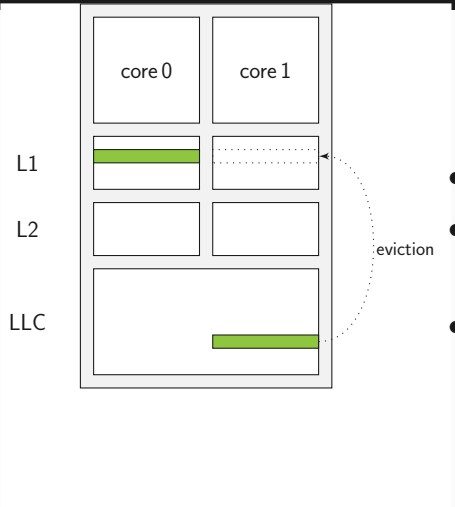
- **inclusive** LLC: superset of L1 and L2

Inclusive property



- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2

Inclusive property



- **inclusive** LLC: superset of L1 and L2
- data evicted from the LLC is also evicted from L1 and L2
- a core can **evict lines** in the private L1 **of another core**

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles

Latency comparison

On current Intel CPUs:

- Registers: 0-1 cycle
- L1 cache: 4 cycles
- L2 cache: 12 cycles
- L3 cache: 26-31 cycles
- DRAM memory: >120 cycles

Measuring timing leakage

How every timing attack works:

- learn timing of different corner cases

Measuring timing leakage

How every timing attack works:

- learn timing of different corner cases
- later, we recognize these corner cases by timing only

Steps

1. build two cases: cache hits and cache misses

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a **histogram**!

Steps

1. build two cases: cache hits and cache misses
2. time each case many times (get rid of noise)
3. we have a **histogram**!
4. find a **threshold** to distinguish the two cases

Cache hits

Loop:

1. measure time

Cache hits

Loop:

1. measure time
2. access variable (always cache **hit**)

Cache hits

Loop:

1. measure time
2. access variable (always cache **hit**)
3. measure time

Cache hits

Loop:

1. measure time
2. access variable (always cache **hit**)
3. measure time
4. update histogram with delta

Cache misses

Loop:

1. measure time

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)
3. measure time

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)
3. measure time
4. update histogram with delta

Cache misses

Loop:

1. measure time
2. access variable (always cache **miss**)
3. measure time
4. update histogram with delta
5. **flush** variable (`clflush` instruction)

Time to code

Accurate timings

- very short timings
- `rdtsc` instruction: cycle-accurate timestamps

Accurate timings

- very short timings
- `rdtsc` instruction: cycle-accurate timestamps

```
[...]
rdtsc
function()
rdtsc
[...]
```

Accurate timings

- do you measure what you *think* you measure?
- **out-of-order** execution → what is really executed

Accurate timings

- do you measure what you *think* you measure?
- **out-of-order** execution → what is really executed

```
rdtsc
function()
[...]
rdtsc
```

```
rdtsc
[...]
rdtsc
function()
```

```
rdtsc
rdtsc
function()
[...]
```

Accurate timings

- use pseudo-serializing instruction `rdtscp` (recent CPUs)

Accurate timings

- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`

Accurate timings

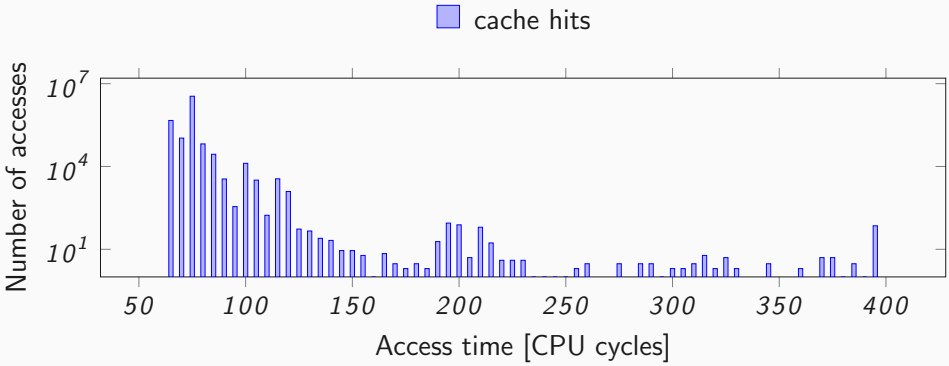
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Accurate timings

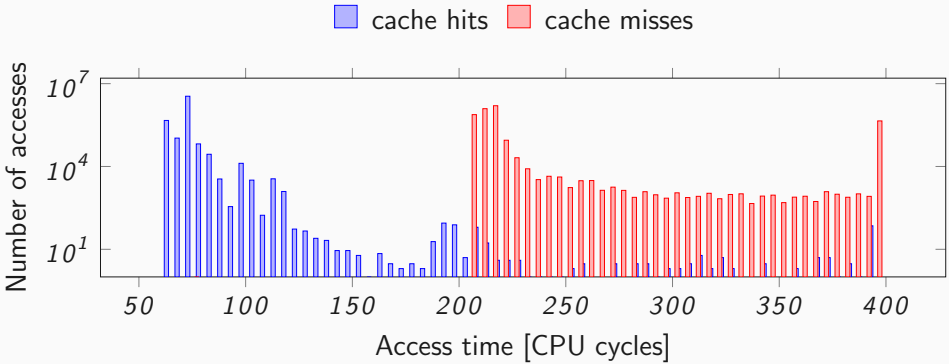
- use pseudo-serializing instruction `rdtscp` (recent CPUs)
- and/or use serializing instructions like `cpuid`
- and/or use fences like `mfence`

Intel, *How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures White Paper*, December 2010.

Timing differences



Timing differences



Find threshold

- as high as possible

Find threshold

- as high as possible
- most cache hits are below

Find threshold

- as high as possible
- most cache hits are below
- no cache miss below

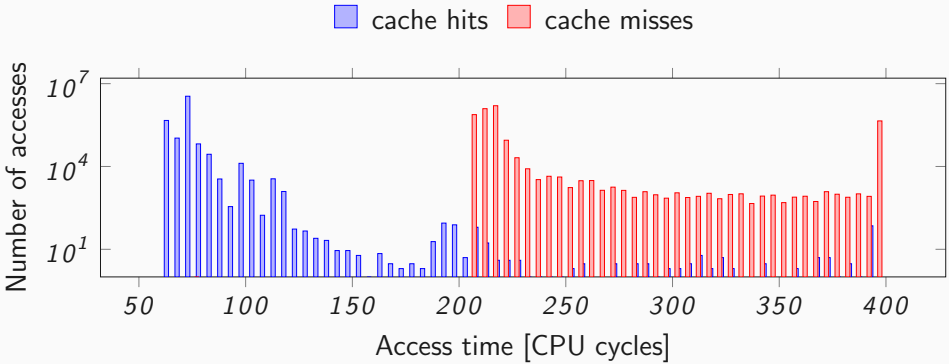
Hit vs. Miss?

- Hit → Data is fetched from buffers, L1, L2, or L3

Hit vs. Miss?

- Hit → Data is fetched from buffers, L1, L2, or L3
- Miss → Data is fetched from DRAM

From Histogram to Attack



Type of attacks

- cache attacks → exploit timing differences of memory accesses

Type of attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content

Type of attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other

Type of attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs

Type of attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- side-channel attack: one malicious process spies on benign processes

Type of attacks

- cache attacks → exploit timing differences of memory accesses
- attacker monitors which lines are accessed, not the content
- covert channel: two processes communicating with each other
  - not allowed to do so, e.g., across VMs
- side-channel attack: one malicious process spies on benign processes
  - e.g., steals crypto keys, spies on keystrokes

