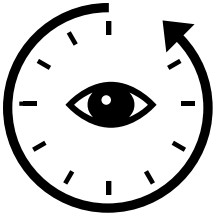
Meltdown & Spectre

Side-channels considered h**ARM**ful

Qualcomm Mobile Security Summit 2018

17 May, 2018 - San Diego, CA

Moritz Lipp (@mlqxyz) Michael Schwarz (@misc0110)



Qualcomm Mobile Security Summit 2017

With great speed comes great leakage

How processor performance is tied to side-channel leakage

Moritz Lipp & Daniel Gruss, Graz University of Technology

May 18, 2017 — Qualcomm Mobile Security Summit

Flashback: Side-channel attacks

- Safe software infrastructure does not mean safe execution
- Information leaks because of the underlying hardware
- Side-channel attacks exploit unintentional information leakage by side-effects
  - Power consumption
  - Execution time
  - CPU cache
  - ...
- Performance optimizations often induce side-channel leakage

Outline

Last year

- Leaking keystroke timings via the cache
- Leaking AES keys from the cache
- Covertly sending data through the cache
- Rowhammer: flipping bits in DRAM

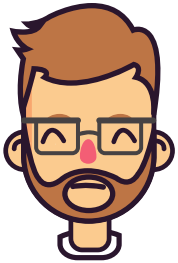
Outline

Last year

- Leaking keystroke timings via the cache
- Leaking AES keys from the cache
- Covertly sending data through the cache
- Rowhammer: flipping bits in DRAM

This year

- More leakage!
- Build upon the tools of last year
- Leaking arbitrary memory content
- Leaking privileged register content

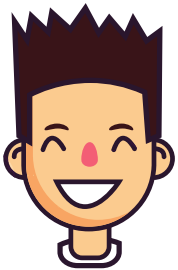


Moritz Lipp

PhD student @ Graz University of Technology

@mlqxyz

mail@mlq.me



Michael Schwarz

PhD student @ Graz University of Technology

@misc0110

michael.schwarz91@gmail.com

Let's Read Kernel Memory from User Space!

Building the Code

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

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

Building the Code

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

Building the Code

- Compile and run

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

Building the Code

- Compile and run

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

- Kernel addresses are of course not accessible

Building the Code

- Compile and run

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

- Kernel addresses are of course not accessible
- Any invalid access throws an exception → segmentation fault

Building the Code

- Just catch the segmentation fault!

Building the Code

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

Building the Code

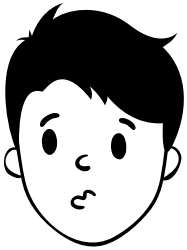
- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue

Building the Code

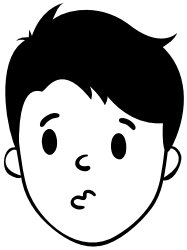
- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue
- Then we can read the value

Building the Code

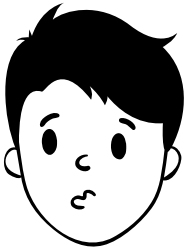
- Just catch the segmentation fault!
- We can simply install a signal handler
- And if an exception occurs, just jump back and continue
- Then we can read the value
- Sounds like a good idea



- Still no kernel memory

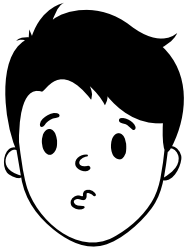


- Still no kernel memory
- Privilege checks seem to work



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

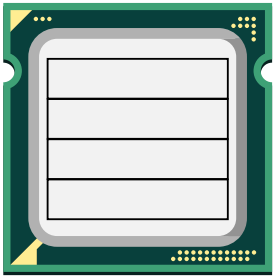
Building the Code

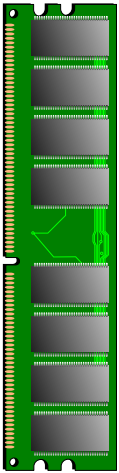


- Still no kernel memory
- Privilege checks seem to work
- Maybe it is not that straight forward
- Back to the drawing board

Caches and Cache Attacks

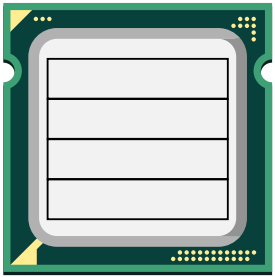
```
printf("%d", i);
printf("%d", i);
```

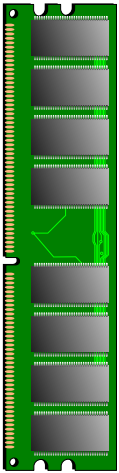




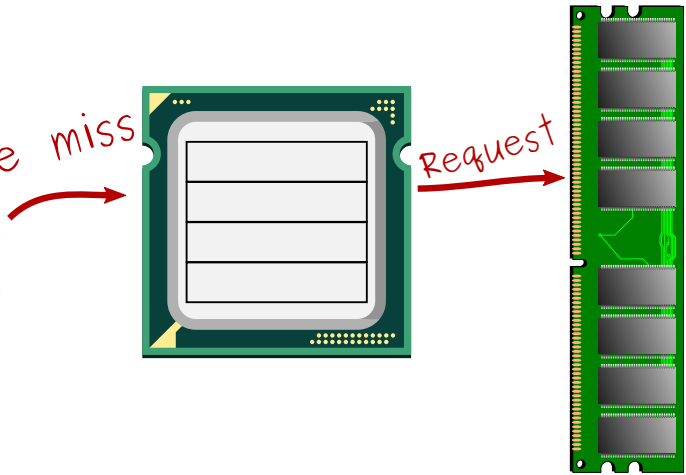
```
printf("%d", i);
printf("%d", i);
```

Cache miss

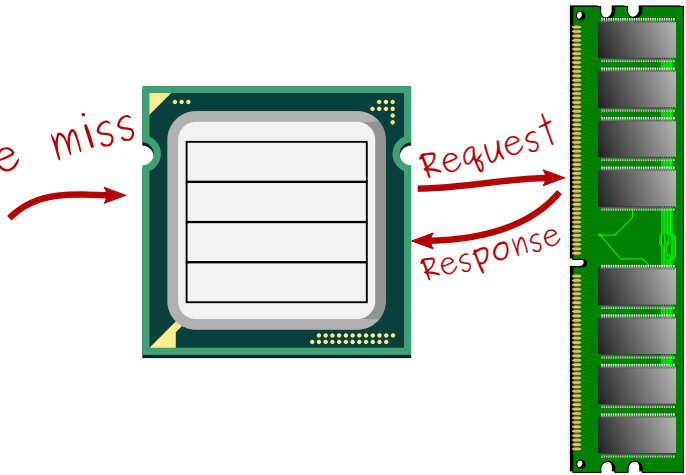




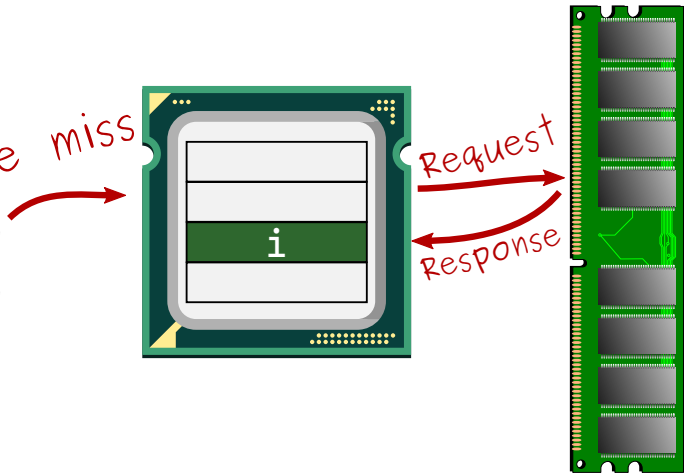
```
printf("%d", i);
printf("%d", i);
```

