

Embedded Security

Chapter 3: Meltdown and Spectre

Michael Schwarz, Moritz Lipp

20.03.2018

www.iaik.tugraz.at



- Meltdown and Spectre are two CPU vulnerabilities
- Discovered in 2017 by 4 independent teams
- Due to an embargo, released at the beginning of 2018
- News coverage followed by a lot of panic



FOX
BUSINESS
WASHINGTON, D.C.

WASHINGTON, D.C.

**NEWS
ALERT**

**INTEL REVEALS DESIGN FLAW THAT
COULD ALLOW HACKERS TO ACCESS DATA**

WINTER STORM



FOX
BUSINESS
NETWORK



@FOXBUSINESS



DEVELOPING STORY

LIVE

COMPUTER CHIP FLAWS IMPACT BILLIONS OF DEVICES



DAX ▲ 164.69

NEWS STREAM



GLOBAL

COMPUTER CHIP SCARE

The bugs are known as 'Spectre' and 'Meltdown'

BBC WORLD NEWS

£:HK\$ 10.58

EURO:£ 0.891



SECURITY FLAW REVEALED

Intel (Prev)		
45.26	-1.59	[-3.39%]

Intel (After Hours)		
44.85	-0.41	[-0.91%]

CAPITAL
CONNECTION

SHROUT: ISSUE NOT UNIQUE TO
INTEL, BUT IT'S AFFECTED THE MOST

 **CNBC**



A lot of confusion fueled the panic

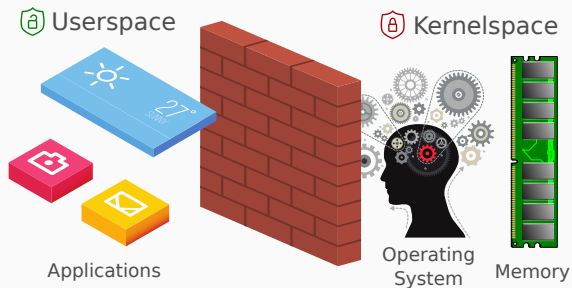
- Which CPUs/vendors are affected?
- Are smartphones/IoT devices affected?
- Can the vulnerabilities be exploited remotely?
- What data is at risk?
- How hard is it to exploit the vulnerabilities?
- Is it already exploited?

Let's try to clarify these questions

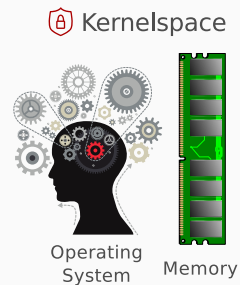
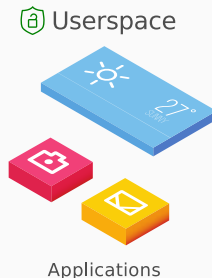


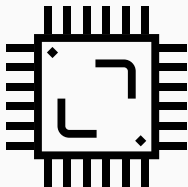
MELTDOWN

- 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 → **syscalls**

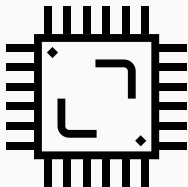


- Breaks isolation between applications and kernel
 - User applications can access kernel addresses
 - Entire physical memory is mapped in the kernel
- Meltdown can read whole DRAM

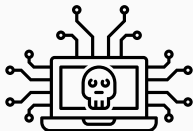




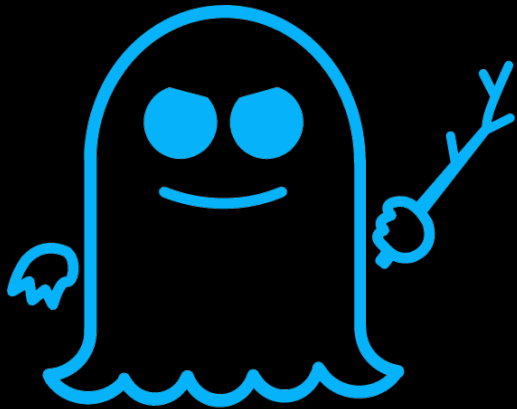
- Only on **Intel** CPUs and some unreleased **ARMs** (Cortex A75)
- AMD and other ARMs seem to be unaffected
- Common cause: permission check done in parallel to load instruction
- Race condition between permission check and dependent operation(s)



- Meltdown variant: read privileged registers
- Limited to some registers, no memory content
- Reported by ARM
- Affects some ARMs (Cortex A15, A57, and A72)



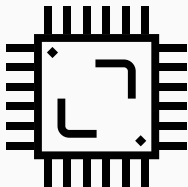
- Meltdown requires code execution on the device (e.g. Apps)
- Untrusted code can read entire memory of device
- Cannot be triggered remotely
- Proof-of-concept code available online
- No info about environment required → easy to reproduce



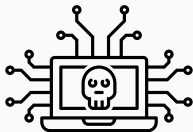
SPECTRE

- Mistrains branch prediction
 - CPU speculatively executes code which should not be executed
 - Can also mistrain indirect calls
- Spectre “convinces” program to execute code





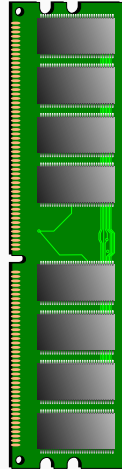
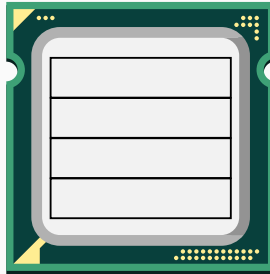
- On Intel and AMD CPUs
- Some ARMv8 (Cortex R and Cortex A) are also affected
- Common cause: speculative execution of branches
- Speculative execution leaves microarchitectural traces which leak secret



- Spectre requires code execution on the device (e.g. Apps)
- Untrusted code can convince trusted code to reveal secrets
- Can be triggered remotely (e.g. in the browser)
- Proof-of-concept code available online
- Info about environment required → hard to reproduce

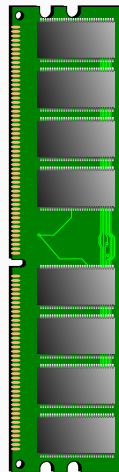
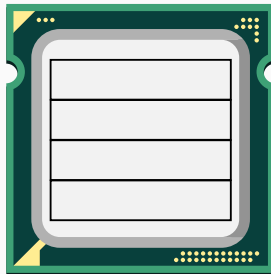
Background