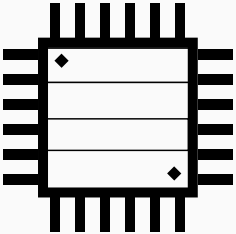
Flush+Reload

Shared Memory

ATTACKER

flush

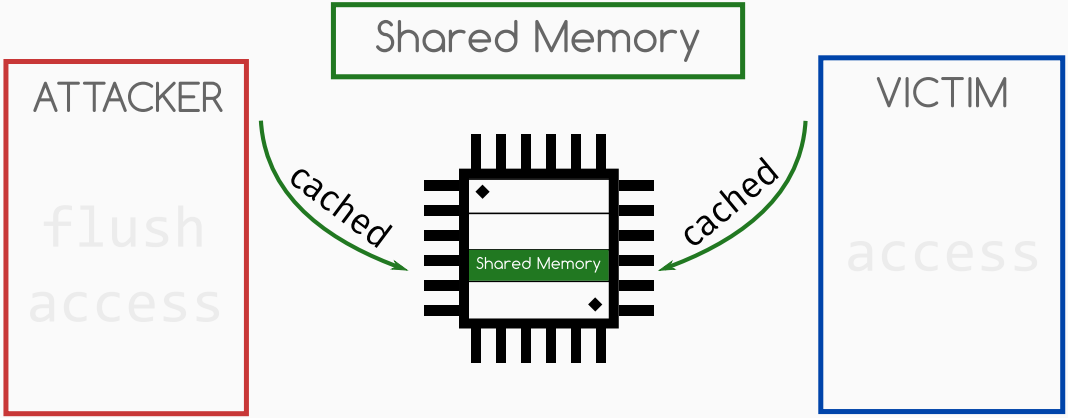
access



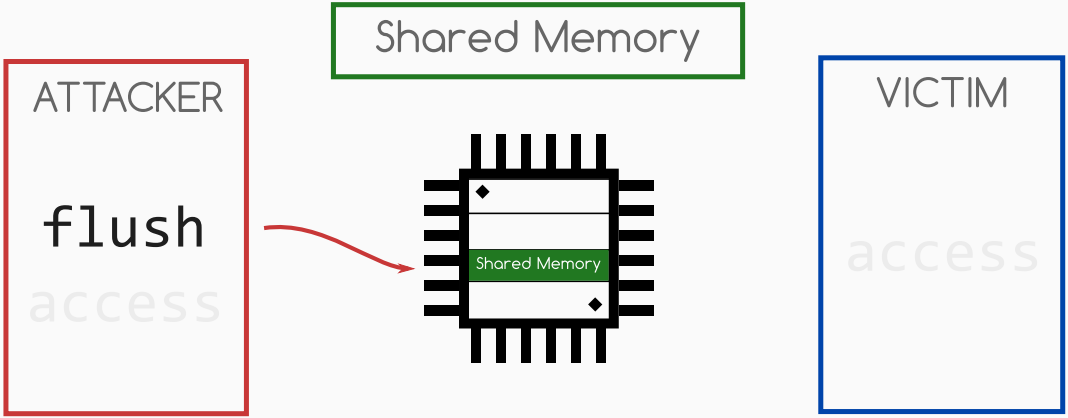
VICTIM

access

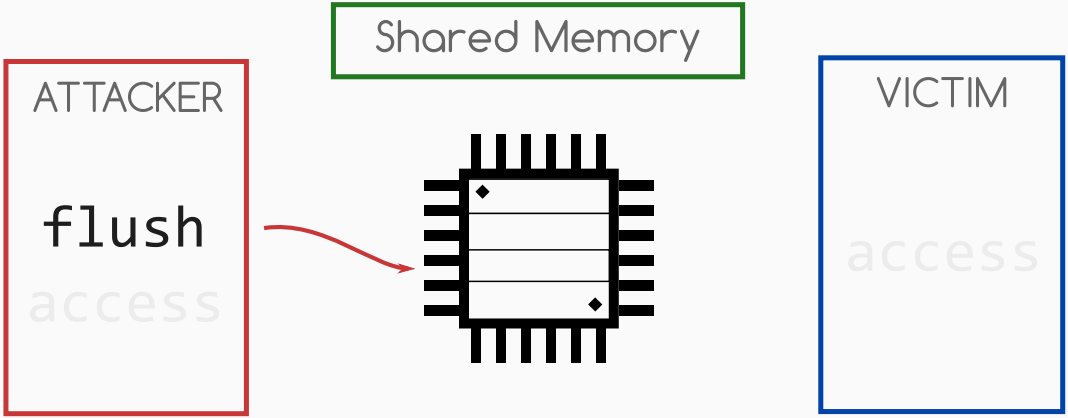
Flush+Reload



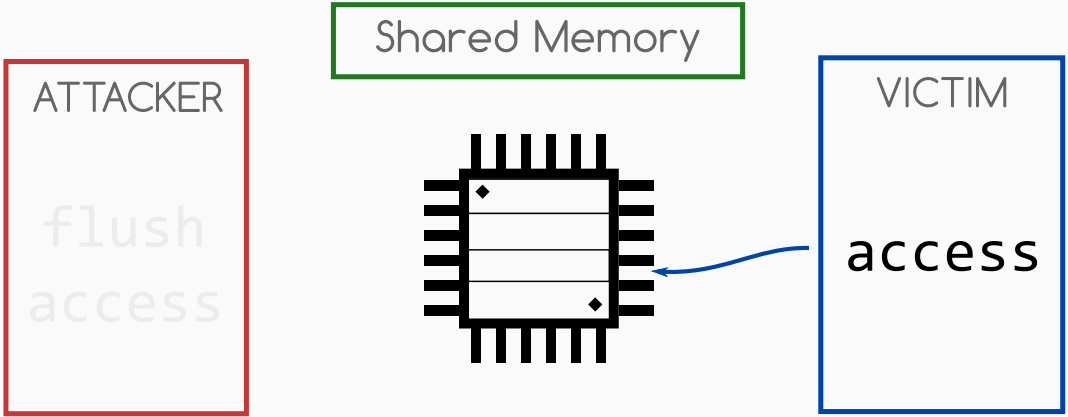
Flush+Reload



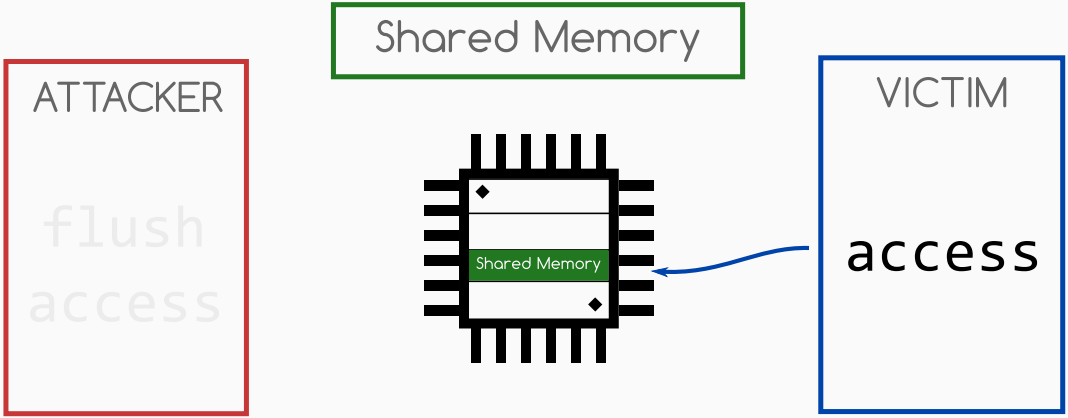
Flush+Reload



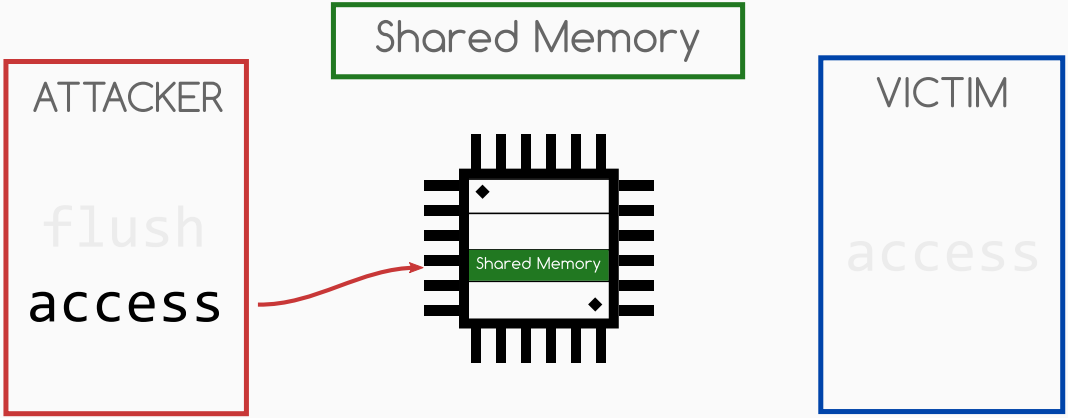
Flush+Reload



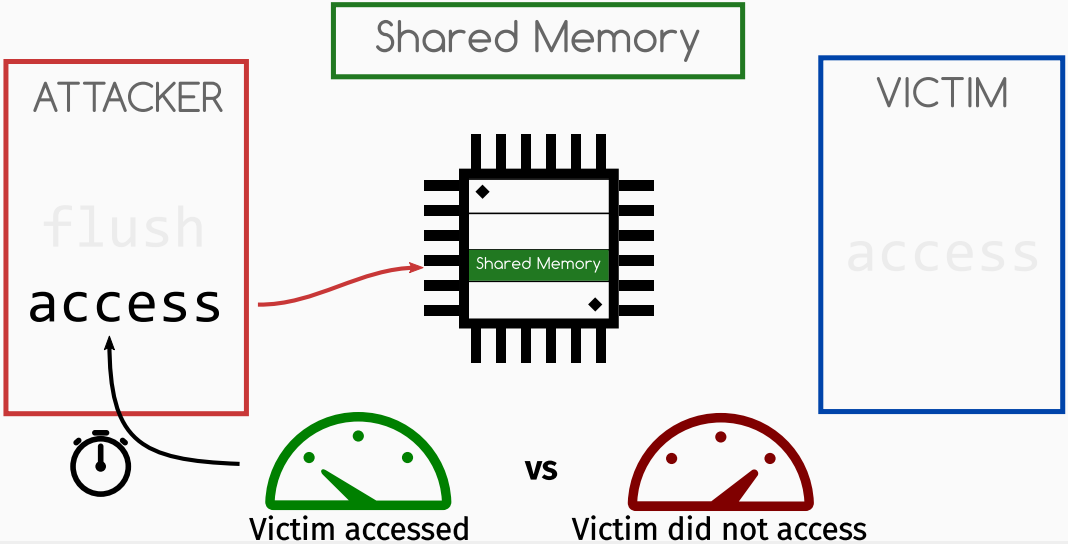
Flush+Reload



Flush+Reload



Flush+Reload



Signatures (RSA)

$$M = C^d \bmod n$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

Result = C

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}} \times \underbrace{\boxed{C}}_{\text{multiply}}$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}}$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}}$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}} \times \underbrace{\boxed{C}}_{\text{multiply}}$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}} \times \underbrace{\boxed{C}}_{\text{multiply}}$$

Signatures (RSA)

$$M = C^{\boxed{d}} \bmod n$$

1	1	0	0	1	1	0	···

$$\boxed{\text{Result}} = \underbrace{\boxed{\text{Result}} \times \boxed{\text{Result}}}_{\text{square}}$$

Time to code

Side-channel attack on user input

- locate **key-dependent** memory accesses

Side-channel attack on user input

- locate **key-dependent** memory accesses
- How to locate key-dependent memory accesses?

Challenges

- It's complicated:

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries

Challenges

- It's complicated:
  - Large binaries and libraries (third-party code)
  - Many libraries (gedit: 60MB)
  - Closed-source / unknown binaries
  - Self-compiled binaries
- Difficult to find **all** exploitable addresses

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically

Exploitation Phase

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. “events” (keystrokes, encryptions, ...)

Exploitation Phase

Cache Template Attacks

Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. "events" (keystrokes, encryptions, ...)
  - called "Cache Template"

Exploitation Phase

Cache Template Attacks

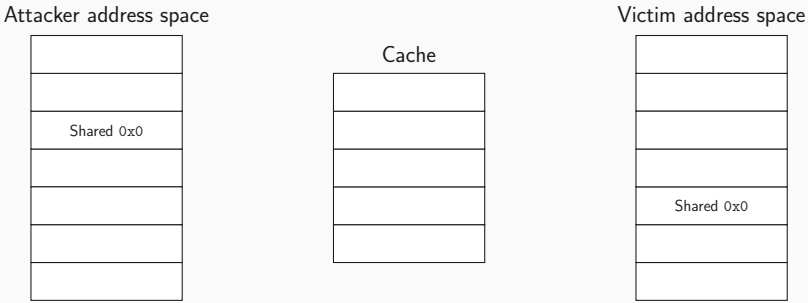
Profiling Phase

- Preprocessing step to find exploitable addresses automatically
  - w.r.t. "events" (keystrokes, encryptions, ...)
  - called "Cache Template"

Exploitation Phase

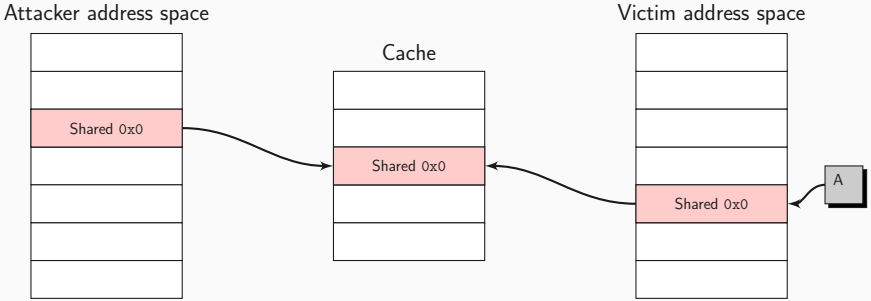
- Monitor exploitable addresses

Profiling Phase



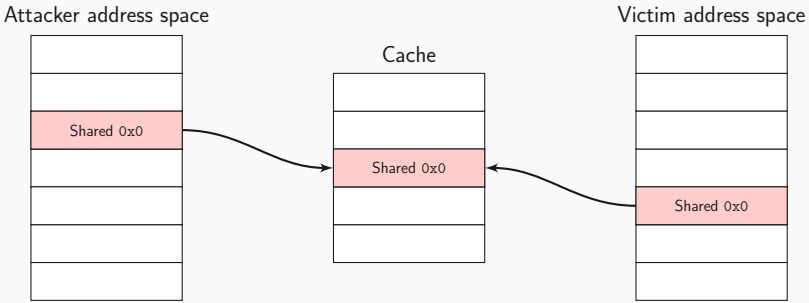
Cache is empty

Profiling Phase



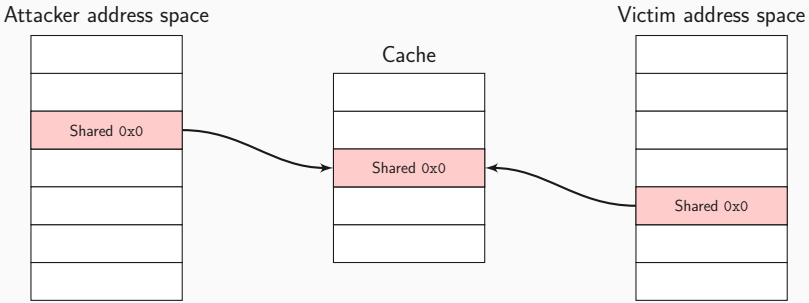
Attacker triggers an event

Profiling Phase



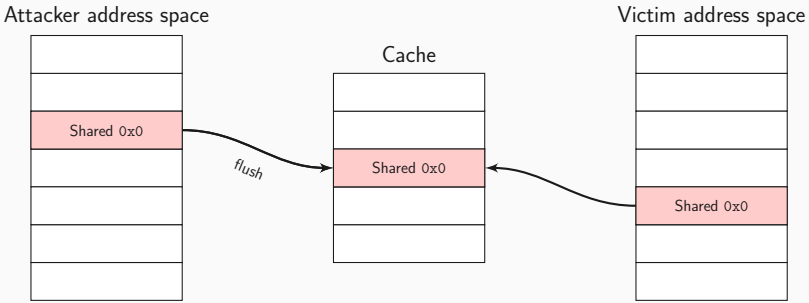
Attacker checks one address for cache hits ("Reload")

Profiling Phase



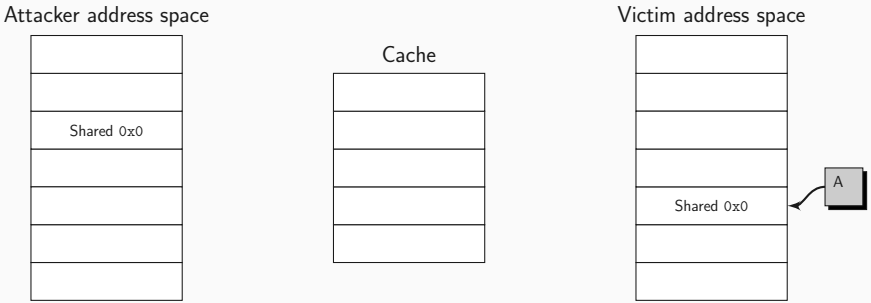
Update number of cache hits per event

Profiling Phase



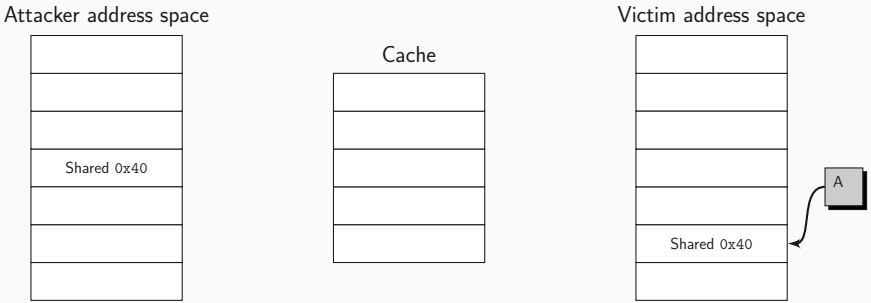
Attacker flushes shared memory

Profiling Phase



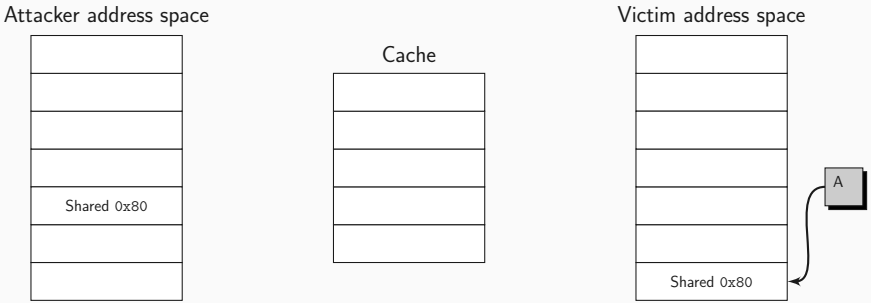
Repeat for higher accuracy

Profiling Phase



Continue with next address

Profiling Phase



Continue with next address

What to profile?


```
$> ps -A | grep gedit
$> cat /proc/<pid>/maps
00400000-00489000 r-xp 00000000 fd:01 396356
/usr/bin/gedit
7f5a96991000-7f5a96a51000 r-xp 00000000 fd:01 399365
/usr/lib/x86_64-linux-gnu/libgdk-3.so.0.2200.30
...
```

memory range, access rights, offset, –, –, file name

Profiling a single event


```
$> cd practicals/02_cache_template_attacks/
$> make
$> # start the targeted program (e.g., gedit)
$> sleep 2; ./profiling /usr/lib/x86_64-linux-gnu/
   libgdk-3.so.0.2200.30
```

... and hold down a key in the target program

Profiling a single event


```
$> cd practicals/02_cache_template_attacks/
$> make
$> # start the targeted program (e.g., gedit)
$> sleep 2; ./profiling /usr/lib/x86_64-linux-gnu/
   libgdk-3.so.0.2200.30
```

... and hold down a key in the target program

save addresses with peaks!