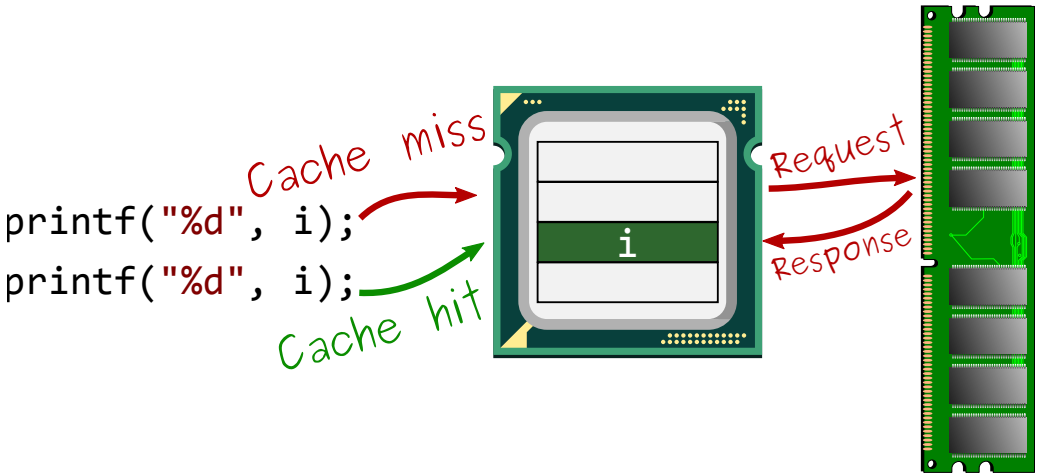


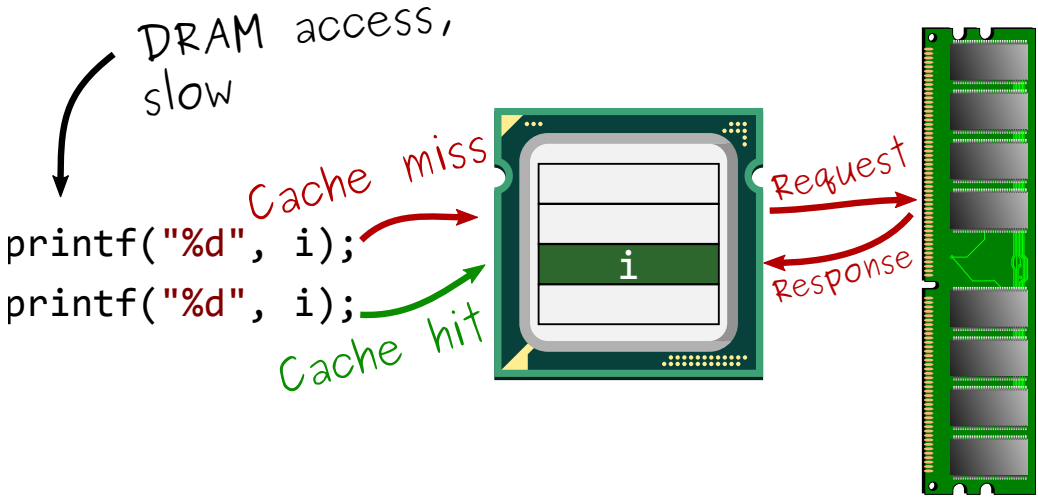
```
printf("%d", i);
printf("%d", i);
```

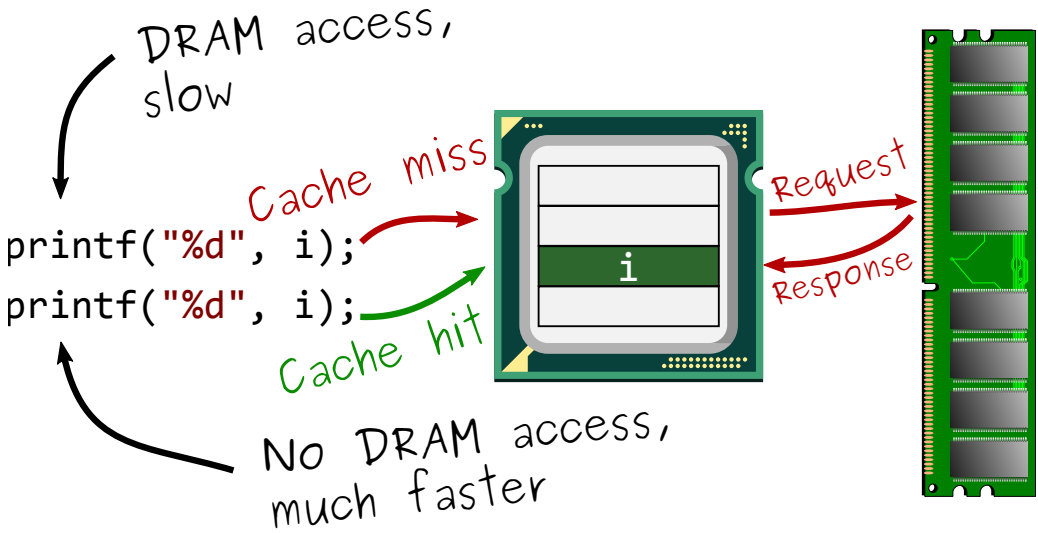


```
printf("%d", i);
printf("%d", i);
```