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



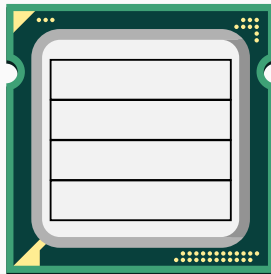
```
printf("%d", i);  
printf("%d", i);
```

Cache miss

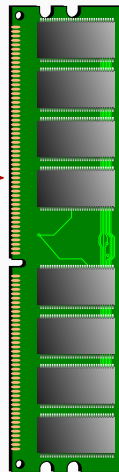


```
printf("%d", i);  
printf("%d", i);
```

Cache miss

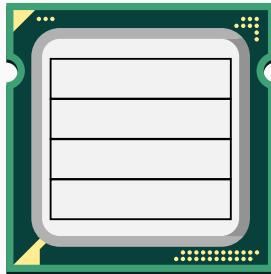


Request



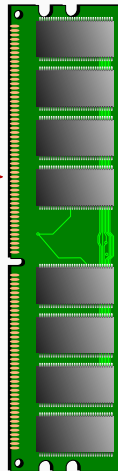
```
printf("%d", i);  
printf("%d", i);
```

Cache miss



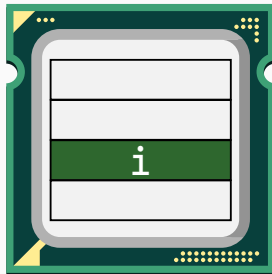
Request

Response



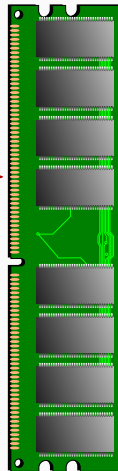
```
printf("%d", i);  
printf("%d", i);
```