Exploitation phase


```
$> # ./spy <file> <offset>
$> ./spy /usr/lib/x86_64-linux-gnu/libgdk-3.so.0.2200.30 336896
Monitoring offset 336896
Hit #0
Hit #1
Hit #2
...
```

Time to code

Terminal

File Edit View Search Terminal Help

```
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 08:02 26
8050                    /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

[prefetch] <DIR> 14.03.2017 21:44:26

Terminal

File Edit View Search Terminal Help

```
shark% ./spy
```

/home/daniel/}:

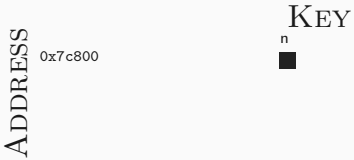
Open Untitled Document 1 Save

1

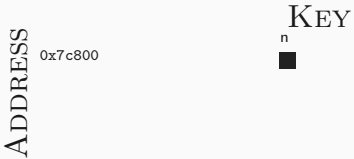
Plain Text Tab Width: 2 Ln 1, Col 1 INS

Cache Template Attack Demo

Profiling Phase: 1 Event, 1 Address



Profiling Phase: 1 Event, 1 Address



Example: Cache Hit Ratio for (`0x7c800`, `n`): 200 / 200

Profiling Phase: All Events, 1 Address

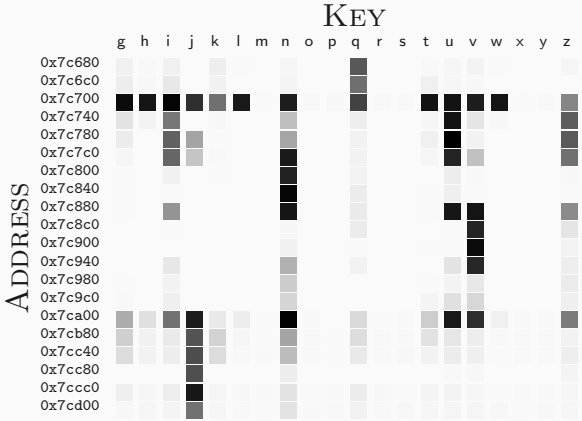
Profiling Phase: All Events, 1 Address

Example: Cache Hit Ratio for $(\texttt{0x7c800}, \texttt{u})$: 13 / 200

Profiling Phase: All Events, 1 Address

Distinguish `n` from other keys by monitoring `0x7c800`

Profiling Phase: All Events, All Addresses



D. Gruss, R. Spreitzer, and S. Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In: USENIX Security Symposium. 2015.

Side-Channel Lab II

Michael Schwarz

Security Week Graz 2019

Covert channel

What is a covert channel?

- Two programs would like to communicate

Covert channel

What is a covert channel?

- Two programs would like to communicate but are not allowed to do so

Covert channel

What is a covert channel?

- Two programs would like to communicate but are not allowed to do so
  - either because there is no communication channel...

Covert channel

What is a covert channel?

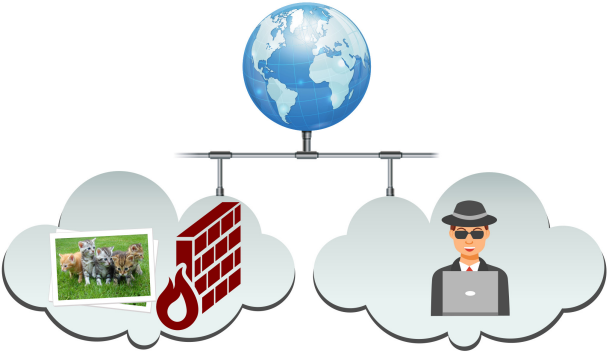
- Two programs would like to communicate but are not allowed to do so
  - either because there is no communication channel...
  - ...or the channels are monitored and programs are stopped on communication attempts

Covert channel

What is a covert channel?

- Two programs would like to communicate but are not allowed to do so
  - either because there is no communication channel...
  - ...or the channels are monitored and programs are stopped on communication attempts
- Use side channels and stay stealthy

Covert channel



Covert channel

State of the art

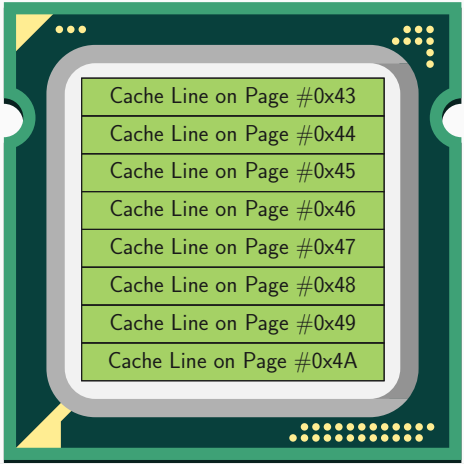
method	raw capacity	err. rate	true capacity	env.
F+F [Gru+16]	3968Kbps	0.840%	3690Kbps	native
F+R [Gru+16]	2384Kbps	0.005%	2382Kbps	native
E+R [Lip+16]	1141Kbps	1.100%	1041Kbps	native
P+P [Mau+17]	601Kbps	0.000%	601Kbps	native
P+P [Liu+15]	600Kbps	1.000%	552Kbps	virt
P+P [Mau+17]	362Kbps	0.000%	362Kbps	native

Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



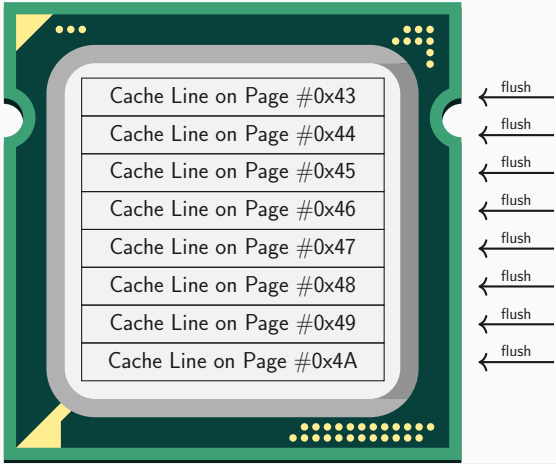
Receiver

Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



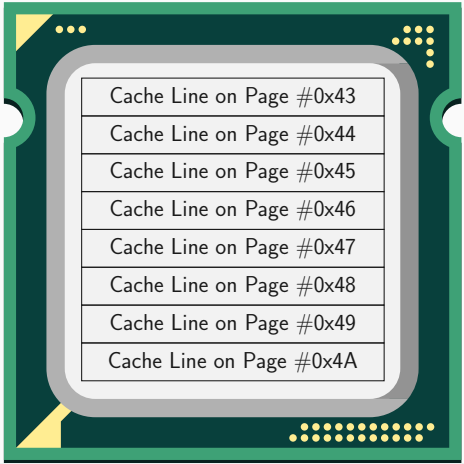
Receiver

Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



Receiver

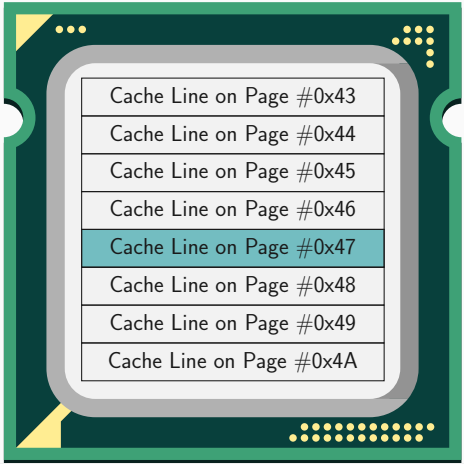
Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

reload

Last-level cache



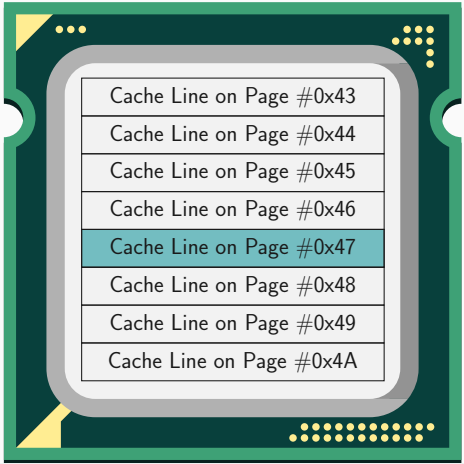
Receiver

Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



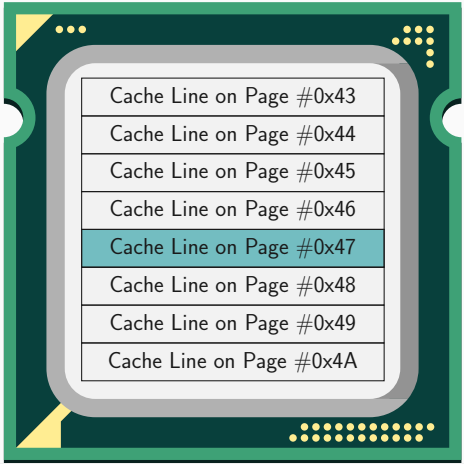
Receiver

Sending Data (easy but inefficient)

Sender

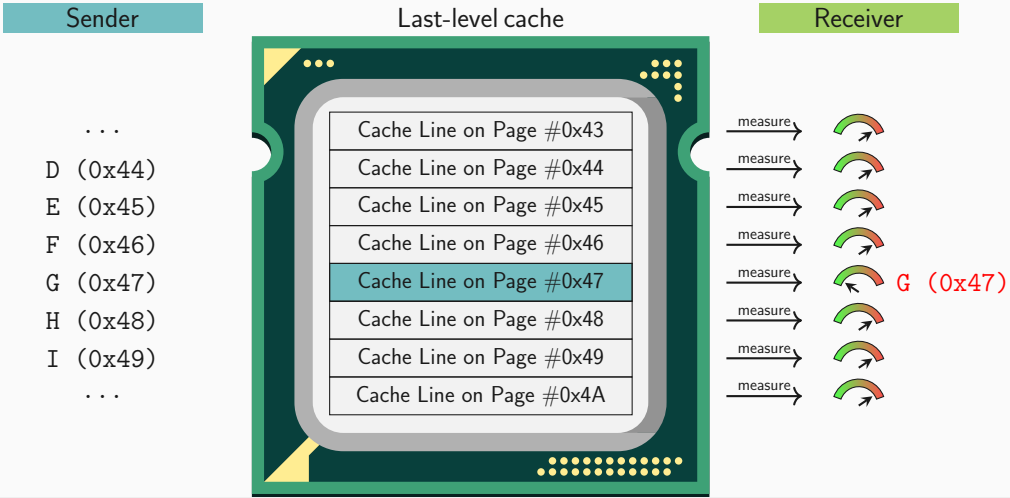
```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



Receiver

Sending Data (easy but inefficient)

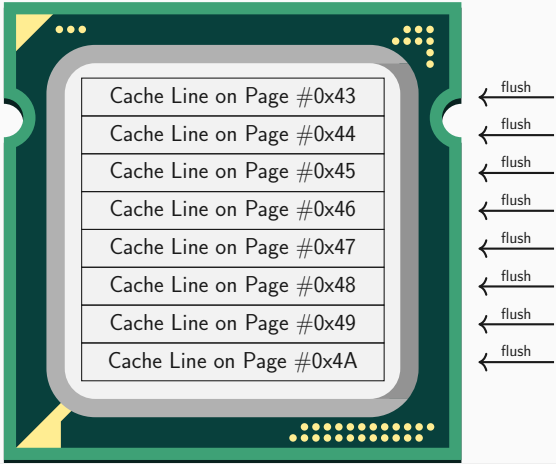


Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



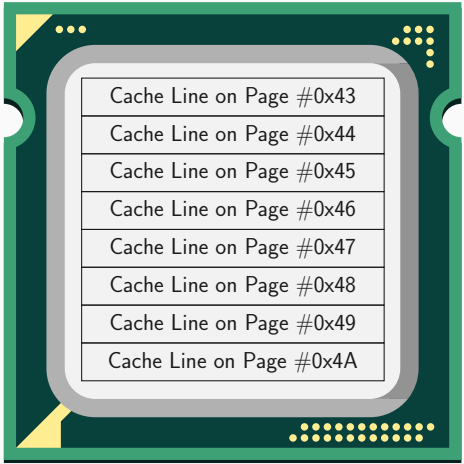
Receiver

Sending Data (easy but inefficient)

Sender

```
...
D (0x44)
E (0x45)
F (0x46)
G (0x47)
H (0x48)
I (0x49)
...
```

Last-level cache



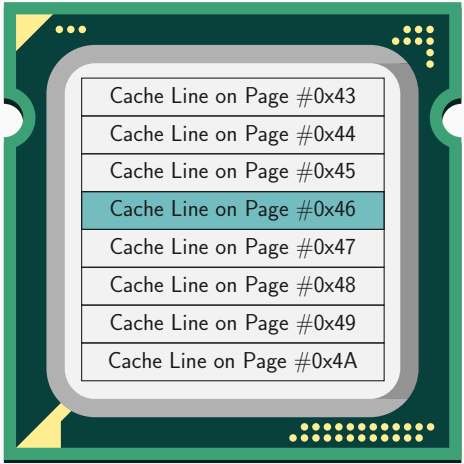
Receiver

Sending Data (easy but inefficient)

Sender

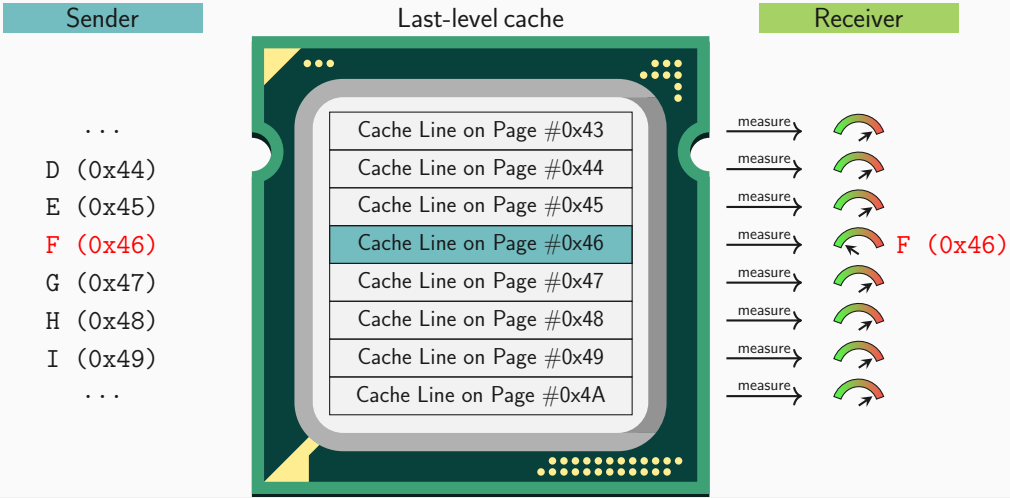
Last-level cache

Receiver




```
...
D (0x44)
E (0x45)
F (0x46) --reload-->
G (0x47)
H (0x48)
I (0x49)
...
```

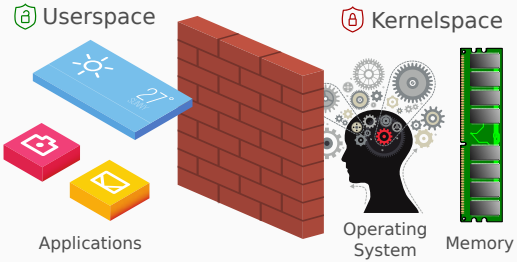
Sending Data (easy but inefficient)