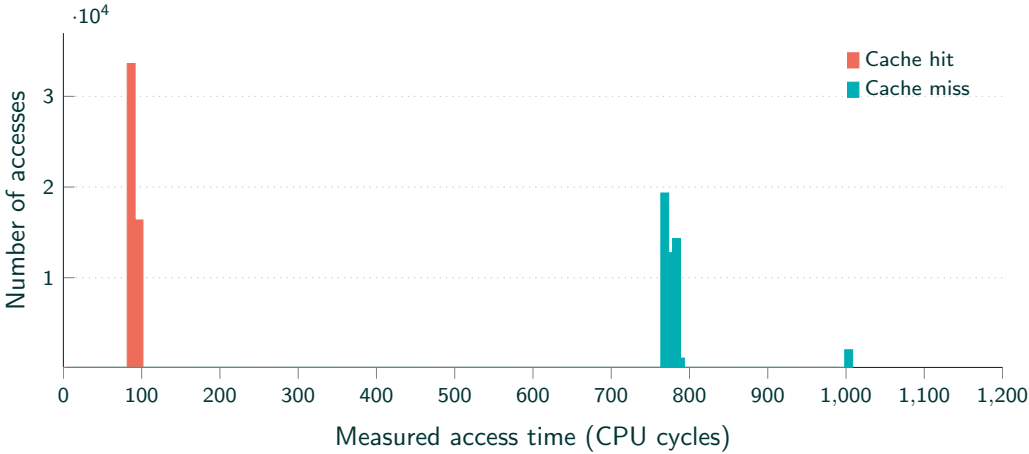



printf("%d", i);
printf("%d", i);
Cache miss
Cache hit
i
Request
Response


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

Memory Access Latency

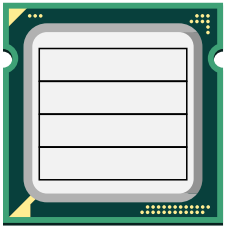


Shared Memory

ATTACKER

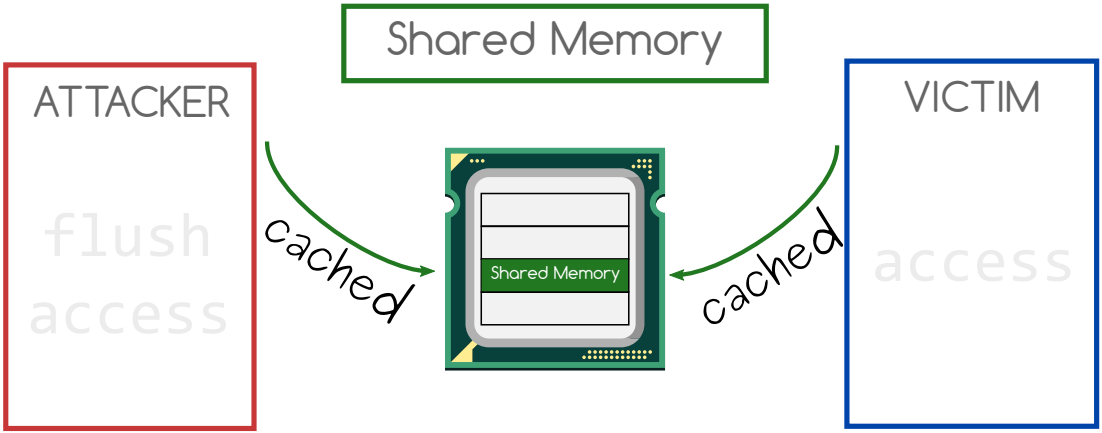
flush

access



VICTIM

access

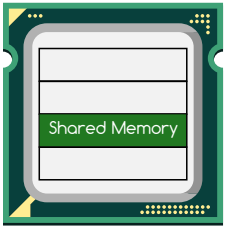

Shared Memory
ATTACKER
flush
access
cached
Shared Memory
cached
VICTIM
access

Shared Memory

ATTACKER

flush

access



VICTIM

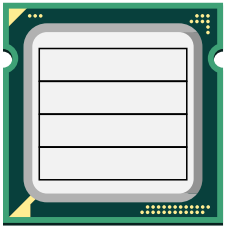
access

Shared Memory

ATTACKER

flush

access



VICTIM

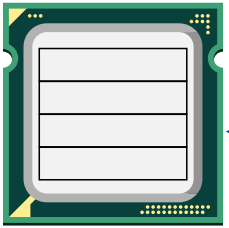
access

Shared Memory

ATTACKER

flush

access



VICTIM

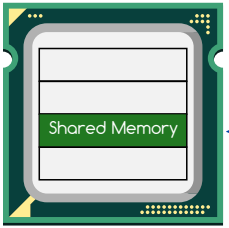
access

Shared Memory

ATTACKER

flush

access



VICTIM

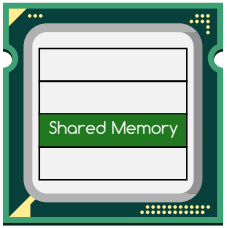
access

Shared Memory

ATTACKER

flush

access



VICTIM

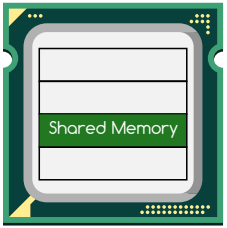
access

Shared Memory

ATTACKER

flush

access



VICTIM

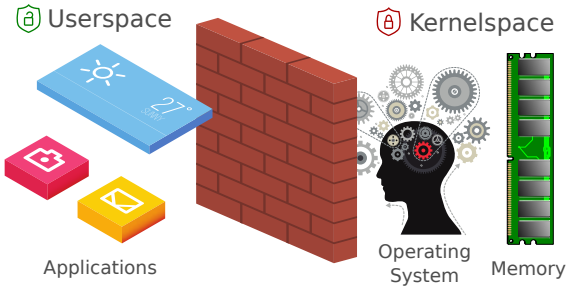
access

fast if victim accessed data,
slow otherwise

Operating Systems 101

Memory Isolation

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
- There is only a well-defined interface called system calls