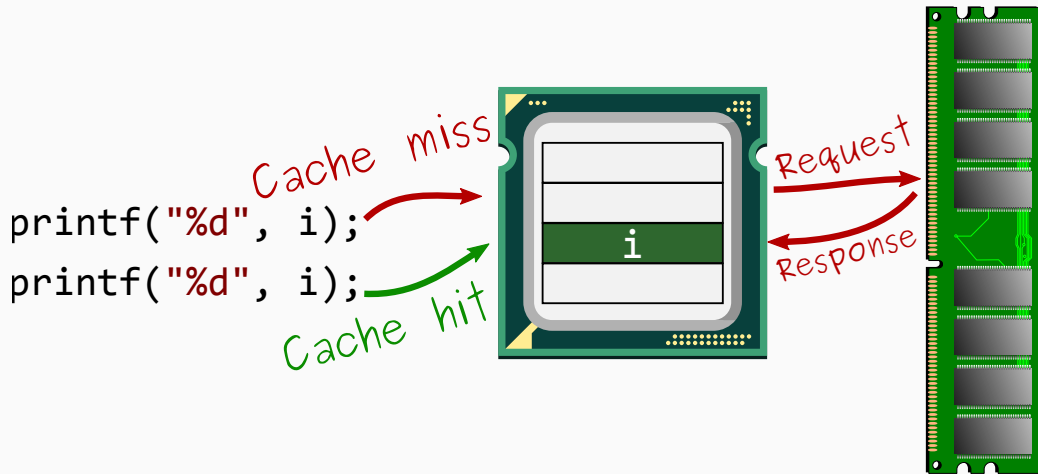
Cache miss

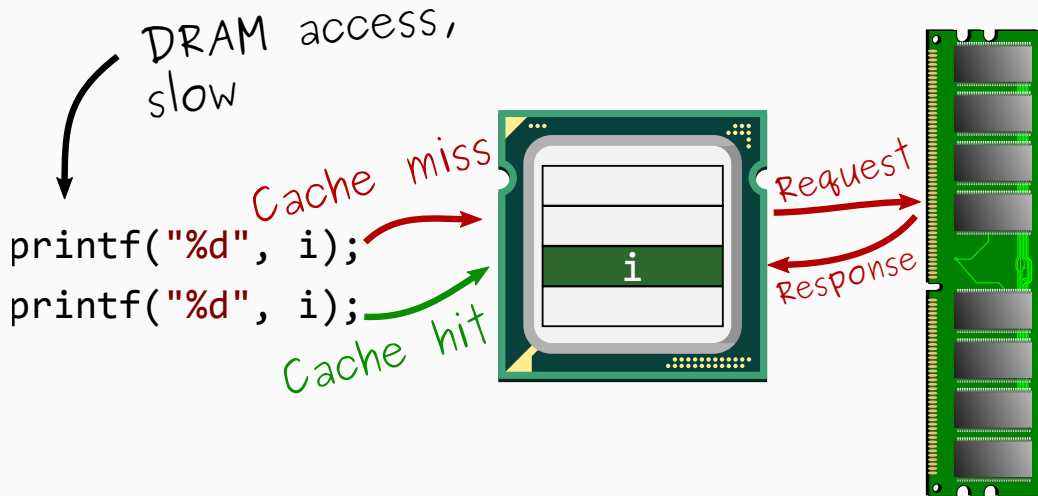


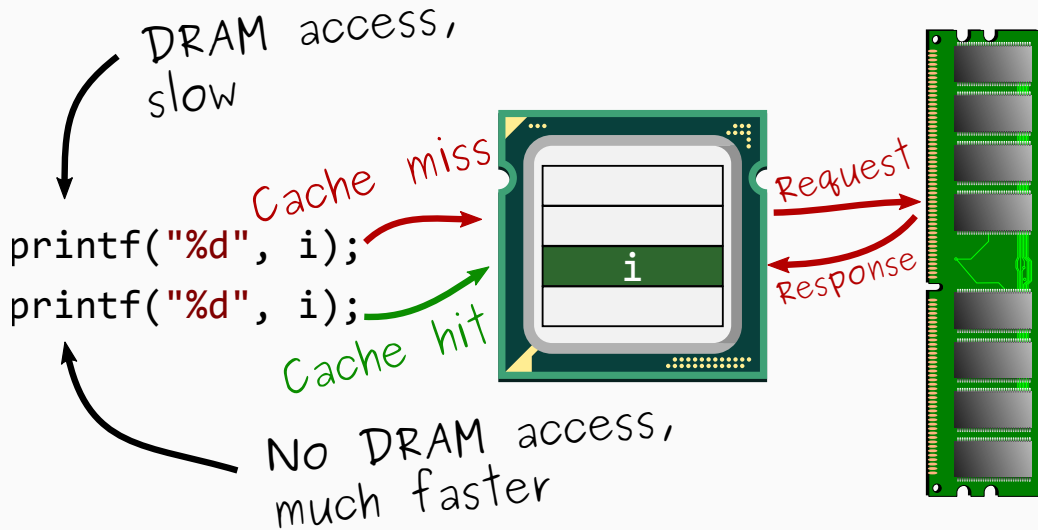
Request

Response





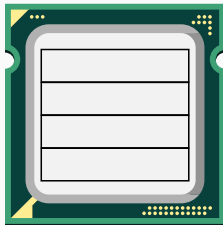




Shared Memory

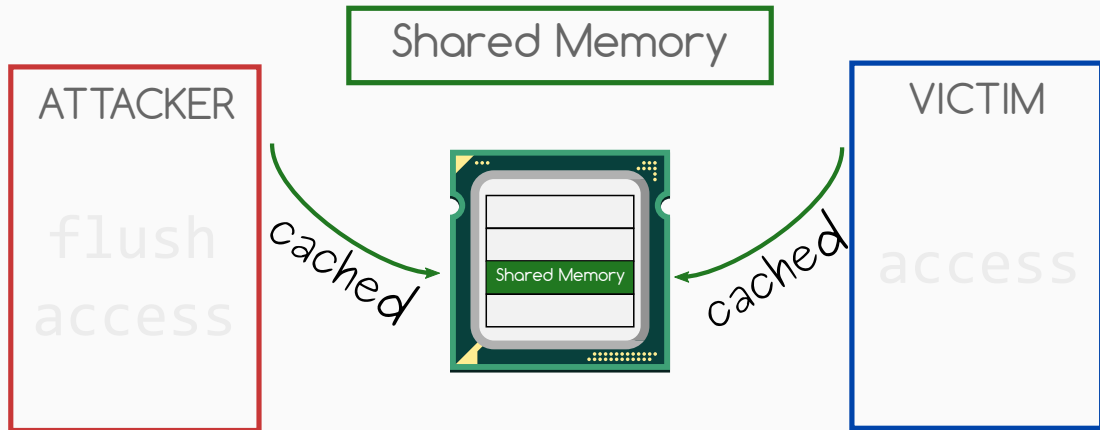
ATTACKER

flush
access



VICTIM

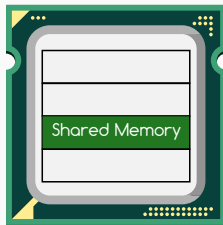
access



Shared Memory

ATTACKER

flush
access



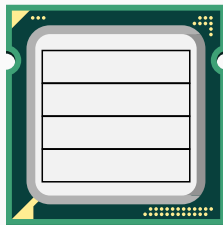
VICTIM

access

Shared Memory

ATTACKER

flush
access



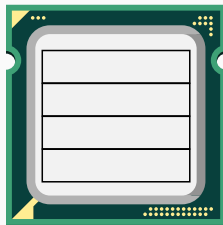
VICTIM

access

Shared Memory

ATTACKER

flush
access



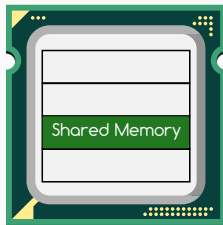
VICTIM

access

Shared Memory

ATTACKER

flush
access



VICTIM

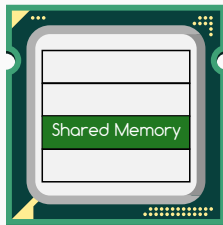
access



Shared Memory

ATTACKER

flush
access



VICTIM

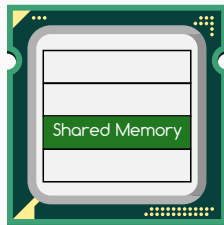
access

Shared Memory

ATTACKER

flush

access



VICTIM

access

fast if victim accessed data,
slow otherwise



Out-of-order Execution

6. Cook everything until
vegetables are soft

6. Add spices to taste
and stir for 10 minutes

*7. Serve with cooked
and peeled potatoes*





Wait for an hour





Wait for an hour

LATENCY

1. *Wash and cut
vegetables*

2. *Pick the basil leaves
and set aside*

3. *Heat 2 tablespoons of
oil in a pan*

4. *Fry vegetables until
golden and softened*



Dependency

1. Wash and cut vegetables

2. Pick the basil leaves and set aside

3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened

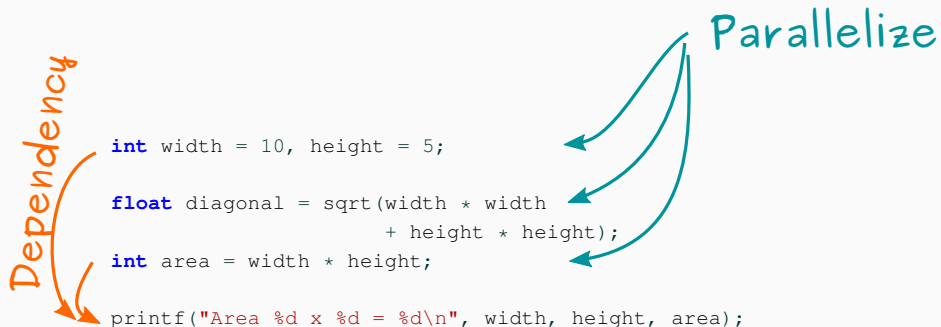
Parallelize

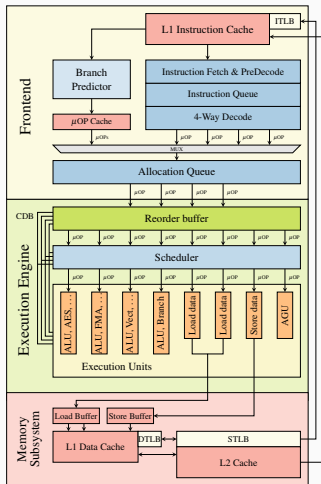


```
int width = 10, height = 5;

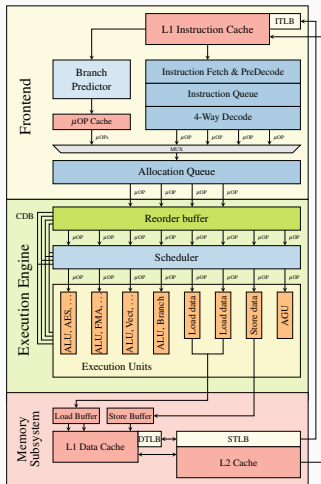
float diagonal = sqrt(width * width
                      + height * height);
int area = width * height;

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





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



- 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

We are ready for the gory details of Meltdown



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