Time to code

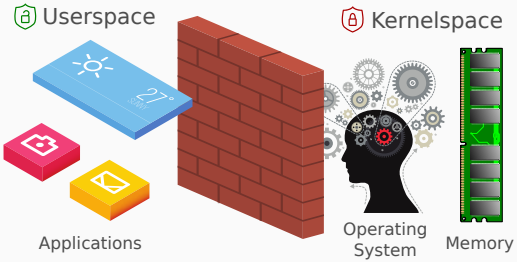
Operating Systems 101

Memory Isolation



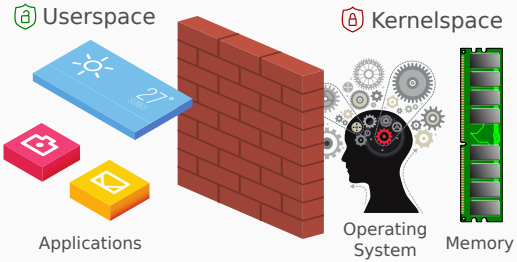
- Kernel is isolated from user space

Memory Isolation



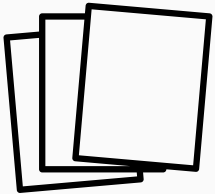
- Kernel is isolated from user space
- This isolation is a combination of hardware and software

Memory Isolation



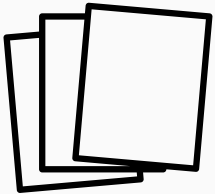
- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel

Paging



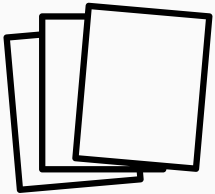
- CPU support virtual address spaces to isolate processes

Paging



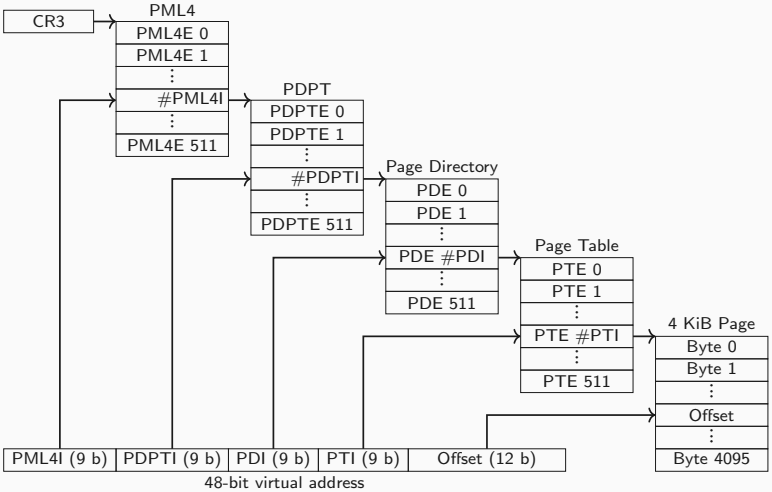
- CPU support virtual address spaces to isolate processes
- Physical memory is organized in page frames

Paging

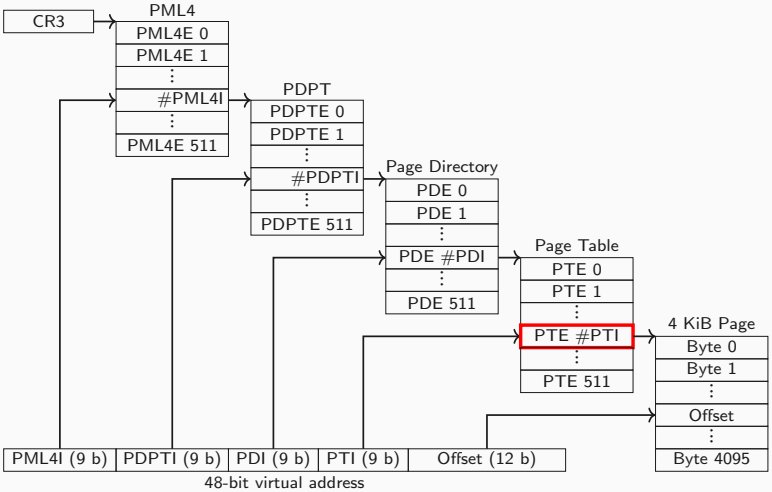


- CPU support virtual address spaces to isolate processes
- Physical memory is organized in page frames
- Virtual memory pages are mapped to page frames using page tables

Address Translation on x86-64



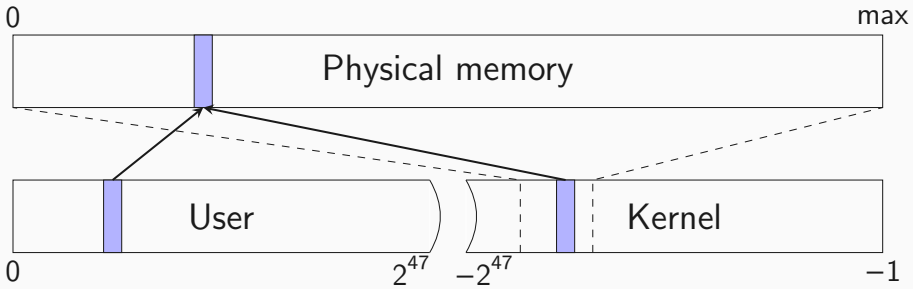
Address Translation on x86-64



Page Table Entry

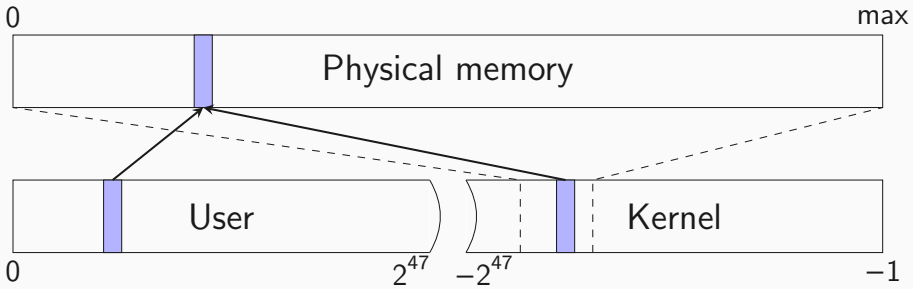
- User/Supervisor bit defines in which privilege level the page can be accessed

Direct-physical map



- Kernel is typically mapped into every address space

Direct-physical map

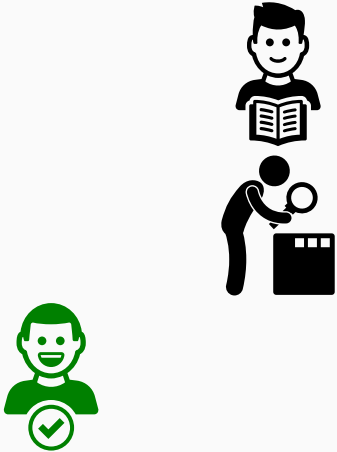


- Kernel is typically mapped into every address space
- Entire physical memory is mapped in the kernel

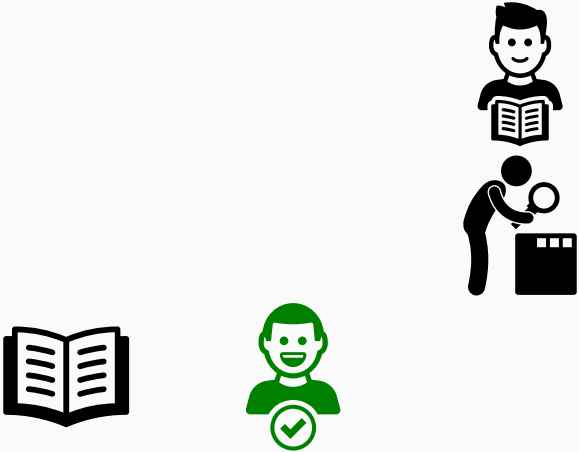
Loading an address

Loading an address

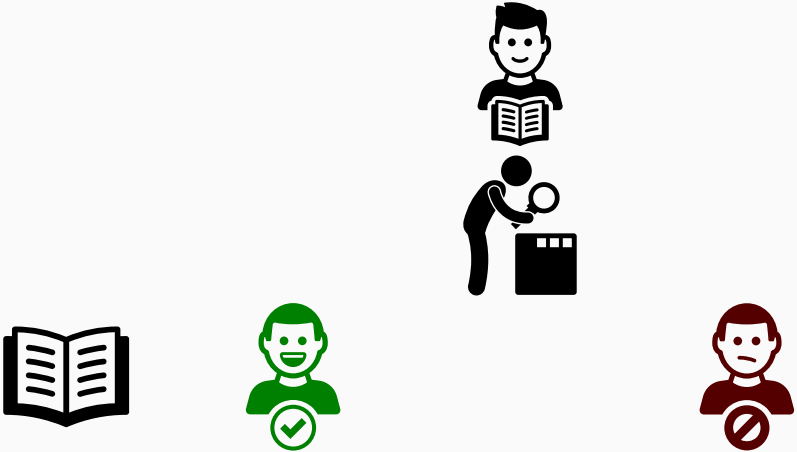
Loading an address



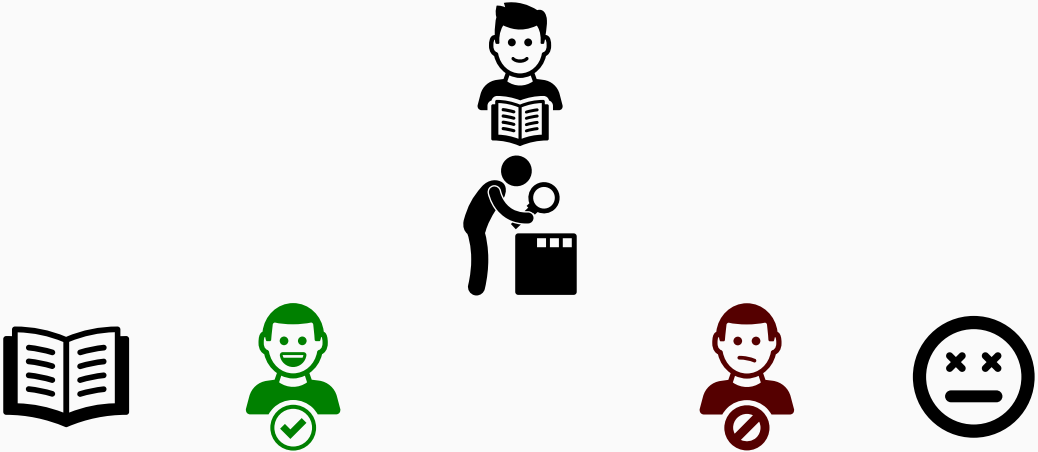
Loading an address



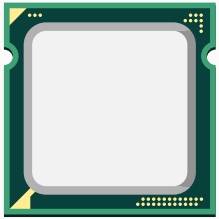
Loading an address



Loading an address

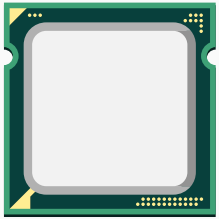


Architecture and Microarchitecture



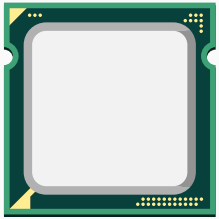
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, . . .)

Architecture and Microarchitecture



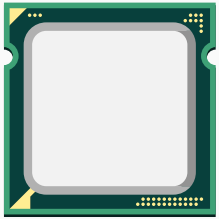
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, . . .)
- Serves as the interface between hardware and software

Architecture and Microarchitecture



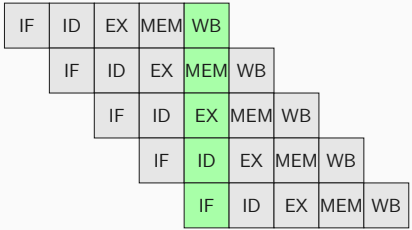
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, . . .)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

Architecture and Microarchitecture



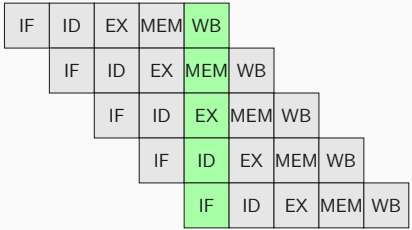
- Instruction Set Architecture (ISA) is an abstract model of a computer (x86, ARMv8, SPARC, . . .)
- Serves as the interface between hardware and software
- Microarchitecture is an actual implementation of the ISA

In-Order Execution



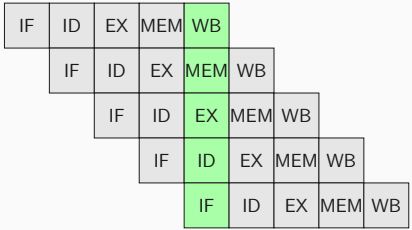
- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache

In-Order Execution



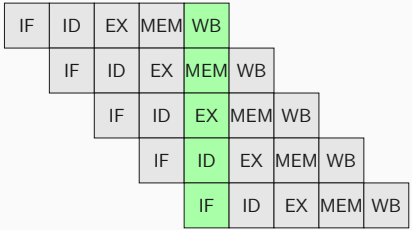
- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache
  - decoded (`ID`)

In-Order Execution



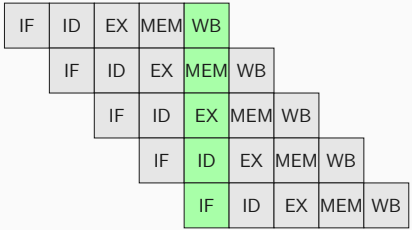
- Instructions are...
  - fetched (`IF`) from the L1 Instruction Cache
  - decoded (`ID`)
  - executed (`EX`) by execution units

In-Order Execution



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

In-Order Execution



- 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


```
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

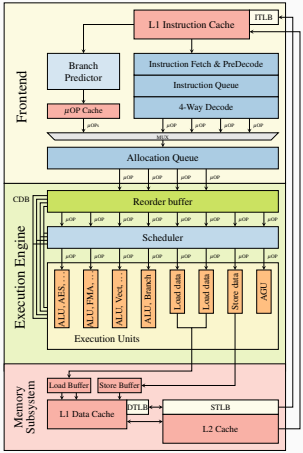
Dependency