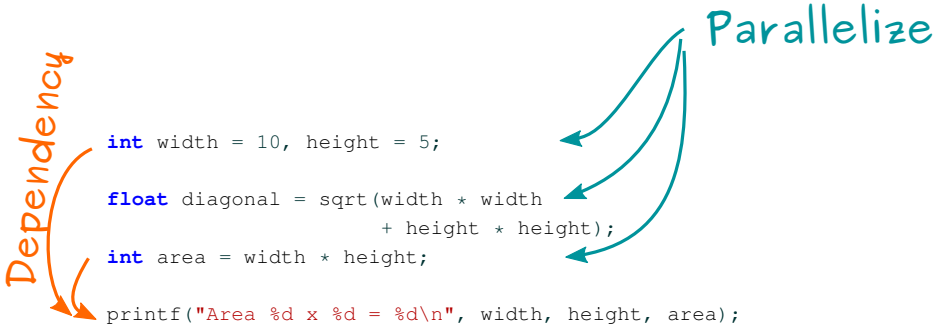
Out-of-order Execution

Out-of-order Execution

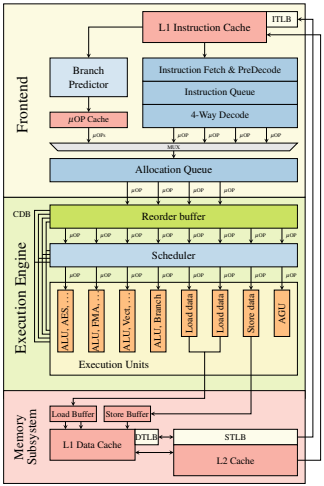
```
int width = 10, height = 5;

float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```

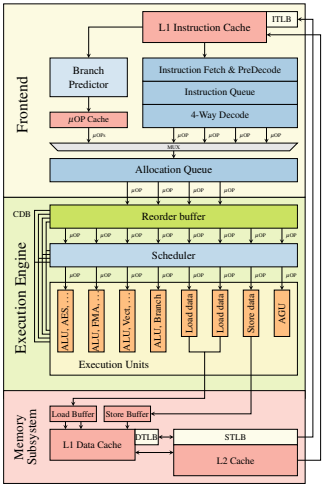

Parallelize
Dependency
int width = 10, height = 5;
float diagonal = sqrt(width * width
+ height * height);
int area = width * height;
printf("Area %d x %d = %d\n", width, height, area);

Out-of-order Execution



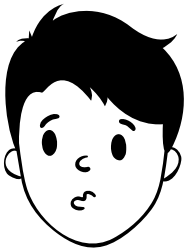
- Instructions are fetched and decoded in the front-end
- Instructions are dispatched to the backend
- Instructions are processed by individual execution units

Out-of-order Execution



- Instructions are executed out-of-order
- Instructions wait until their dependencies are ready
  - Later instructions might execute prior earlier instructions
- Instructions retire in-order
  - State becomes architecturally visible

Reading memory



- If an application reads memory, ...
  - ... permissions are checked
  - ... data is loaded
- If an application tries to read inaccessible memory, ...
  - ... an error occurs
  - ... application is stopped
- But what happens if the checks are reordered?
- Would we know?

- Adapted code

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

- Adapted code

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

- `volatile` because compiler was not happy

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

- Adapted code

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

- `volatile` because compiler was not happy

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

- Static code analyzer is still not happy

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

- Flush+Reload over all pages of the array

- "Unreachable" code line was actually executed

- Flush+Reload over all pages of the array

- "Unreachable" code line was actually executed
- Exception was only thrown afterwards

- Out-of-order instructions leave microarchitectural traces

Building the Code

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

Building the Code

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

Building the Code

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

- Maybe there is no permission check in transient instructions...

- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them

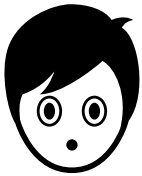
- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Add another layer of indirection to test

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

- Maybe there is no permission check in transient instructions...
- ...or it is only done when commiting them
- Add another layer of indirection to test

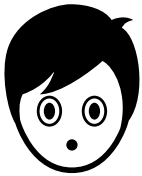
```
char data = *(char*) 0xffffffff81a000e0;
array[data * 4096] = 0;
```

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



- Flush+Reload over all pages of the array

- Index of cache hit reveals data



- Flush+Reload over all pages of the array

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

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

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory
- Entire physical memory is typically also accessible in kernel address space

- Using out-of-order execution, we can read data at any address
- Privilege checks are sometimes too slow
- Allows to leak kernel memory
- Entire physical memory is typically also accessible in kernel address space
- Works on Intel CPUs and ARM Cortex-A75

Demo

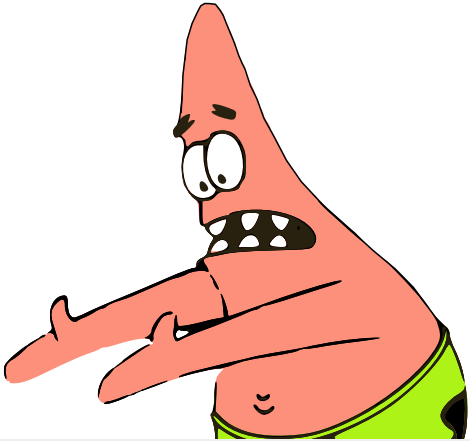
Can we fix that?

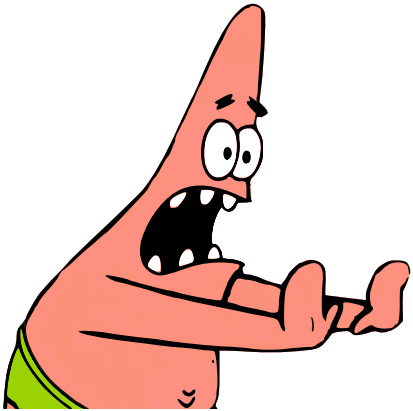
Take the kernel addresses...

- Kernel addresses in user space are a problem

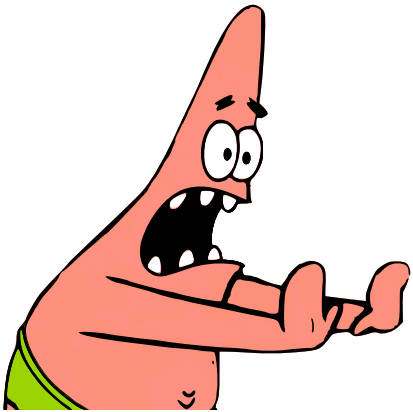
Take the kernel addresses...

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



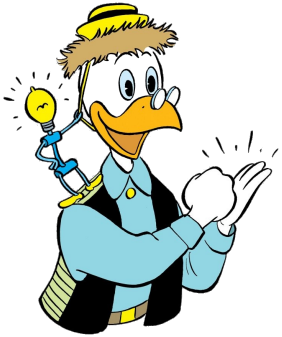


- ...and remove them if not needed?



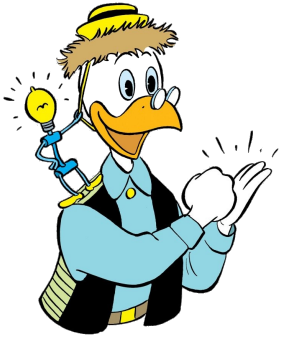
- ...and remove them if not needed?
- User accessible check in hardware is not reliable

Idea



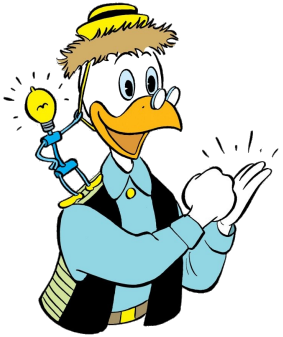
- Let's just unmap the kernel in user space

Idea