```
# sudo grep linux_banner /proc/kallsyms  
fffffffff81a000e0 R linux_banner
```



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



- Compile and run

```
segfault at ffffffff81a000e0 ip 000000000400535  
sp 00007ffce4a80610 error 5 in reader
```



- Compile and run

```
segfault at ffffffff81a000e0 ip 000000000400535  
sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are of course **not accessible**



- Compile and run

```
segfault at ffffffff81a000e0 ip 000000000400535  
sp 00007ffce4a80610 error 5 in reader
```

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



- 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
- Maybe it is not that straight forward
- Privilege checks seem to work
- Are privilege checks also done when executing instructions out of order?
- Problem: out-of-order instructions are not visible

- Adapted code

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





- Adapted code

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

- volatile because compiler was not happy

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



- Adapted code

```
*(volatile char*) 0;  
array[0] = 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
- Exception was only thrown afterwards



- 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



- Combine the two things

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



- Combine the two things

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

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



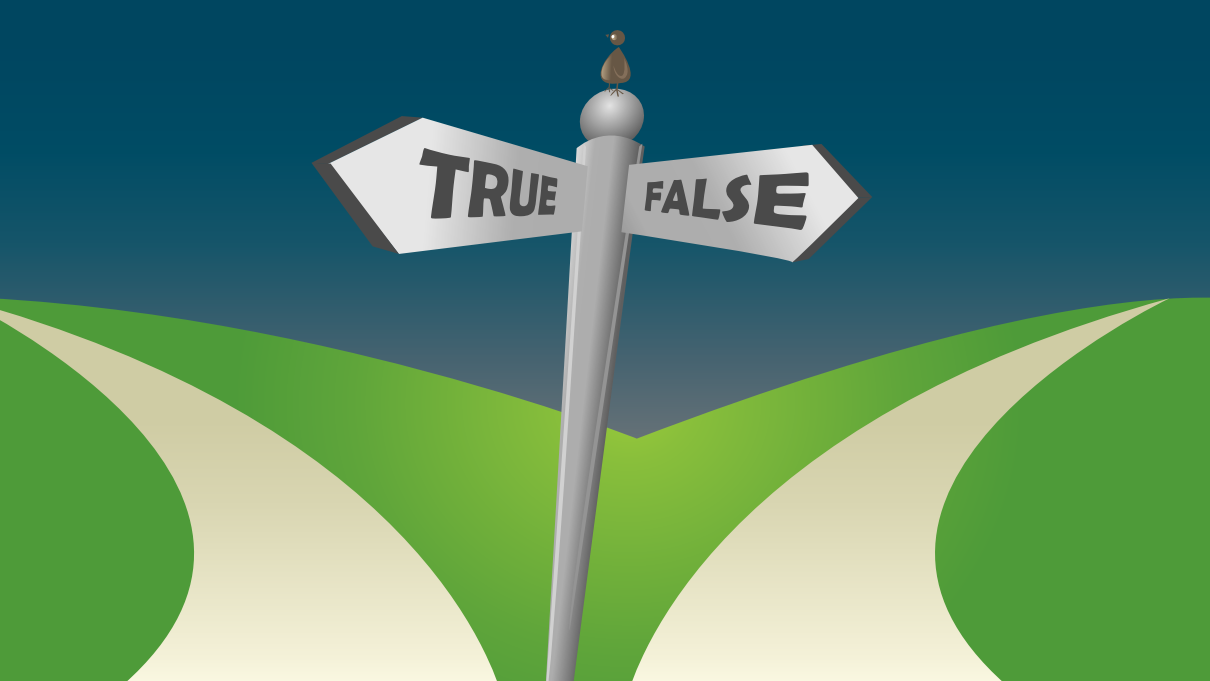
- Flush+Reload over all pages of the array



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



- 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

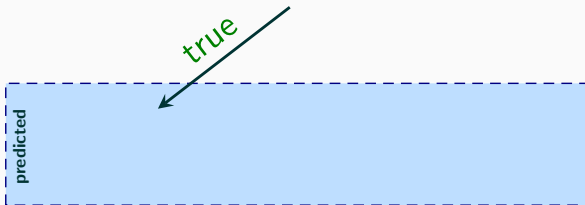