```
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);
```

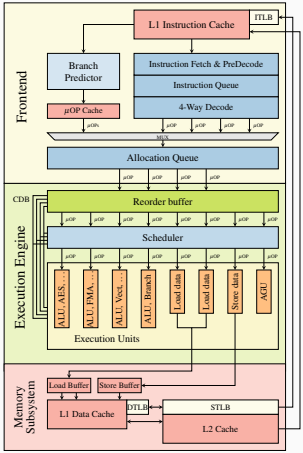
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end

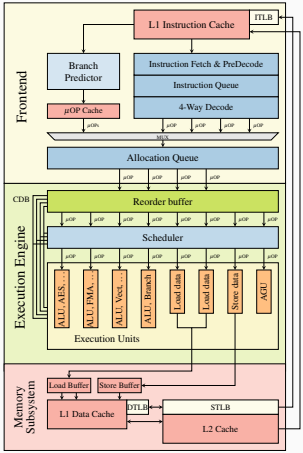
Out-of-Order Execution



Instructions are

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

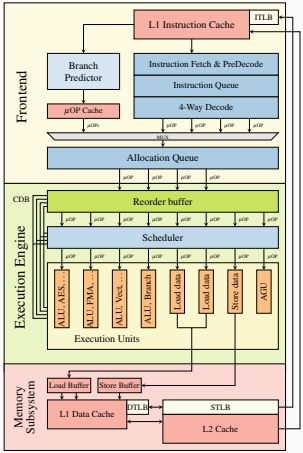
Out-of-Order Execution



Instructions are

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

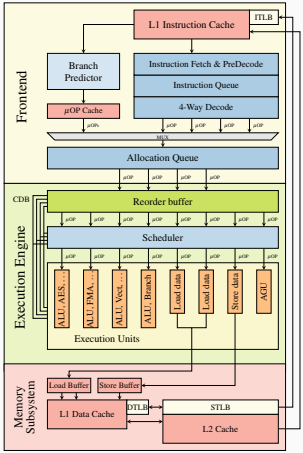
Out-of-Order Execution



Instructions

- are executed out-of-order

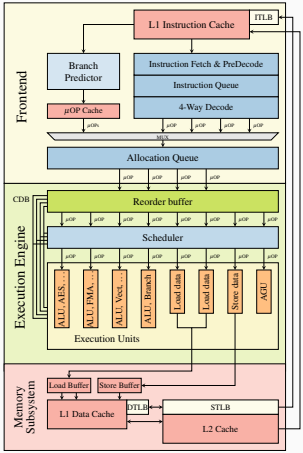
Out-of-Order Execution



Instructions

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

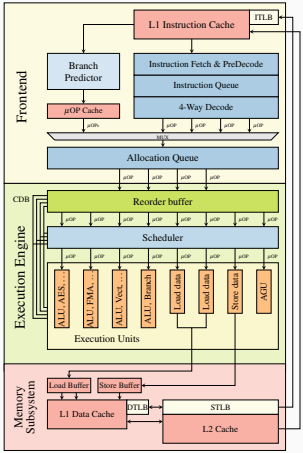
Out-of-Order Execution



Instructions

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

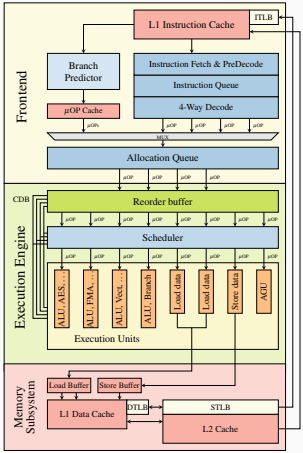
Out-of-Order Execution



Instructions

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

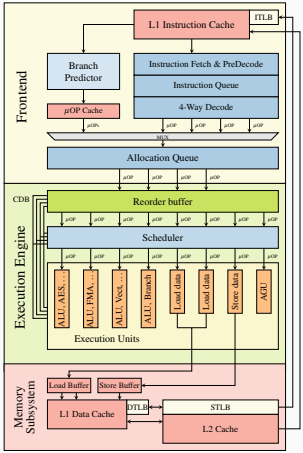
Out-of-Order Execution



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

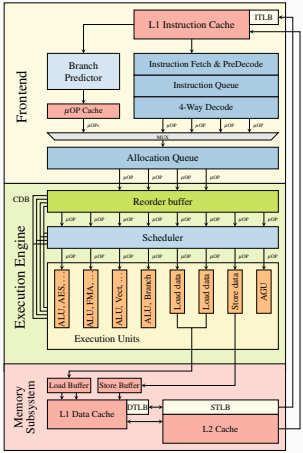
Out-of-Order Execution



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 . . .

Getting started...

Getting started...

- New code

```
char data = 'S'; // a "secret" value
// ...
*(volatile char*) 0;
array[data * 4096] = 0;
```

Getting started...

- New code

```
char data = 'S'; // a "secret" value
// ...
*(volatile char*) 0;
array[data * 4096] = 0;
```

- Luckily we know how to catch a segfault

Getting started...

- New code

```
char data = 'S'; // a "secret" value
// ...
*(volatile char*) 0;
array[data * 4096] = 0;
```

- Luckily we know how to catch a segfault
- Then check whether any part of `array` is cached

Checking the array

- Flush+Reload over all pages of the array

Time to code

Meltdown

- Add another layer of indirection to test

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

Meltdown

- Add another layer of indirection to test

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

Which address?

- Check /proc/kallsyms

```
sudo cat /proc/kallsyms | grep banner
```

Which address?

- Check /proc/kallsyms

```
sudo cat /proc/kallsyms | grep banner
```

- or check /proc/pid/pagemap and print address

```
printf("target: %p\n",
    libsc_get_physical_address(ctx, vaddr));
```

Which address?

- Check /proc/kallsyms

```
sudo cat /proc/kallsyms | grep banner
```

- or check /proc/pid/pagemap and print address

```
printf("target: %p\n",
    libsc_get_physical_address(ctx, vaddr));
```

- or start at a random address and iterate

Time to code

Spectre-PHT (aka Spectre Variant 1)

`index = 0`

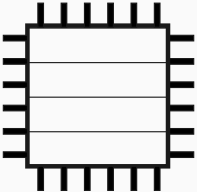
`if (index < 4)`

then → `glyph[data[index]]`

else → `{}`

Shared Memory

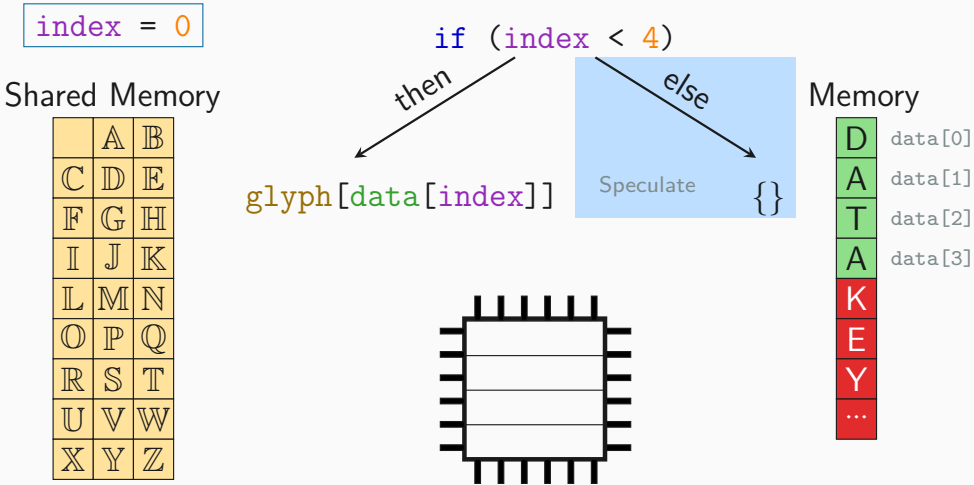
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z



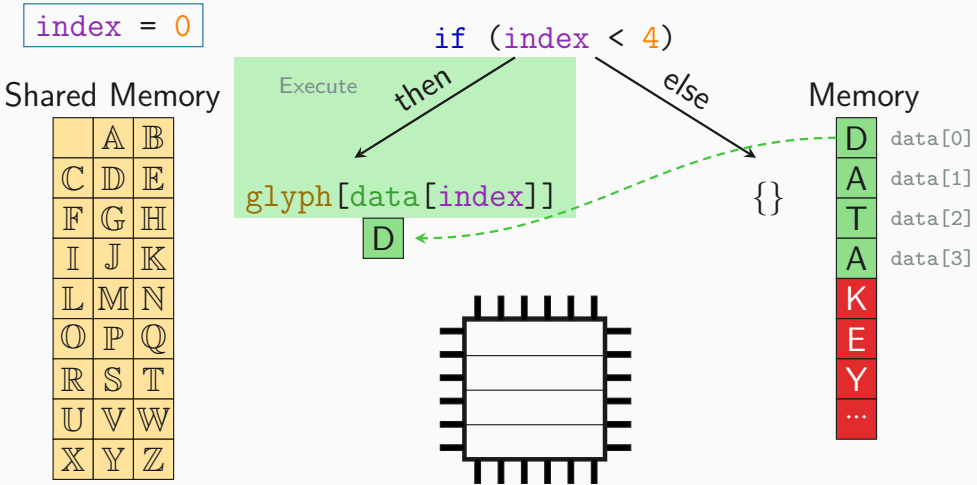
Memory

D	data[0]
A	data[1]
T	data[2]
A	data[3]
K	
E	
Y	
...	

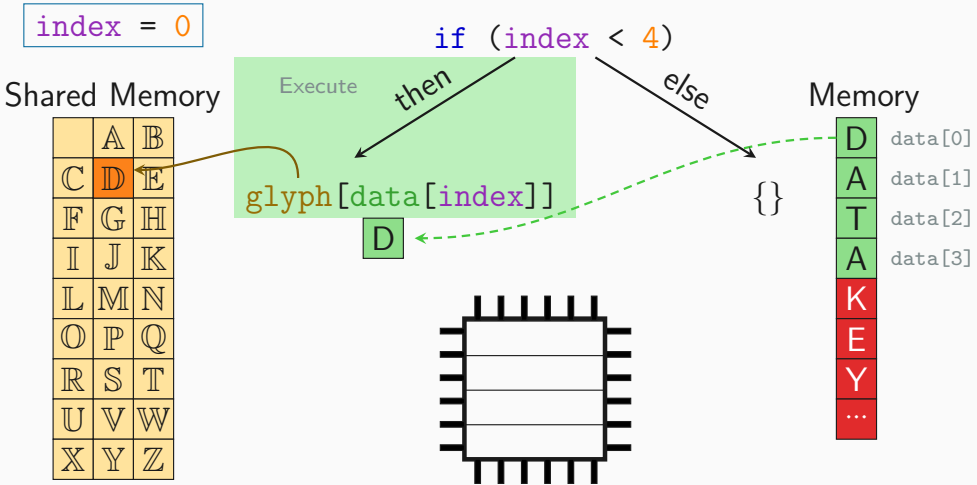
Spectre-PHT (aka Spectre Variant 1)



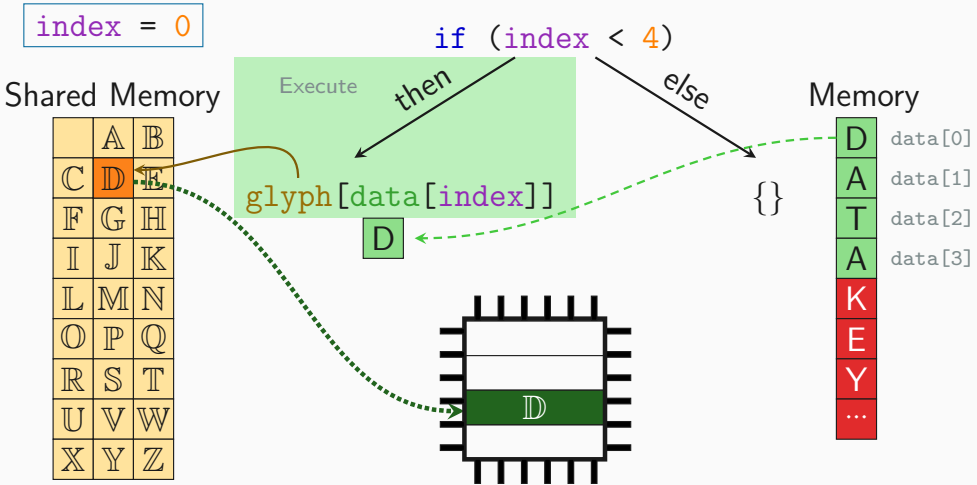
Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)