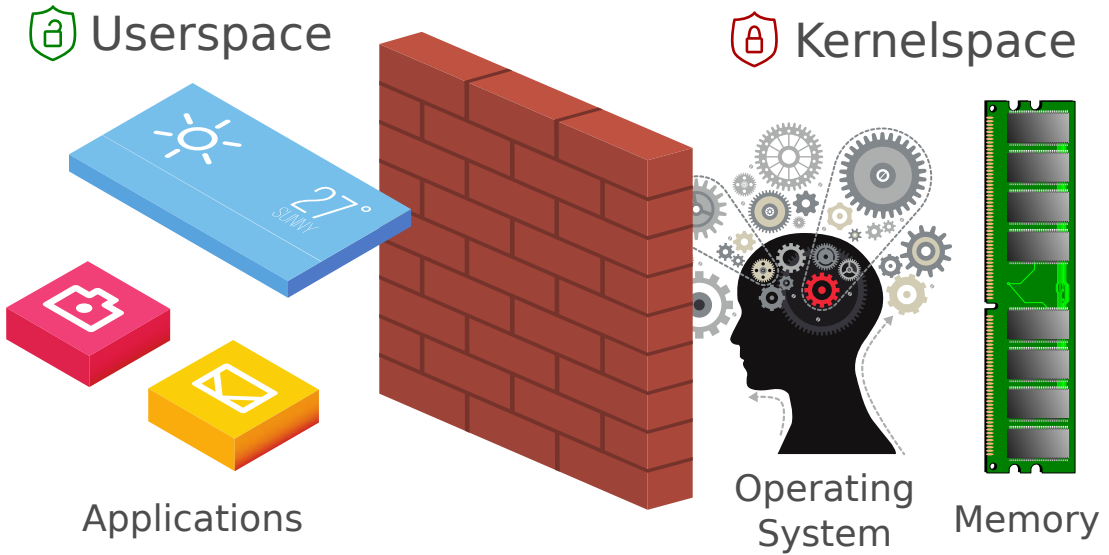


- Let's just unmap the kernel in user space
- Kernel addresses are then no longer present

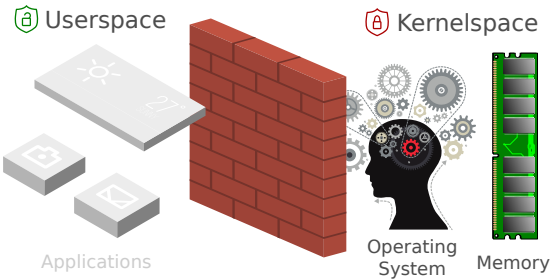
Idea



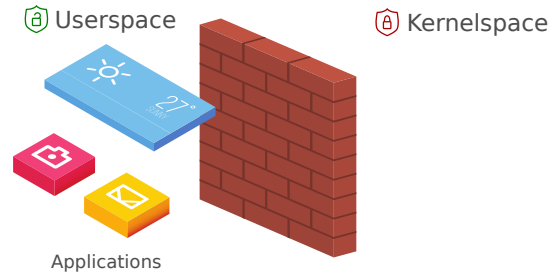
- Let's just unmap the kernel in user space
- Kernel addresses are then no longer present
- Memory which is not mapped cannot be accessed at all

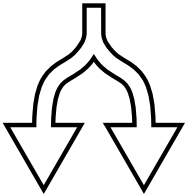

Userspace
Kernelspace
27°
Applications
Operating System
Memory

Kernel View

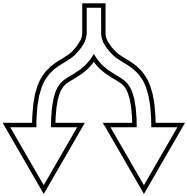


User View



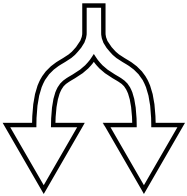


- We published KAISER in July 2017



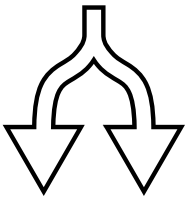
- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)

Kernel Address Space Isolation

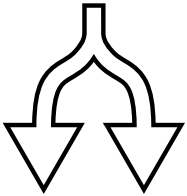


- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Kernel patches available for arm64

Kernel Address Space Isolation



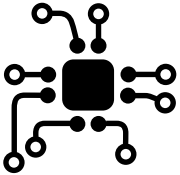
- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Kernel patches available for arm64
- Microsoft and Apple implemented similar concepts, for x86 and ARM



- We published KAISER in July 2017
- Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation)
- Kernel patches available for arm64
- Microsoft and Apple implemented similar concepts, for x86 and ARM
- All share the same idea: switching address spaces on context switch

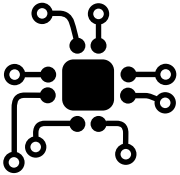
But wait, what about privileged registers?

Meltdown for Registers (Variant 3a)



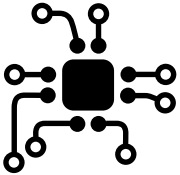
- ARM found a closely related Meltdown variant

Meltdown for Registers (Variant 3a)



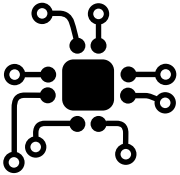
- ARM found a closely related Meltdown variant
- Read of system registers that are not accessible from current exception level

Meltdown for Registers (Variant 3a)



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

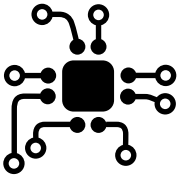
Meltdown for Registers (Variant 3a)



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

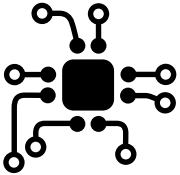
Demo

Meltdown for Registers (Variant 3a)



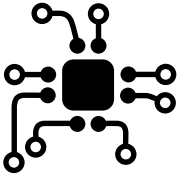
- Solution? Substitute registers with dummy values on context switch

Meltdown for Registers (Variant 3a)



- Solution? Substitute registers with dummy values on context switch
- Only necessary for virtual addresses and secrets

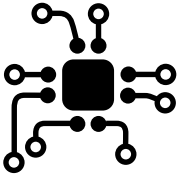
Meltdown for Registers (Variant 3a)



- Solution? Substitute registers with dummy values on context switch
- Only necessary for virtual addresses and secrets

$\rightarrow$ only a few registers affected

Meltdown for Registers (Variant 3a)



- Solution? Substitute registers with dummy values on context switch
- Only necessary for virtual addresses and secrets

$\rightarrow$ only a few registers affected

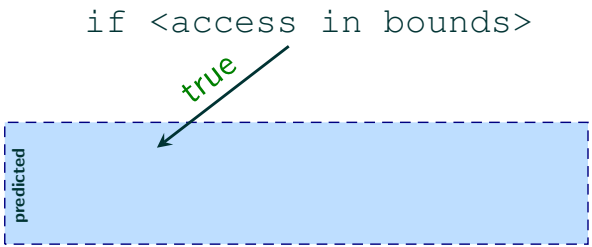
- Performance overhead should be minimal

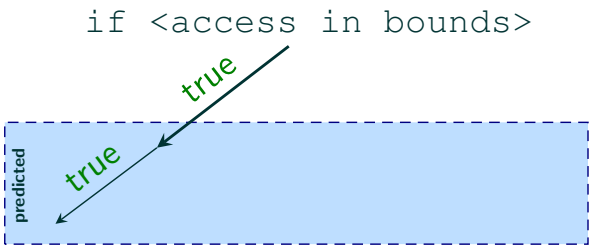
Where are Variant 1 and 2?

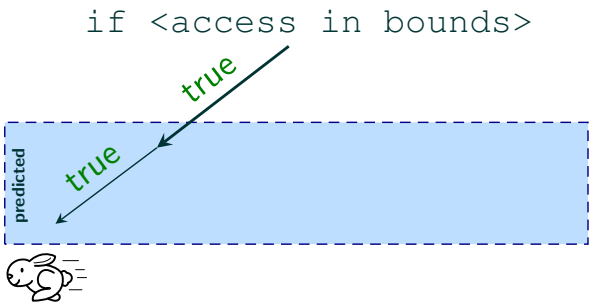
Speculative Execution

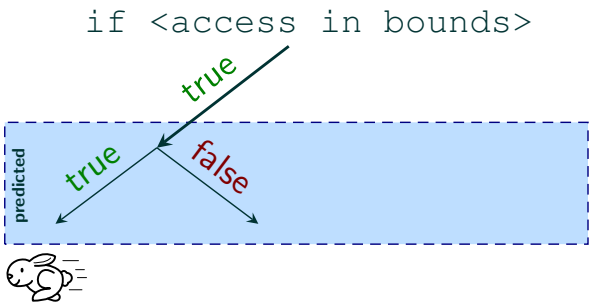
- CPU tries to predict the future (branch predictor), ...
  - ... based on events learned in the past
- Speculative execution of instructions
- If the prediction was correct, ...
  - ... very fast
  - otherwise: Discard results
- Measurable side-effects?

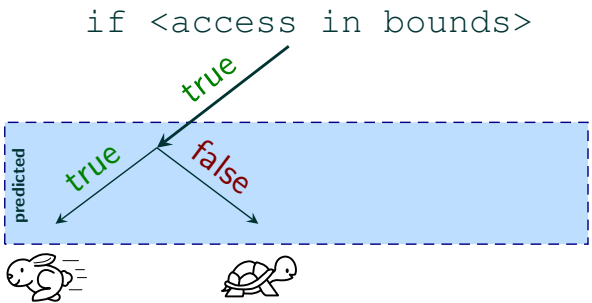
```
if <access in bounds>
```

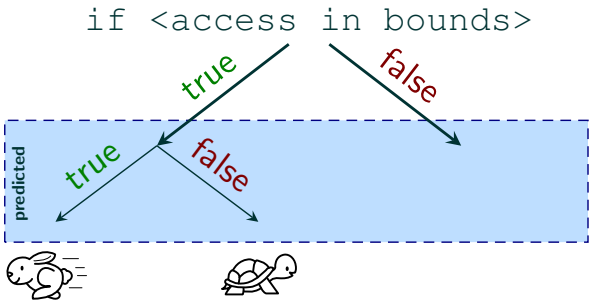

if <access in bounds>
true
predicted

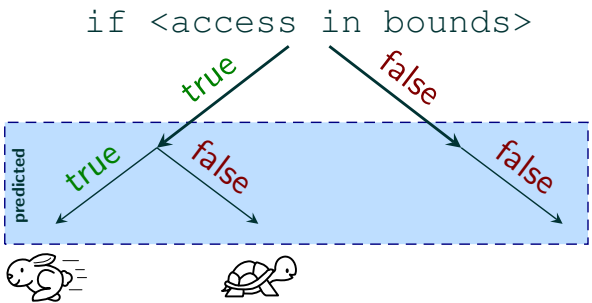

if <access in bounds>
true
predicted
true

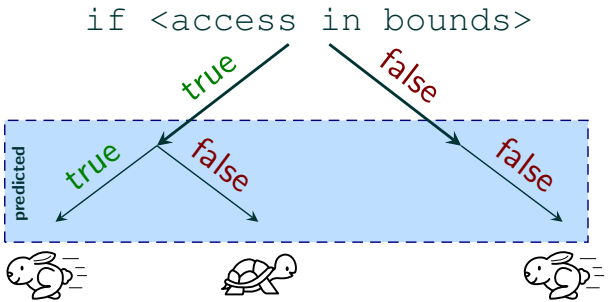

if <access in bounds>
true
predicted
true

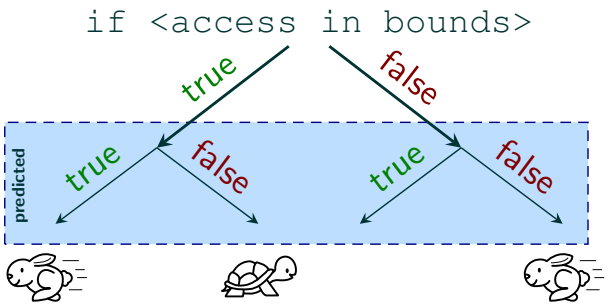

if <access in bounds>
true
predicted
true
false

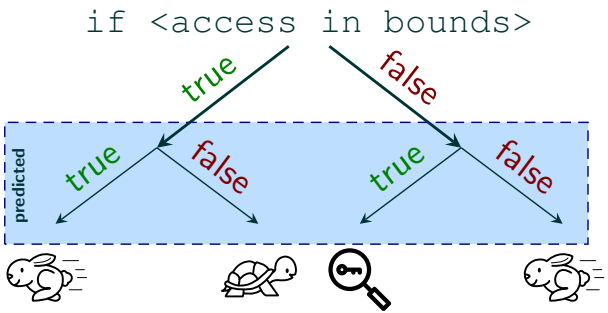

if <access in bounds>
true
predicted
true
false


if <access in bounds>
true
false
predicted
true
false


if <access in bounds>
true
false
predicted
true
false
false


if <access in bounds>
true
false
predicted
true
false
false


if <access in bounds>
true
false
predicted
true
false
true
false


if <access in bounds>
true
false
predicted
true
false
true
false