```
if <access in bounds>
```

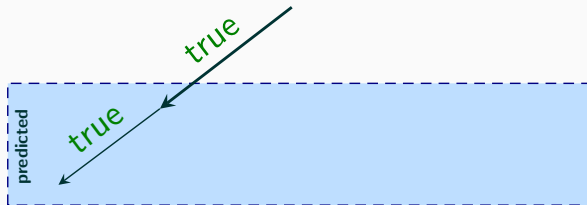


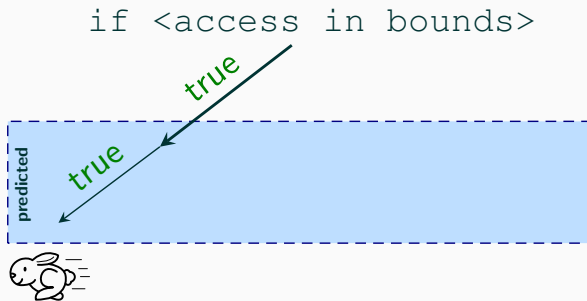
```
if <access in bounds>
```

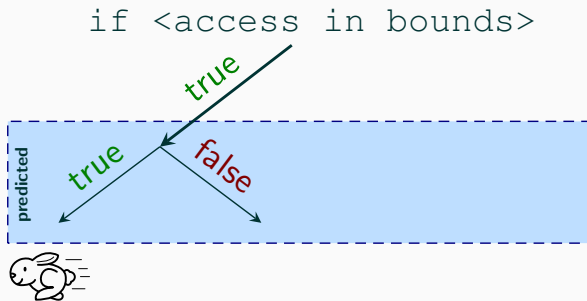
true

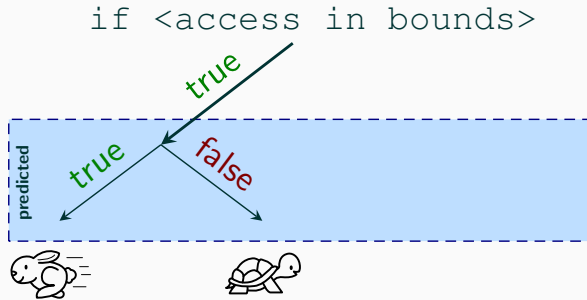


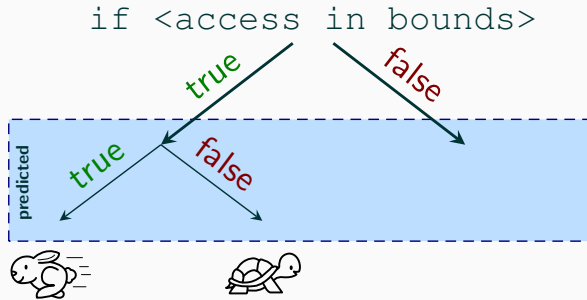
```
if <access in bounds>
```

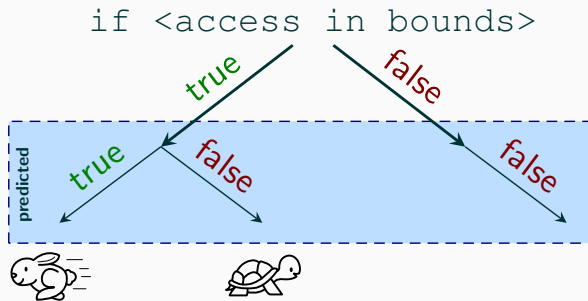


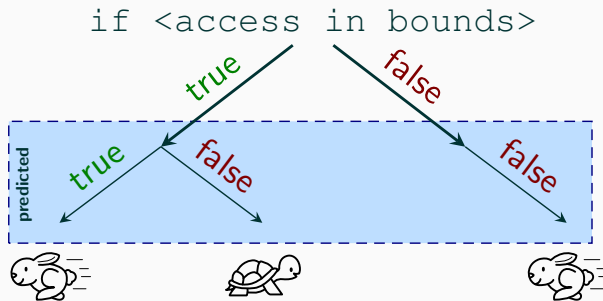


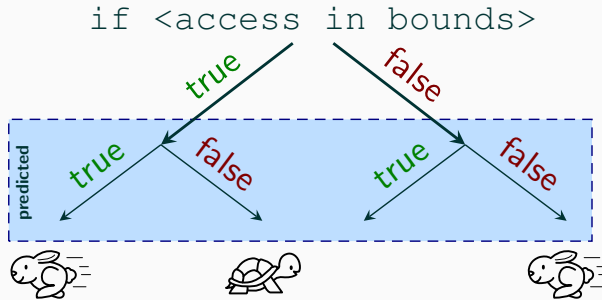


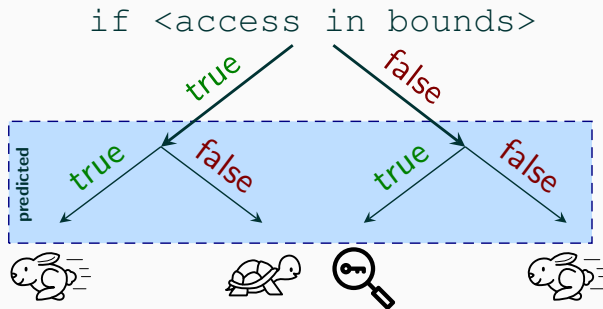












We are ready for the gory details of Spectre

```
index = 0;
```

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

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

then



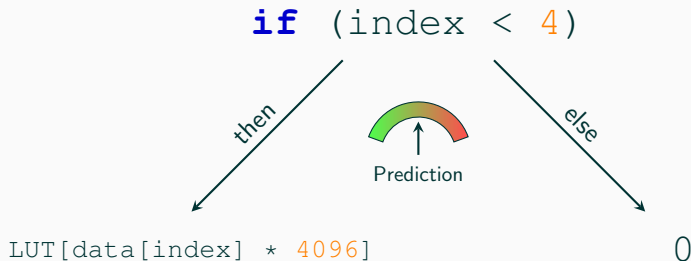
Prediction

else

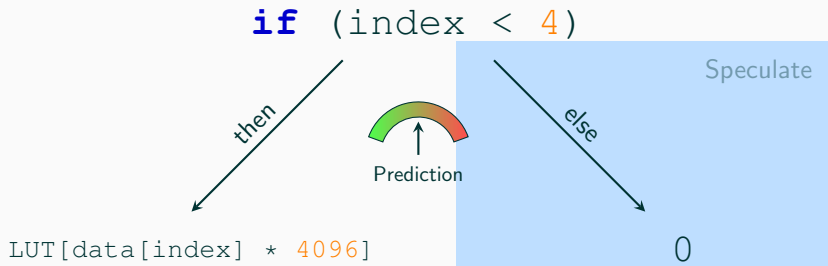
```
LUT[data[index] * 4096]
```

```
0
```

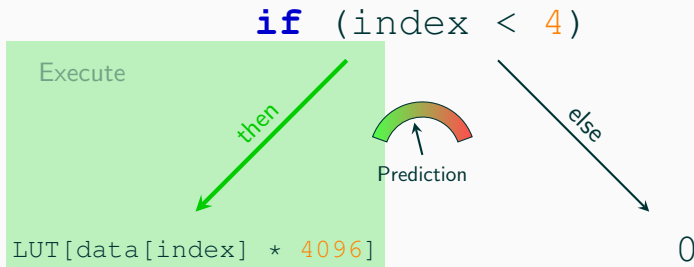
```
index = 0;  
  
char* data = "textKEY";
```



```
index = 0;  
  
char* data = "textKEY";
```



```
index = 0;  
  
char* data = "textKEY";
```



```
index = 1;
```

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

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

then



Prediction

else

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

```
0
```

```
index = 1;
```

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

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

then



Prediction

else

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

```
0
```

```
index = 1;
```

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

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

Speculate

then

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



Prediction

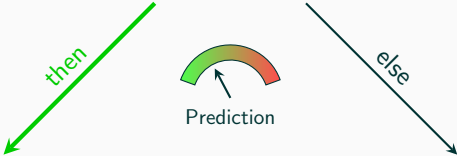
else

0