```
index = 1
```

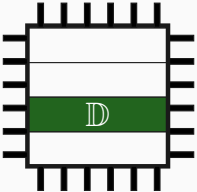
```
if (index < 4)
```

then: `glyph[data[index]]`

else: `{}`

Shared Memory

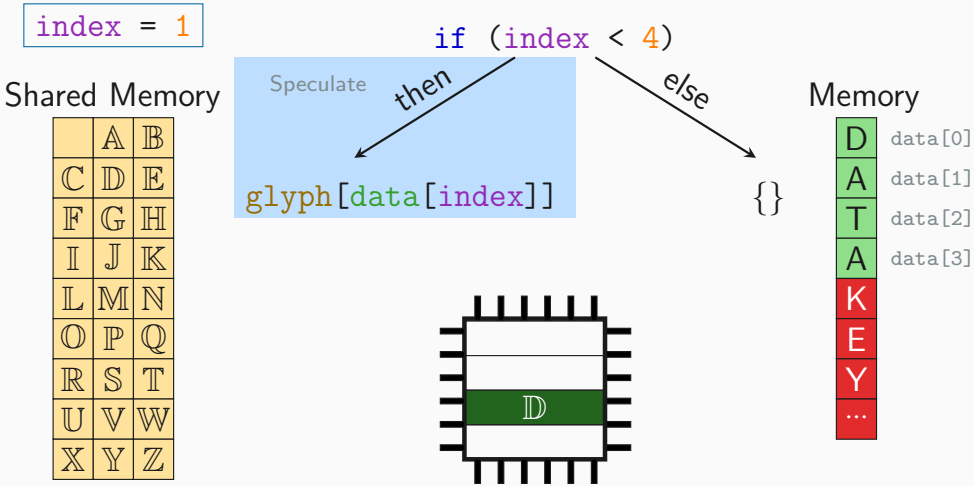
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z



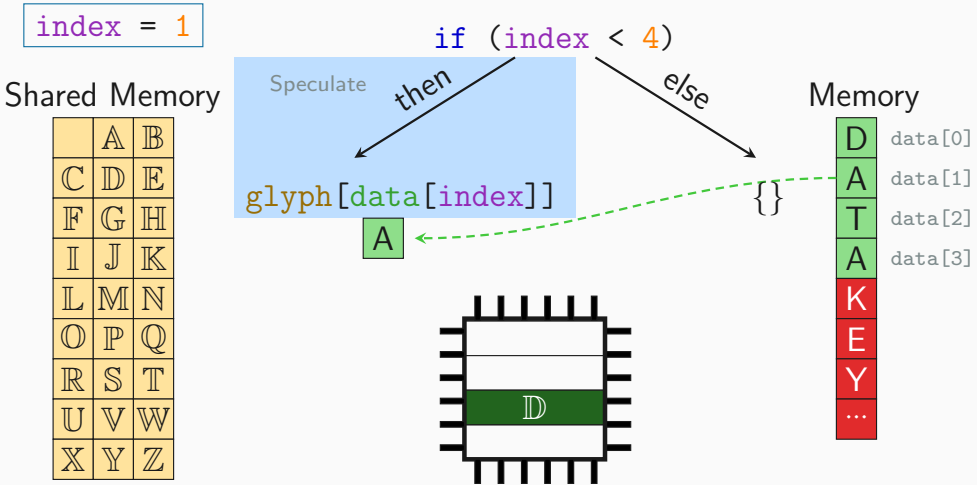
Memory

D	data[0]
A	data[1]
T	data[2]
A	data[3]
K	
E	
Y	
...	

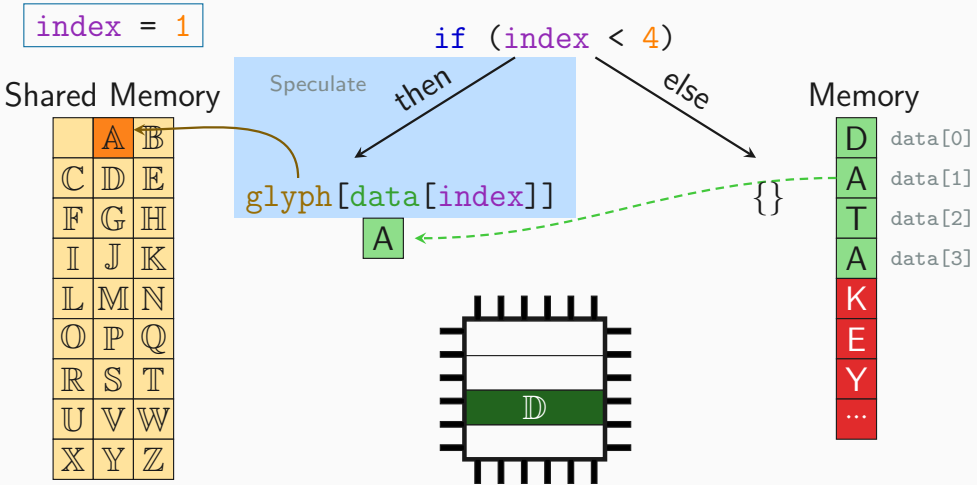
Spectre-PHT (aka Spectre Variant 1)



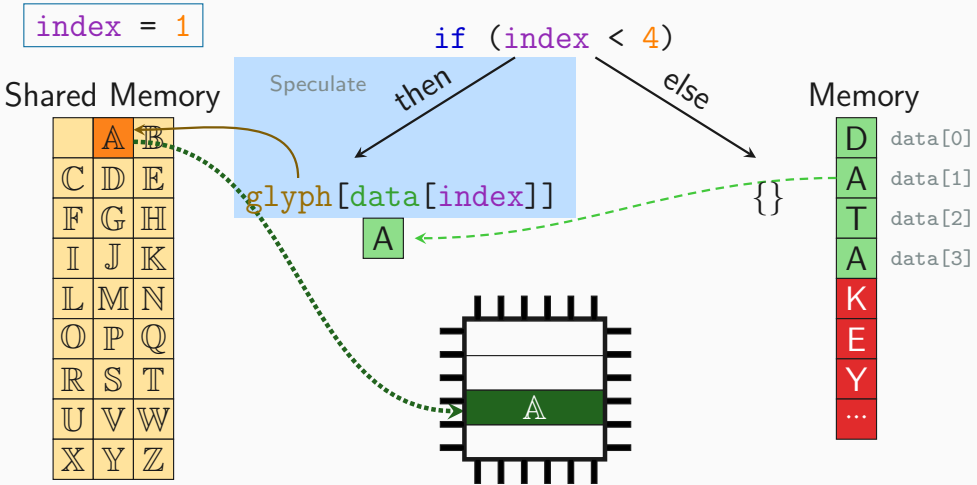
Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)