```
index = 0;

char* data = "textKEY";

if (index < 4)
```


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

0

```
index = 0;
```

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

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

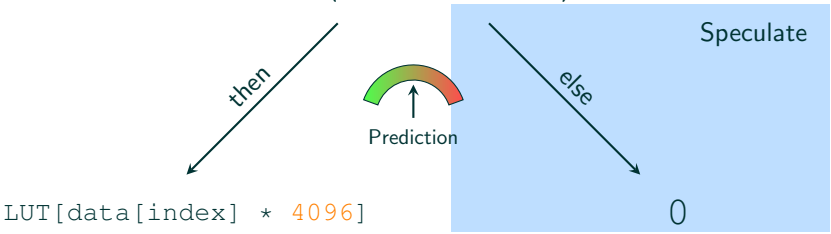

```
LUT[data[index] * 4096]
```

0

```
index = 0;
```

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

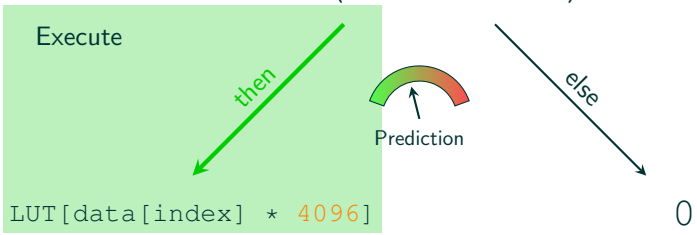
```
if (index < 4)
```



```
index = 0;
```

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

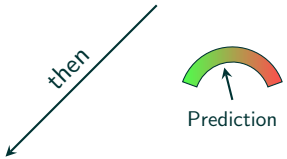
```
if (index < 4)
```



```
index = 1;
```

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

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



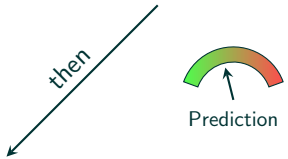

```
LUT[data[index] * 4096]
```

0

```
index = 1;
```

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

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




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

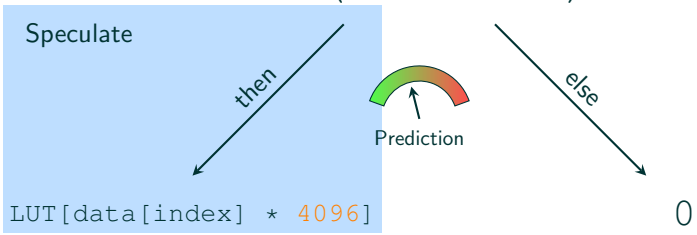
0

Spectre (Variant 1: Bounds-check bypass)

```
index = 1;
```

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

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



```
index = 1;
```

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

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


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

0

```
index = 2;

char* data = "textKEY";

if (index < 4)
```


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

0

```
index = 2;
```

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

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

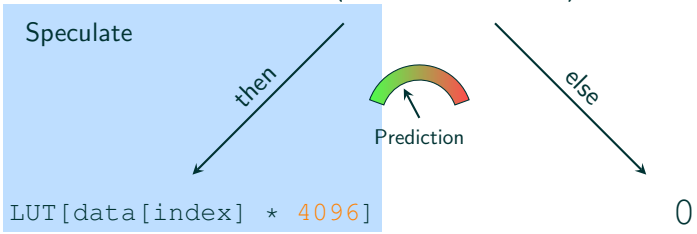

```
LUT[data[index] * 4096]
```

0

```
index = 2;
```

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

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



Spectre (Variant 1: Bounds-check bypass)

```
index = 2;
```

```
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)
```


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

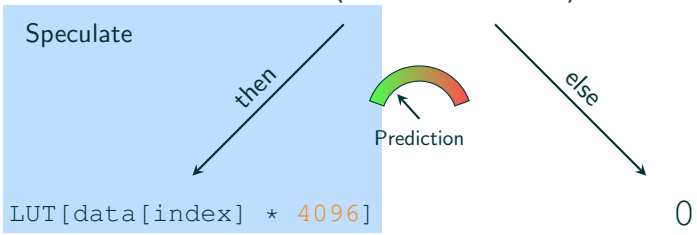
0

Spectre (Variant 1: Bounds-check bypass)

```
index = 3;
```

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

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



```
index = 3;
```

```
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)
```

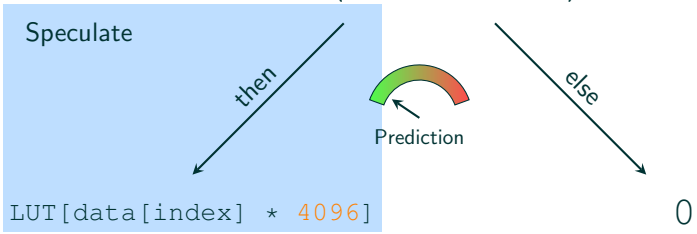

```
LUT[data[index] * 4096]
```

0

```
index = 4;
```

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

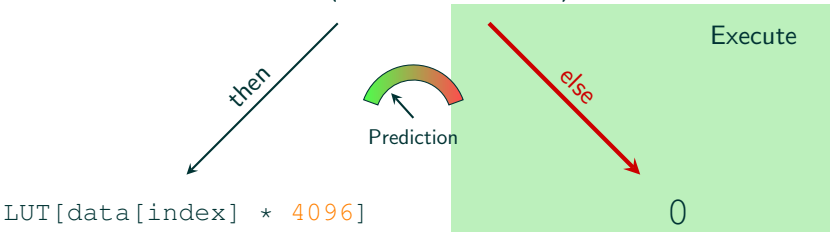
```
if (index < 4)
```



```
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)
```

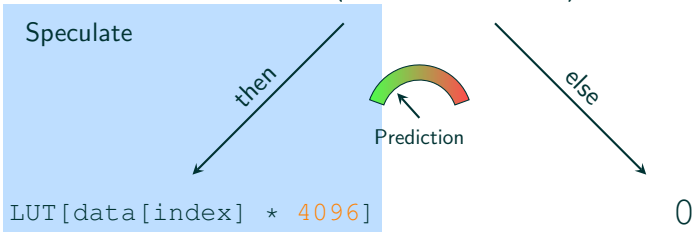

```
LUT[data[index] * 4096]
```

0

```
index = 5;
```

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

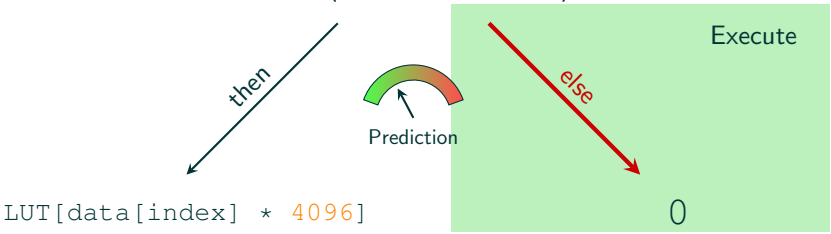
```
if (index < 4)
```



```
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)
```

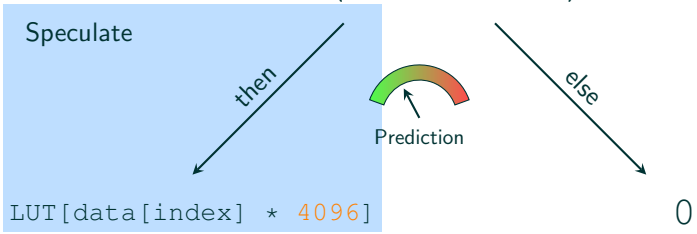

```
LUT[data[index] * 4096]
```

0

```
index = 6;
```

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

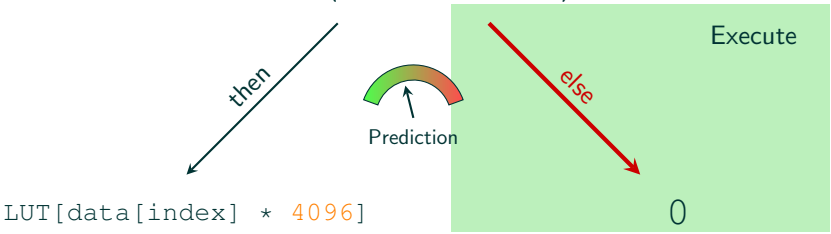
```
if (index < 4)
```