```
index = 1;  
  
char* data = "textKEY";
```



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



$\text{LUT}[\text{data}[\text{index}] * 4096]$ 0

```
index = 2;
```

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

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

then

else



Prediction

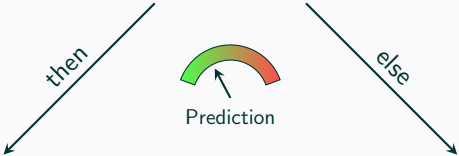
```
LUT[data[index] * 4096]
```

```
0
```

```
index = 2;  
  
char* data = "textKEY";
```



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



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

```
0
```

```
index = 2;
```

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

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

Speculate

then

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



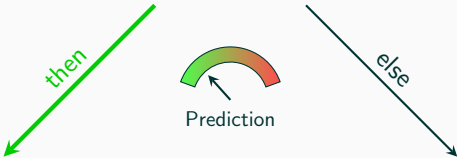
else

0

```
index = 2;  
  
char* data = "textKEY";
```



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



then

else

Prediction

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

```
index = 3;
```

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

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

then

else



Prediction

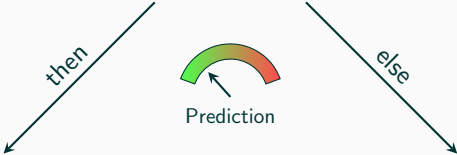
```
LUT[data[index] * 4096]
```

```
0
```

```
index = 3;  
  
char* data = "textKEY";
```



```
if (index < 4)  
    LUT[data[index] * 4096]  
else  
    0
```



```
index = 3;
```

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

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

Speculate

then

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



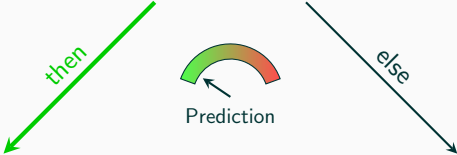
else

0

```
index = 3;  
  
char* data = "textKEY";
```



```
if (index < 4)  
    LUT[data[index] * 4096]  
else  
    0
```



```
index = 4;
```

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

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

then

else



Prediction

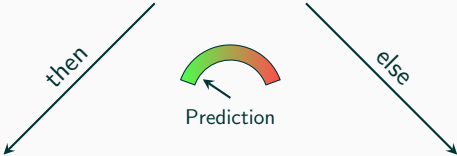
```
LUT[data[index] * 4096]
```

```
0
```

```
index = 4;  
  
char* data = "textKEY";
```

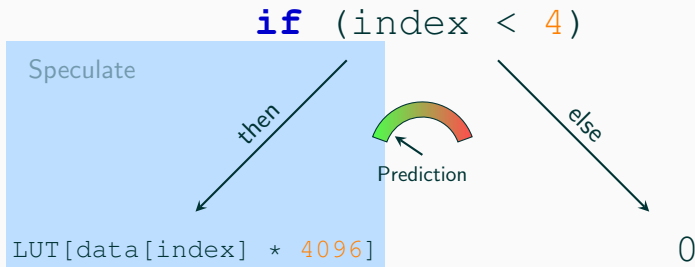


```
if (index < 4)  
    LUT[data[index] * 4096]  
else  
    0
```



```
index = 4;
```

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



```
index = 4;
```

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

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



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



```
index = 5;
```

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

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

then

else



Prediction

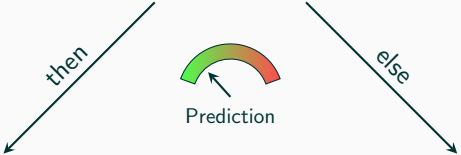
```
LUT[data[index] * 4096]
```

```
0
```

```
index = 5;  
  
char* data = "textKEY";
```

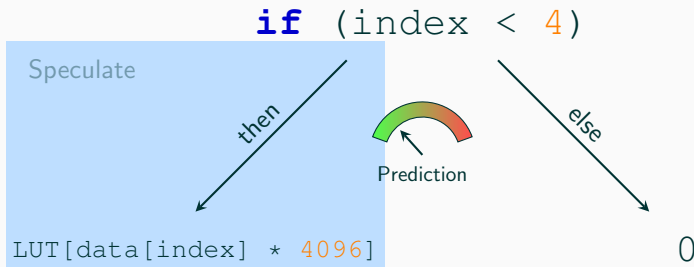


```
if (index < 4)  
    LUT[data[index] * 4096]  
else  
    0
```



```
index = 5;
```

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



```
index = 5;
```

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

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



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



```
0
```

```
index = 6;
```

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

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

then

else



Prediction

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

```
0
```

```
index = 6;
```

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

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

then



Prediction

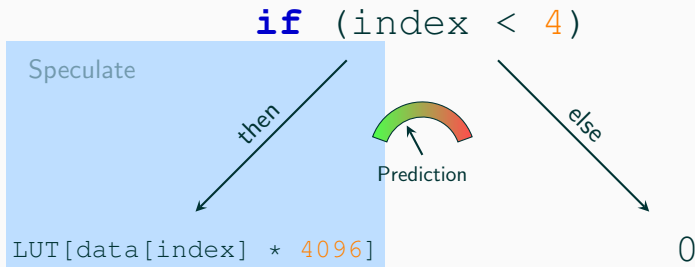
else

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