```
index = 1
```

```
if (index < 4)
```

Execute

then

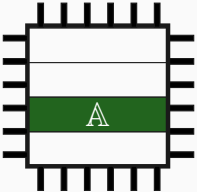
```
glyph[data[index]]
```

else

{}

Shared Memory

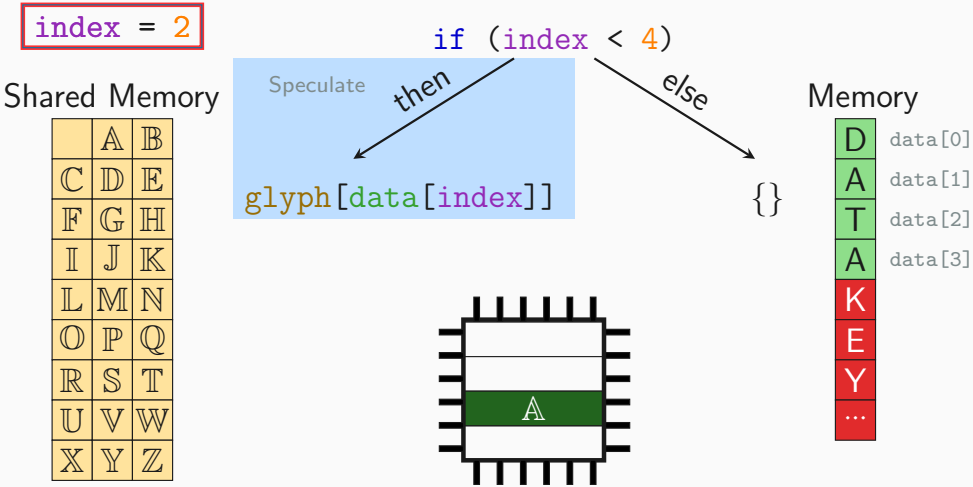
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z



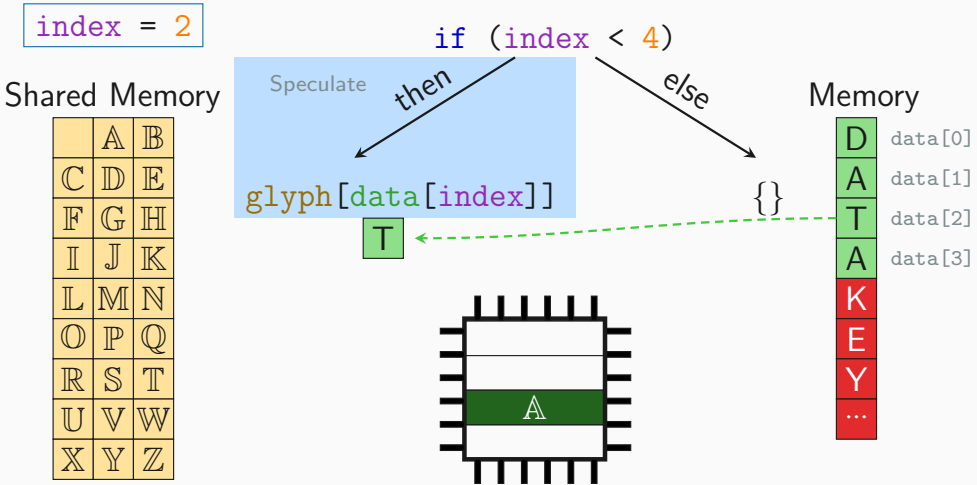
Memory

D	data[0]
A	data[1]
T	data[2]
A	data[3]
K	
E	
Y	
...	

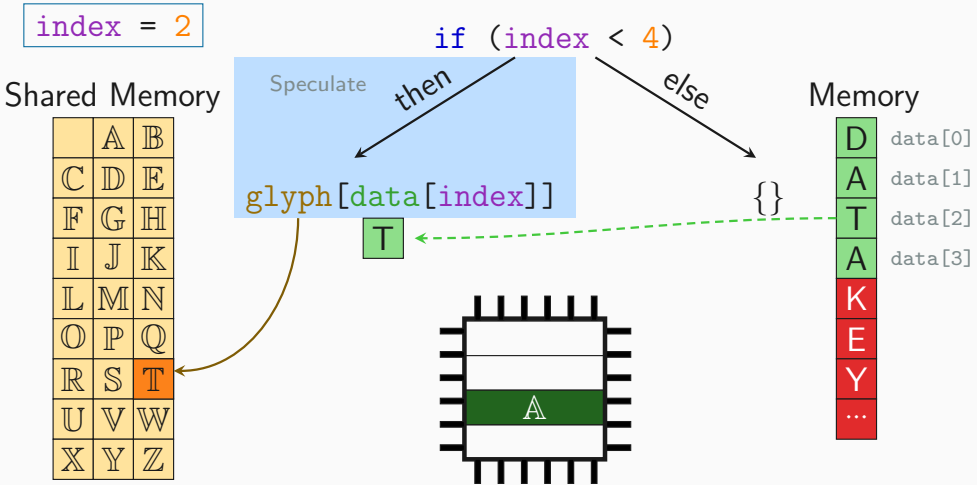
Spectre-PHT (aka Spectre Variant 1)



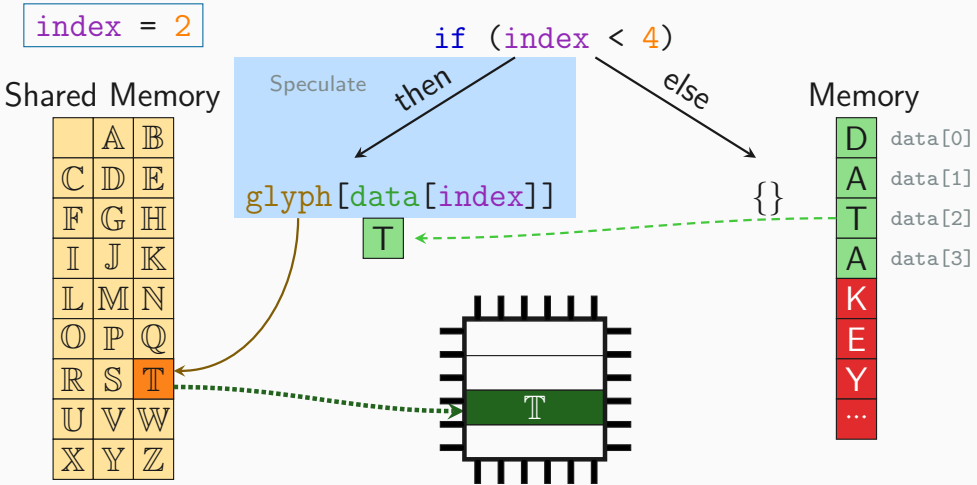
Spectre-PHT (aka Spectre Variant 1)



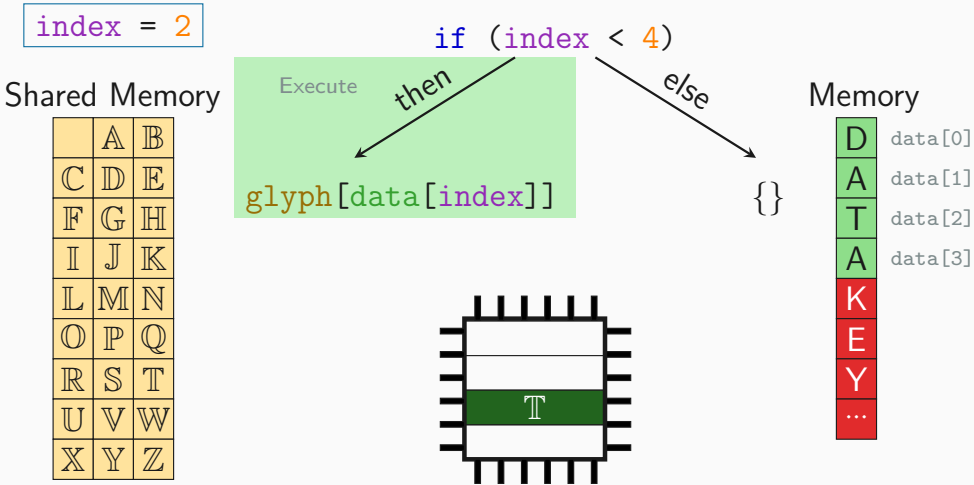
Spectre-PHT (aka Spectre Variant 1)



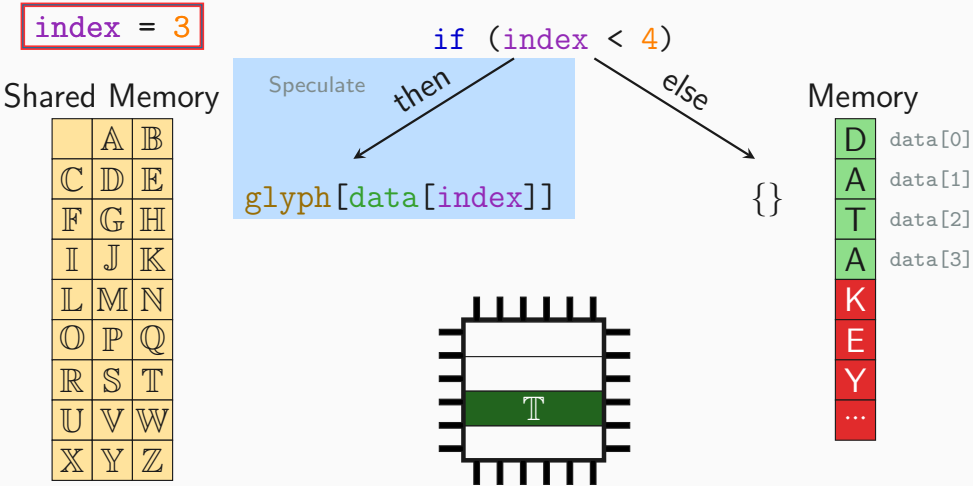
Spectre-PHT (aka Spectre Variant 1)



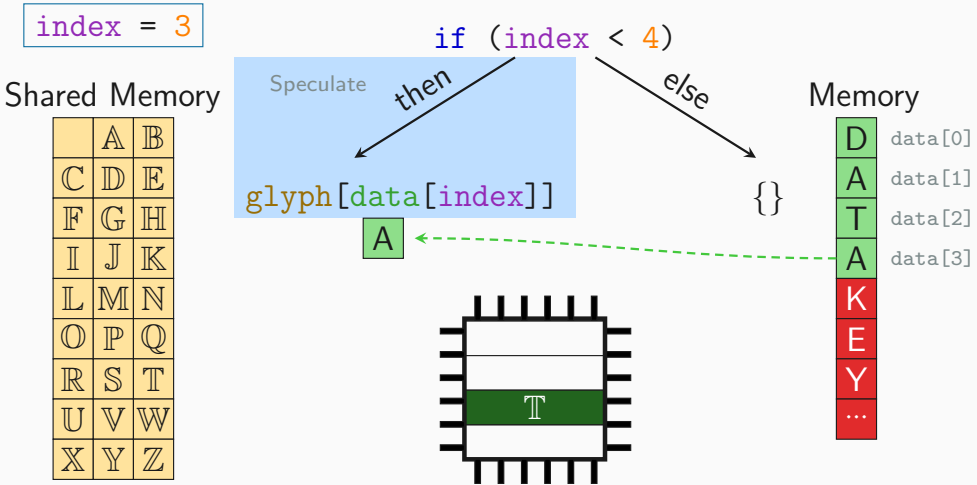
Spectre-PHT (aka Spectre Variant 1)



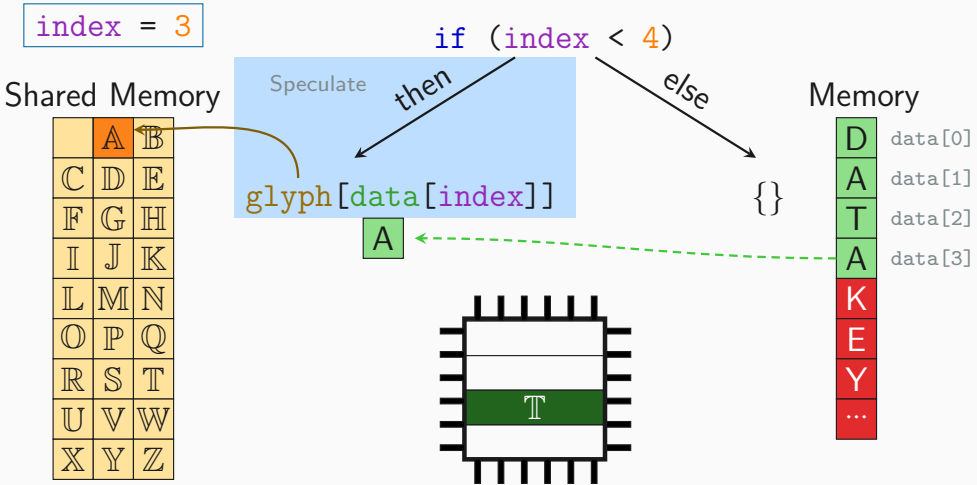
Spectre-PHT (aka Spectre Variant 1)



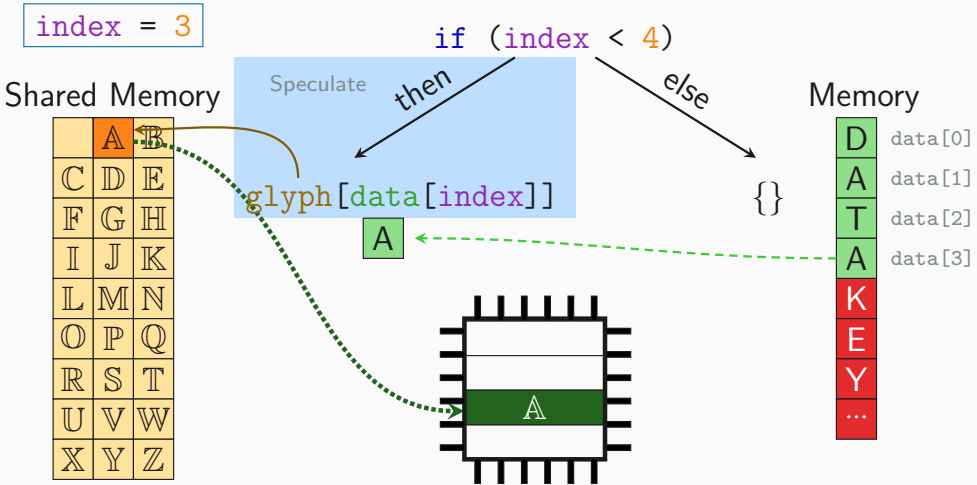
Spectre-PHT (aka Spectre Variant 1)



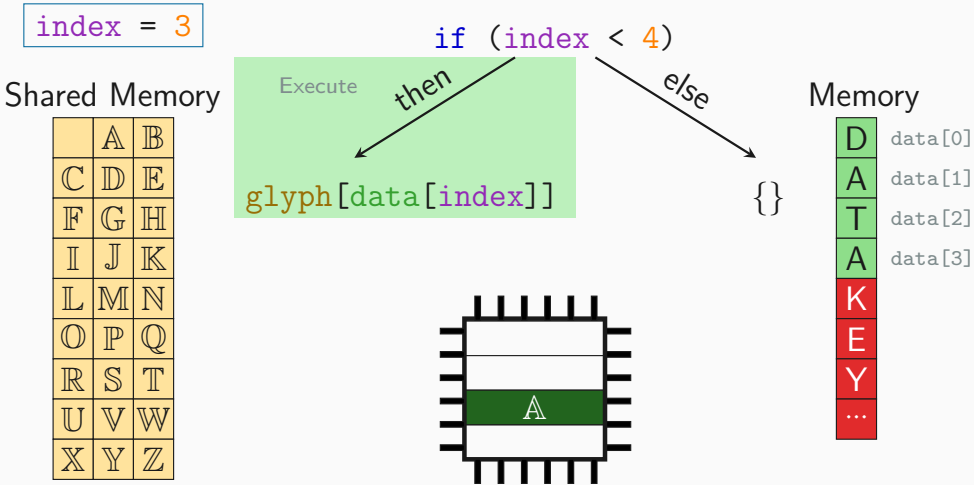
Spectre-PHT (aka Spectre Variant 1)



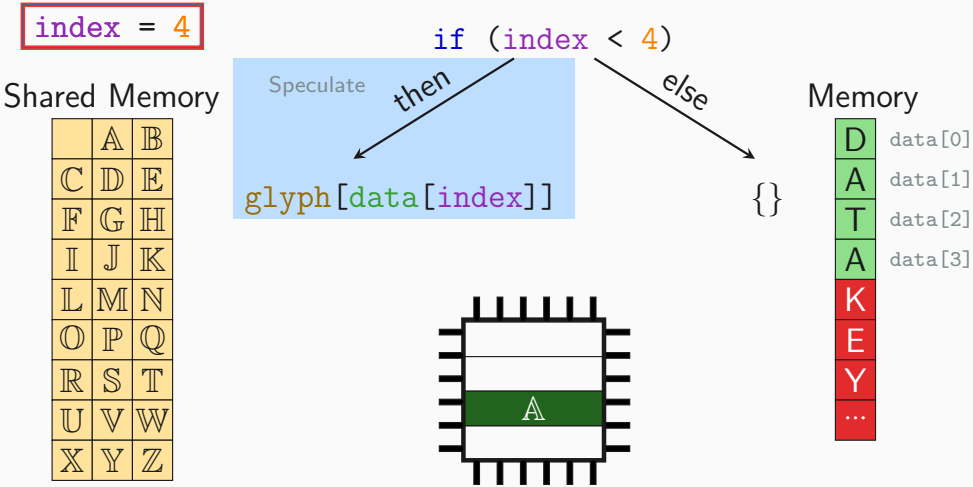
Spectre-PHT (aka Spectre Variant 1)



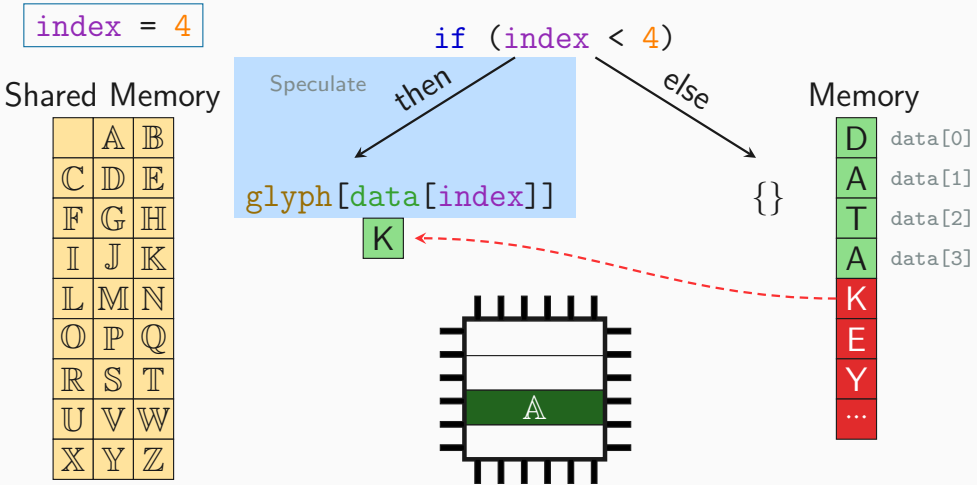
Spectre-PHT (aka Spectre Variant 1)



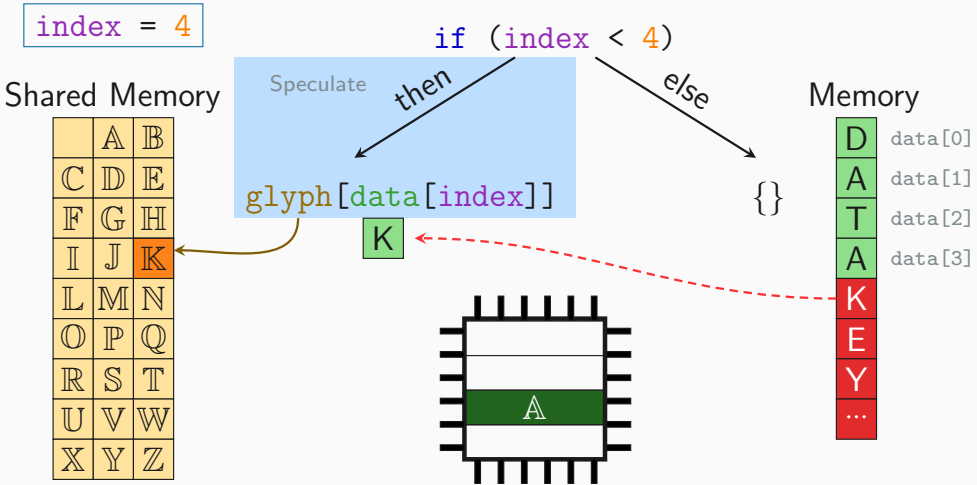
Spectre-PHT (aka Spectre Variant 1)



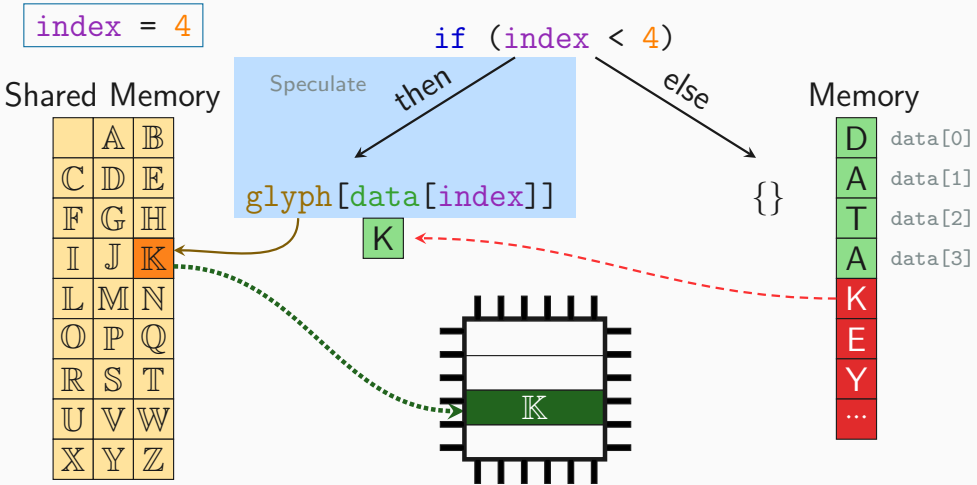
Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)



Spectre-PHT (aka Spectre Variant 1)


```
index = 4
```

```
if (index < 4)
```

then

```
glyph[data[index]]
```

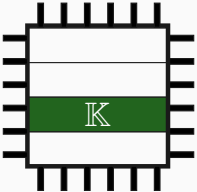
else

Execute