```
index = 6;
```

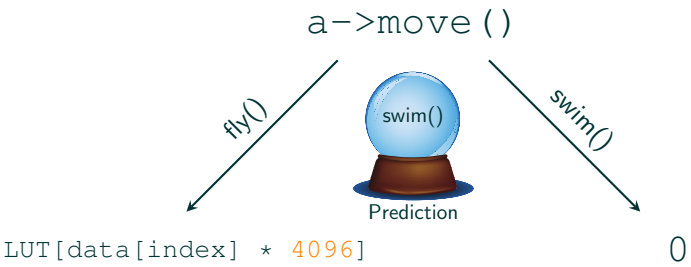
```
char* data = "textKEY";
```

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

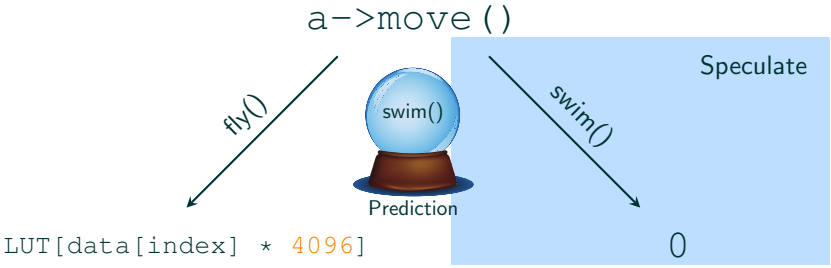


Demo

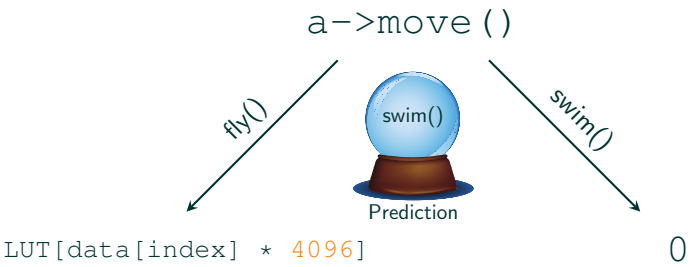
```
Animal* a = bird;
```



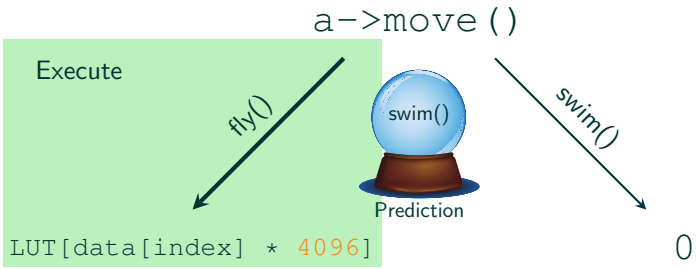
```
Animal* a = bird;
```



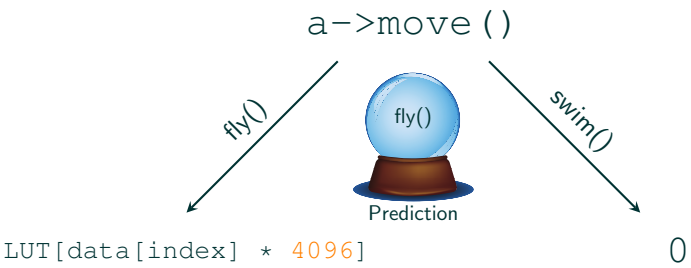
```
Animal* a = bird;
```



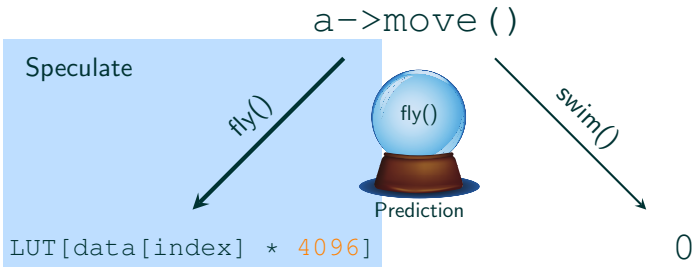
```
Animal* a = bird;
```



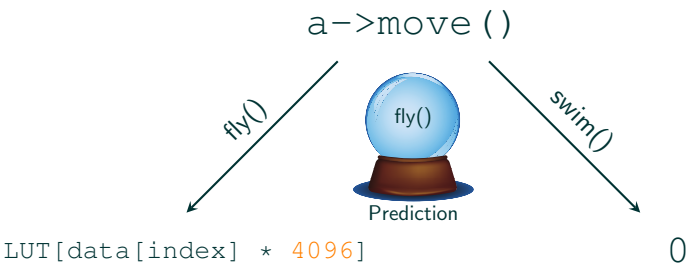
```
Animal* a = bird;
```



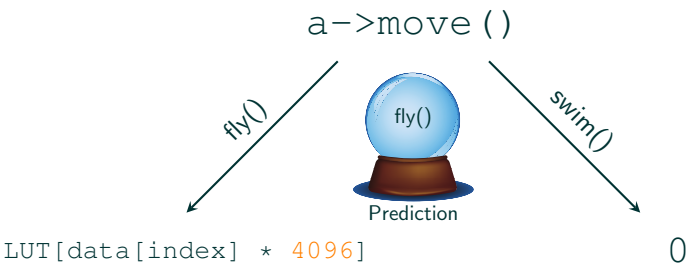
```
Animal* a = bird;
```



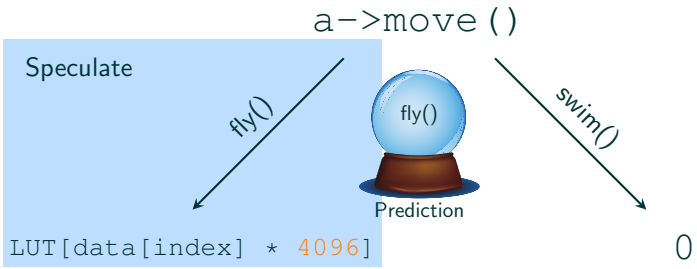
```
Animal* a = bird;
```



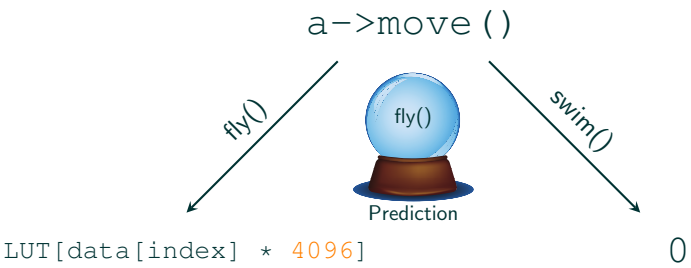
```
Animal* a = fish;
```



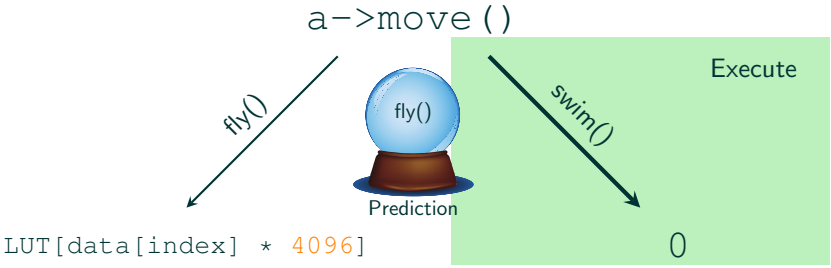
```
Animal* a = fish;
```



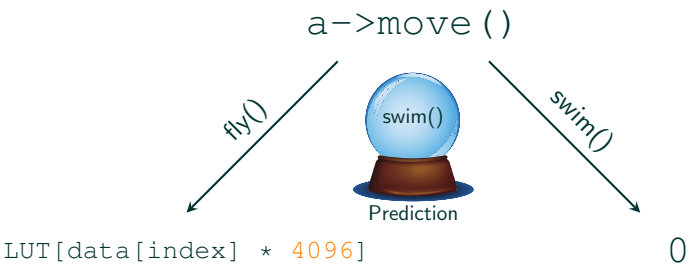
```
Animal* a = fish;
```



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



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



Demo

- We can influence the CPU to mispredict the future

- We can influence the CPU to mispredict the future
- “Convince” other programs to reveal their secrets

- We can influence the CPU to mispredict the future
- “Convince” other programs to reveal their secrets
- Can even be triggered from the browser

Spectre

- We can influence the CPU to mispredict the future
- “Convince” other programs to reveal their secrets
- Can even be triggered from the browser
- Untrusted code can convince trusted code to reveal secrets

- Demonstrated on Intel, AMD and ARM CPUs

- Demonstrated on Intel, AMD and ARM CPUs
- Affects processor with branch target speculation

- Demonstrated on Intel, AMD and ARM CPUs
- Affects processor with branch target speculation
- Much harder to fix, KAISER does not help

Can we fix that?

- Trivial approach: disable speculative execution

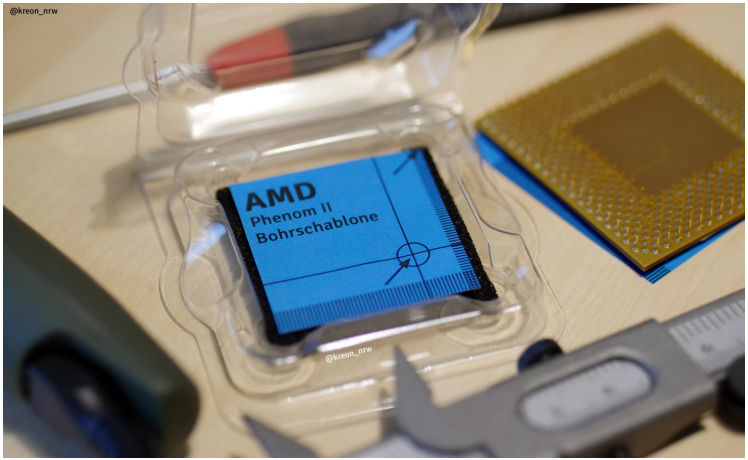
- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation

- 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

Drilling template



Drilling template (@kreon_nrw)

More things which do not work

- Prevent access to high-resolution timer

More things which do not work

- Prevent access to high-resolution timer

→ Own timer using timing thread (last year)

More things which do not work

- Prevent access to high-resolution timer

→ Own timer using timing thread (last year)

- Flush instruction only privileged

More things which do not work

- Prevent access to high-resolution timer

→ Own timer using timing thread (last year)

- Flush instruction only privileged

→ Cache eviction through memory accesses (last year)

More things which do not work

- Prevent access to high-resolution timer

→ Own timer using timing thread (last year)

- Flush instruction only privileged

→ Cache eviction through memory accesses (last year)

- Just move secrets into secure world

More things which do not work

- Prevent access to high-resolution timer

→ Own timer using timing thread (last year)

- Flush instruction only privileged

→ Cache eviction through memory accesses (last year)

- Just move secrets into secure world

→ Spectre works on secure enclaves


GHOSTBUSTERS

- Workaround: insert instructions stopping speculation

Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

→ insert after every bounds check

- Workaround: insert instructions stopping speculation

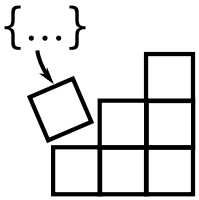
→ insert after every bounds check

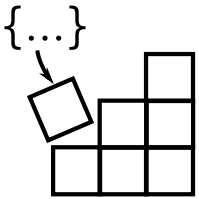
- ARM: Conditional select or conditional move and new barrier `CSDB`

- Workaround: insert instructions stopping speculation

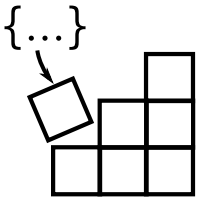
→ insert after every bounds check

- ARM: Conditional select or conditional move and new barrier `CSDB`
- Alternative: `DSB SYS` + `ISB` → greater performance hit
- Retrofitted to existing ARMv7 and ARMv8

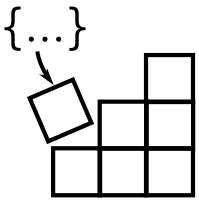




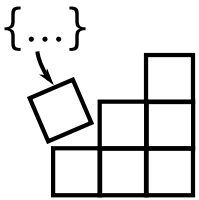
- Speculation barrier requires compiler supported



- 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) → not really reliable
- Explicit use by programmer: `__builtin_load_no_speculate`

Spectre Variant 1 Mitigations

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
  int tmp;

  if (n < N) {
    tmp = array[n]
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

Spectre Variant 1 Mitigations