```
0
```

```
index = 6;
```

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



```
index = 6;
```

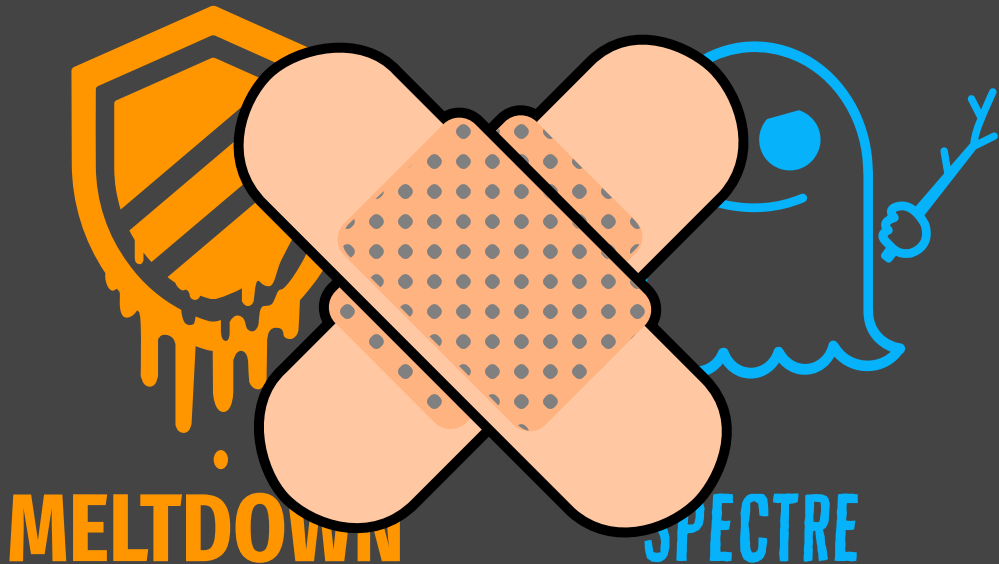
```
char* data = "textKEY";
```

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



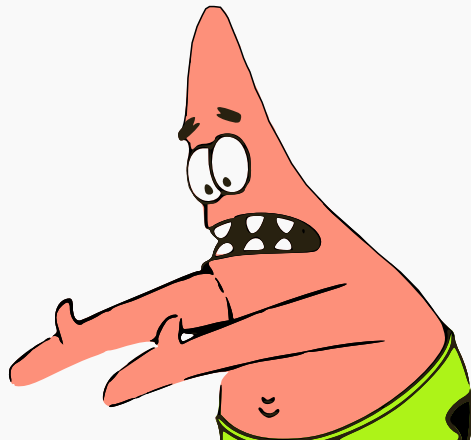
```
LUT[data[index] * 4096]
```





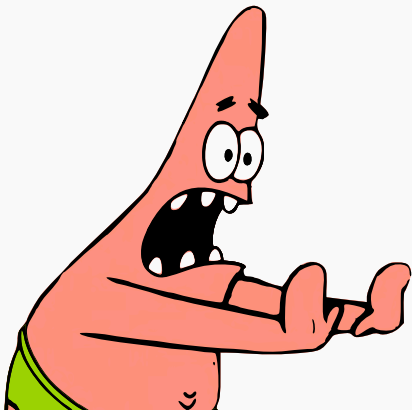
- Kernel addresses in user space are a problem

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

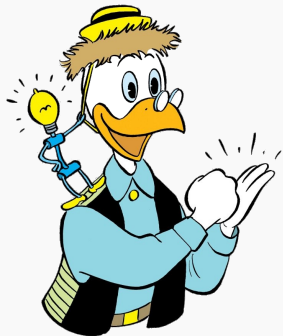




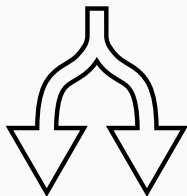
- ...and remove them if not needed?



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



- 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



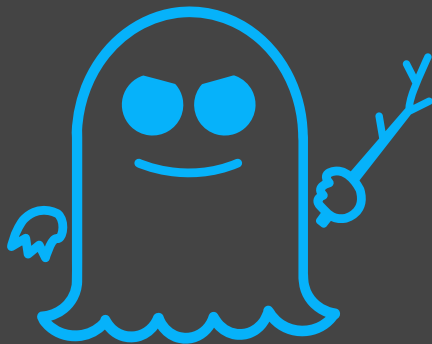
- We published **KAISER** in July 2017
- Intel and others improved and merged it into Linux as **KPTI** (Kernel Page Table Isolation)
- Microsoft implemented similar concept in Windows 10
- Apple implemented it in macOS 10.13.2 and called it “**Double Map**”
- All share the same idea: switching address spaces on context switch



- Depends on how often you need to switch between kernel and user space
- Can be slow, 40% or more on old hardware
- But modern CPUs have additional features
- $\Rightarrow$ Performance overhead on average below 2%



MELTDOWN



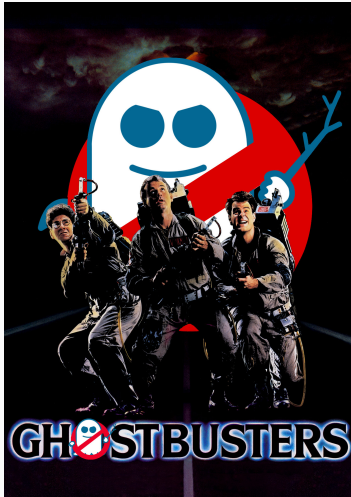
SPECTRE



- Does not directly access kernel
- “Convinces” other programs to reveal their secrets
- Much harder to fix, KAISER does not help
- Ongoing effort to patch via microcode update and compiler extensions



- 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





- Workaround: insert instructions stopping speculation



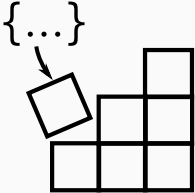
- Workaround: insert instructions stopping speculation
→ insert after every bounds check

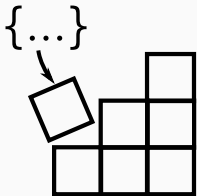


- Workaround: insert instructions stopping speculation
→ insert after every bounds check
- x86: LFENCE, ARM: CSDB

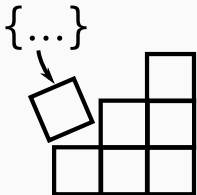


- Workaround: insert instructions stopping speculation
→ insert after every bounds check
- x86: LFENCE, ARM: CSDB
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8

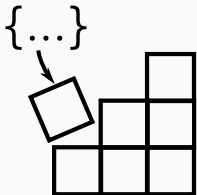




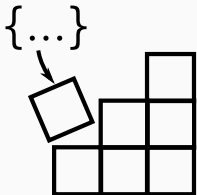
- Speculation barrier requires compiler supported



- 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) → 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`

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
    int tmp;

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

    return tmp;
}
```

```
// Unprotected

int array[N];

int get_value(unsigned int n) {
    int tmp;

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

    return tmp;
}
```