```
{}
```

Shared Memory

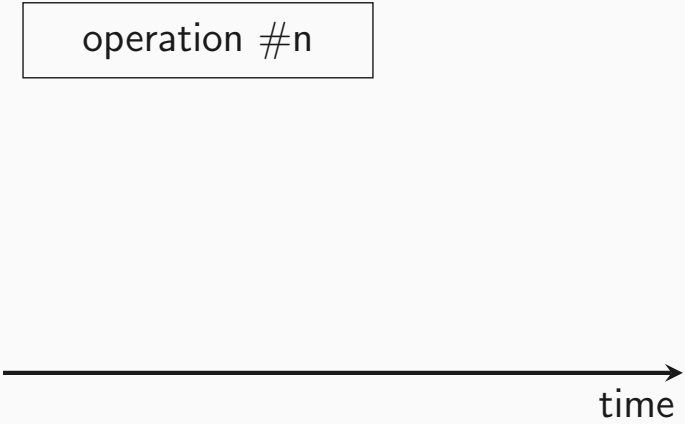
	A	B
C	D	E
F	G	H
I	J	K
L	M	N
O	P	Q
R	S	T
U	V	W
X	Y	Z



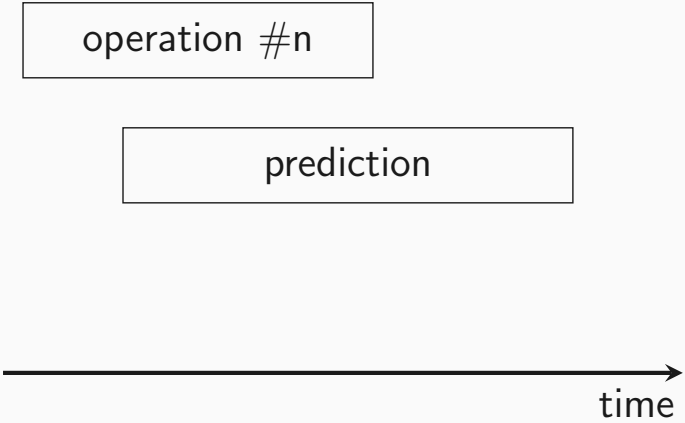
Memory

D	data[0]
A	data[1]
T	data[2]
A	data[3]
K	
E	
Y	
...	

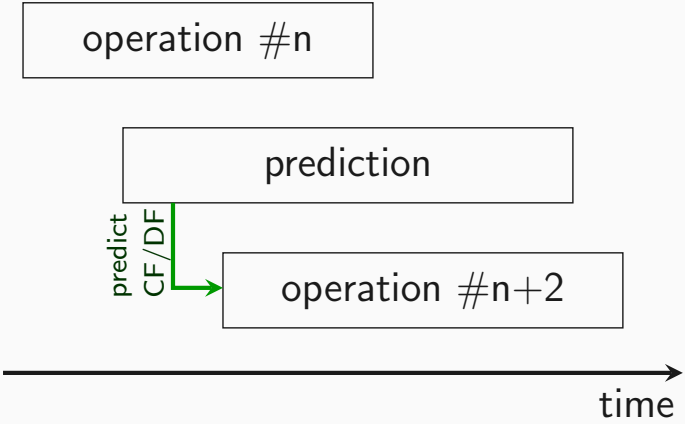
Spectre Root Cause



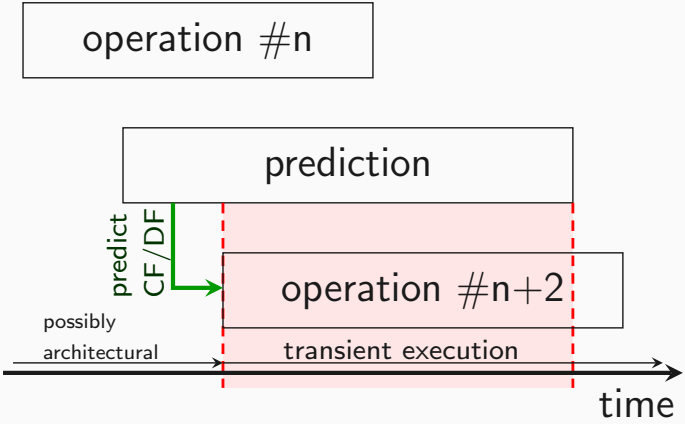
Spectre Root Cause



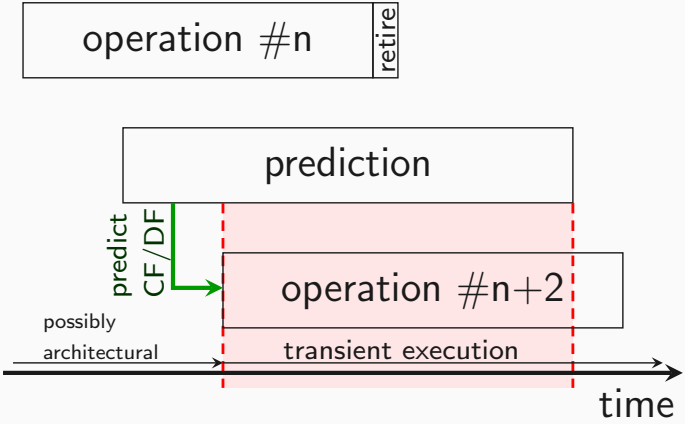
Spectre Root Cause



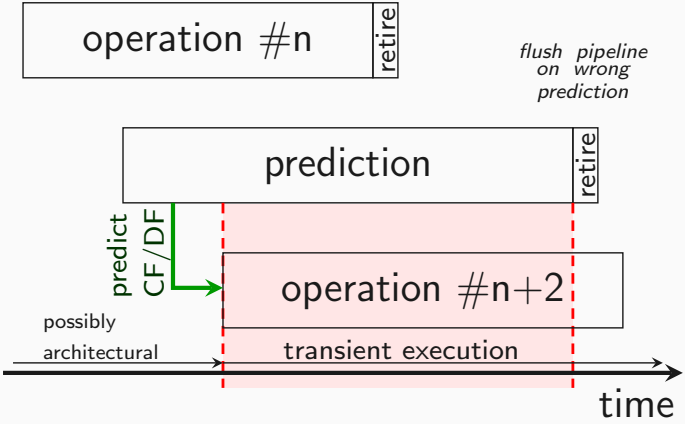
Spectre Root Cause



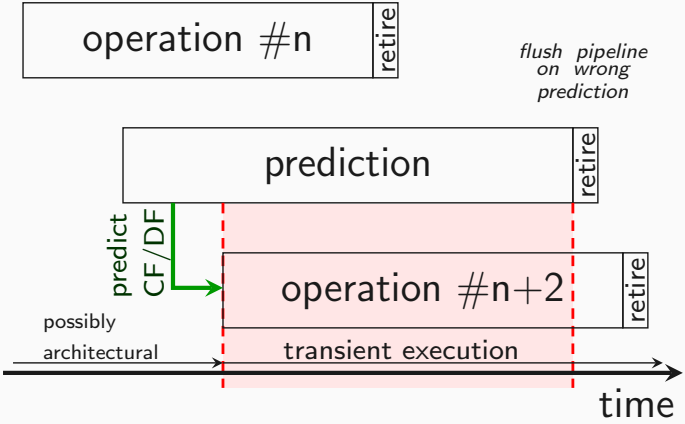
Spectre Root Cause



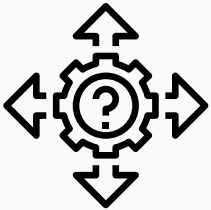
Spectre Root Cause



Spectre Root Cause

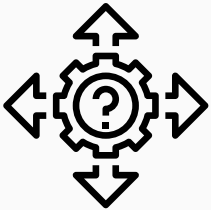


Spectre Root Cause



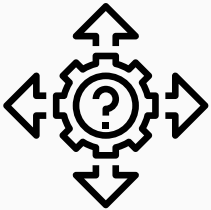
- Many predictors in modern CPUs

Spectre Root Cause



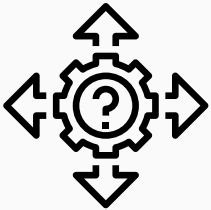
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)

Spectre Root Cause



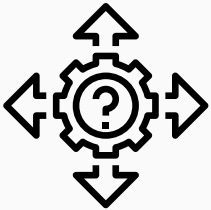
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)

Spectre Root Cause



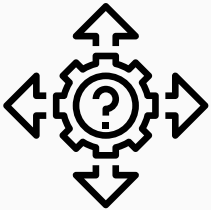
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)

Spectre Root Cause



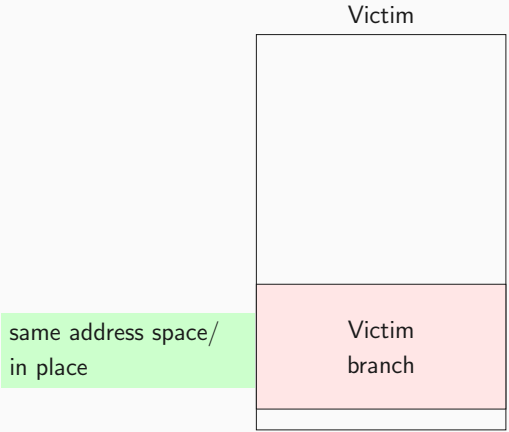
- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)

Spectre Root Cause

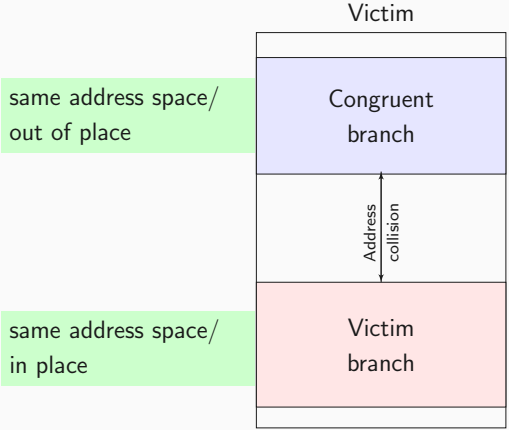


- Many predictors in modern CPUs
  - Branch taken/not taken (PHT)
  - Call/Jump destination (BTB)
  - Function return destination (RSB)
  - Load matches previous store (STL)
- Most are even shared among processes

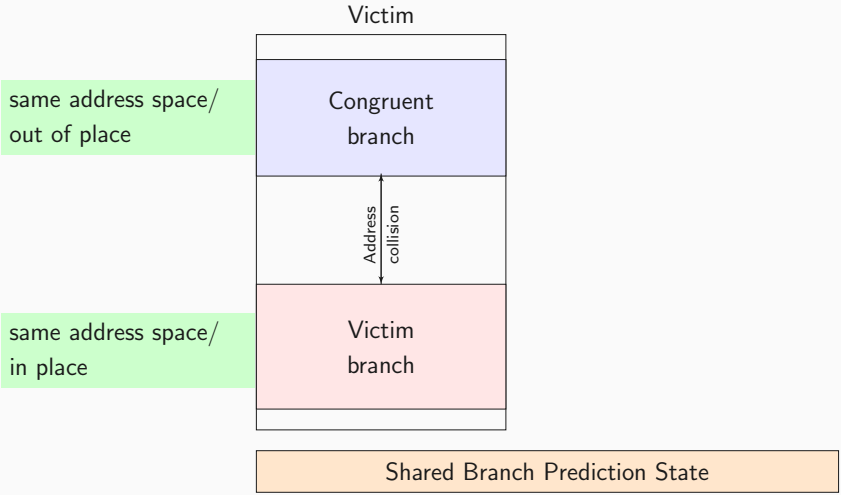
Spectre Mistraining



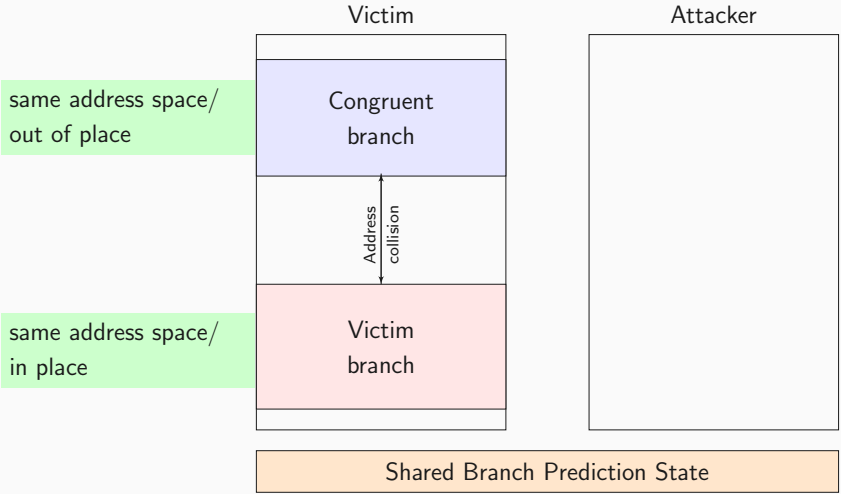
Spectre Mistraining



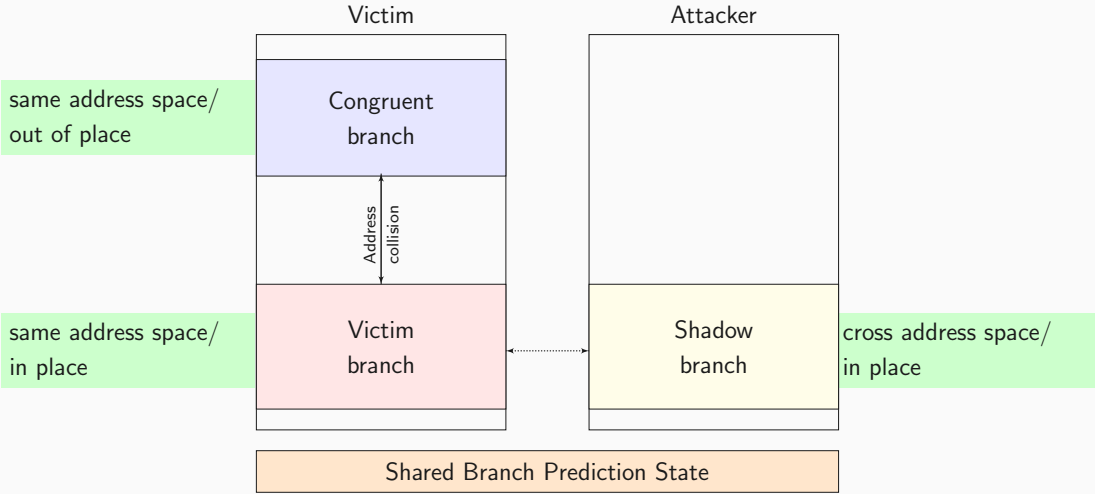
Spectre Mistraining



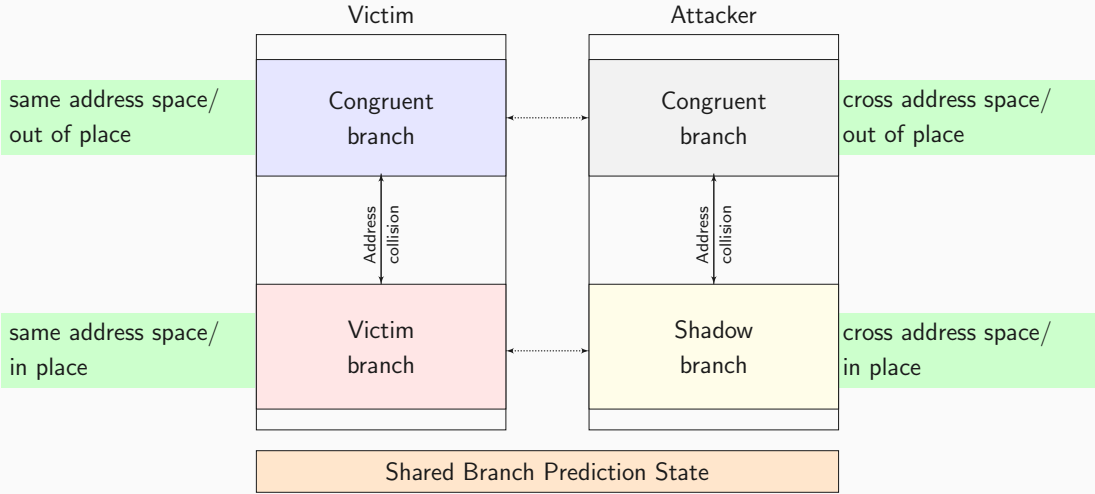
Spectre Mistraining



Spectre Mistraining



Spectre Mistraining



Time to code

Side-Channel Lab II

Michael Schwarz

Security Week Graz 2019

D. Gruss, C. Maurice, K. Wagner, and S. Mangard. Flush+Flush: A Fast and Stealthy Cache Attack. In: DIMVA. 2016.

M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In: USENIX Security Symposium. 2016.

F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In: S&P. 2015.

C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. Alberto Boano, S. Mangard, and K. Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In: NDSS. 2017.