```
// Unprotected

int array[N];

int get_value(unsigned int n) {
  int tmp;

  if (n < N) {
    tmp = array[n]
  } else {
    tmp = FAIL;
  }

  return tmp;
}
```

```
// Protected

int array[N];

int get_value(unsigned int n) {

  int *lower = array;
  int *ptr = array + n;
  int *upper = array + N;

  return
    __builtin_load_no_speculate
    (ptr, lower, upper, FAIL);

}
```


GHOSTBUSTERS

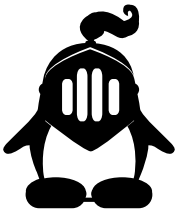
- Speculation barrier works if affected code constructs are known

- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability

- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable

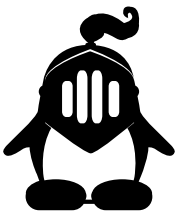
- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable
- Non-negligible performance overhead of barriers

Spectre Variant 2 Mitigations (Software)



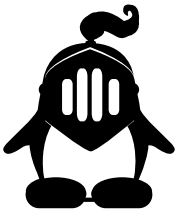
- ARM provides hardened Linux kernel and ARM Trusted Firmware patches

Spectre Variant 2 Mitigations (Software)



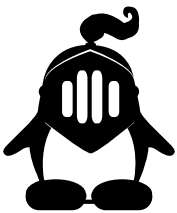
- ARM provides hardened Linux kernel and ARM Trusted Firmware patches
- Clears branch-predictor state on context switch

Spectre Variant 2 Mitigations (Software)



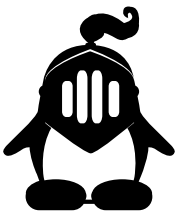
- ARM provides hardened Linux kernel and ARM Trusted Firmware patches
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...

Spectre Variant 2 Mitigations (Software)



- ARM provides hardened Linux kernel and ARM Trusted Firmware patches
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...
- ...or workaround (disable/enable MMU)

Spectre Variant 2 Mitigations (Software)



- ARM provides hardened Linux kernel and ARM Trusted Firmware patches
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...
- ...or workaround (disable/enable MMU)
- Google's Retpoline does not work on ARM

Learn from it

We have ignored software side-channels for many many years:

Learn from it

We have ignored software side-channels for many many years:

- attacks on crypto

Learn from it

We have ignored software side-channels for many many years:

- attacks on crypto → "software should be fixed"

Learn from it

We have ignored software side-channels for many many years:

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

Learn from it

We have ignored software side-channels for many many years:

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

Learn from it

We have ignored software side-channels for many many years:

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

Learn from it

We have ignored software side-channels 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"

Learn from it

We have ignored software side-channels 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"

→ 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:

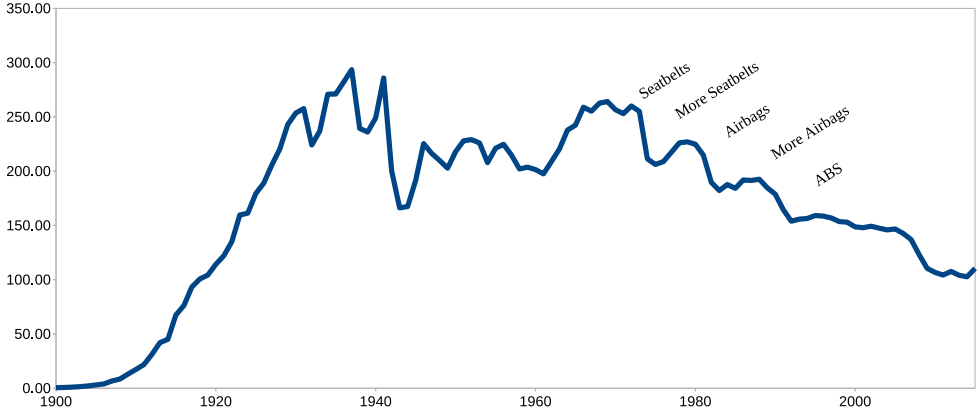
- the side channels were documented in the processor manual

When you read the manuals...

After learning about a side channel you realize:

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

What do we learn from it?



Motor Vehicle Deaths in U.S. by Year

A unique chance

A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
- find good trade-offs between security and performance

Conclusion

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

Meltdown & Spectre

Side-channels considered h**ARM**ful

Qualcomm Mobile Security Summit 2018

17 May, 2018 - San Diego, CA

Moritz Lipp (@mlqxyz) Michael Schwarz (@misc0110)