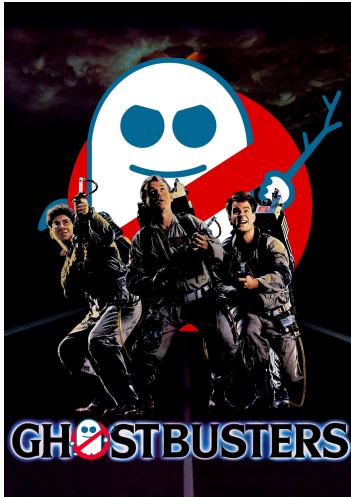
```
// Protected

int array[N];

int get_value(unsigned int n) {

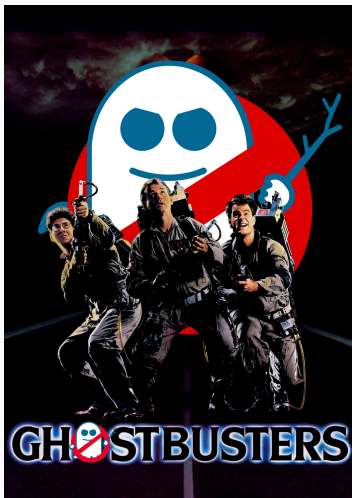
    int *lower = array;
    int *ptr = array + n;
    int *upper = array + N;

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





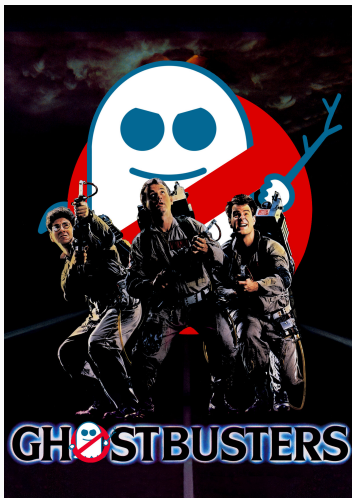
- Speculation barrier works if affected code constructs are known



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



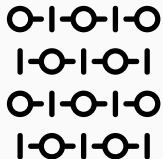
- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable

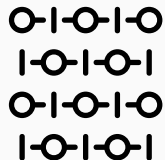


- Speculation barrier works if affected code constructs are known
- Programmer has to fully understand vulnerability
- Automatic detection is not reliable
- Non-negligible performance overhead of barriers

Intel released microcode updates

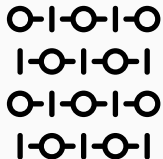
- **Indirect Branch Restricted Speculation (IBRS):**





Intel released microcode updates

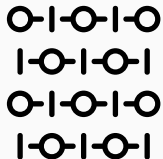
- **Indirect Branch Restricted Speculation (IBRS):**
 - Do not speculate based on anything before entering IBRS mode



Intel released microcode updates

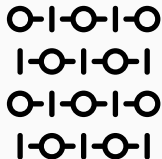
- **Indirect Branch Restricted Speculation (IBRS):**

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



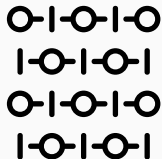
Intel released microcode updates

- **Indirect Branch Restricted Speculation (IBRS):**

- Do not speculate based on anything before entering IBRS mode
→ lesser privileged code cannot influence predictions

- **Indirect Branch Predictor Barrier (IBPB):**

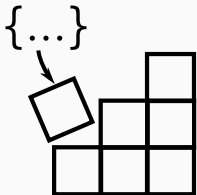
- Flush branch-target buffer



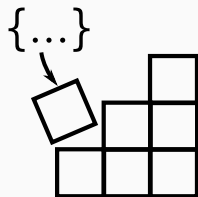
Intel released microcode updates

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



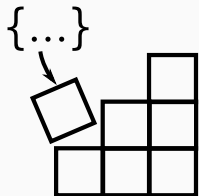
Retpoline (compiler extension)



```
push <call_target>
call 1f
2:                                ; speculation will continue here
    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)

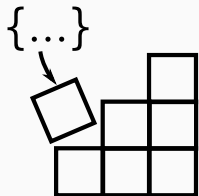


```
push <call_target>
call 1f
2:                                ; speculation will continue here
    lfence                       ; speculation barrier
    jmp 2b                       ; endless loop
1:
    lea 8(%rsp), %rsp           ; restore stack pointer
    ret                        ; the actual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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function

Retpoline (compiler extension)

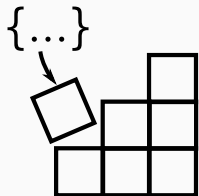


```
push <call_target>
call 1f
2:                                ; speculation will continue here
lfence                          ; speculation barrier
jmp 2b                          ; endless loop
1:
lea 8(%rsp), %rsp ; restore stack pointer
ret                    ; the actual call to <call_target>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

→ always predict to enter an endless loop

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

Retpoline (compiler extension)

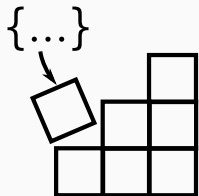


```
push <call_target>
call 1f
2:                                ; speculation will continue here
    lfence                       ; speculation barrier
    jmp 2b                       ; endless loop
1:
    lea 8(%rsp), %rsp            ; restore stack pointer
    ret                         ; the actual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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:

Retpoline (compiler extension)

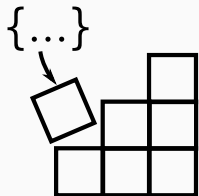


```
push <call_target>
call 1f
2:                                ; speculation will continue here
    lfence                        ; speculation barrier
    jmp 2b                        ; endless loop
1:
    lea 8(%rsp), %rsp             ; restore stack pointer
    ret                          ; the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

→ always predict to enter an endless loop

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

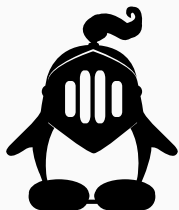
Retpoline (compiler extension)



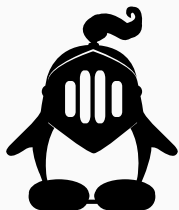
```
push <call_target>
call 1f
2:                                ; speculation will continue here
lfence                           ; speculation barrier
jmp 2b                           ; endless loop
1:
lea 8(%rsp), %rsp ; restore stack pointer
ret                          ; the actual call to <call_target>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

→ always predict to enter an endless loop

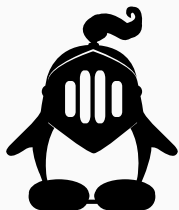
- instead of the correct (or wrong) target function → performance?
 - On Broadwell or newer:
 - **ret** may fall-back to the BTB for prediction
- microcode patches to prevent that



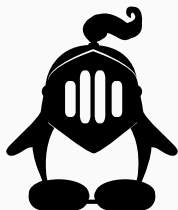
- ARM provides hardened Linux kernel



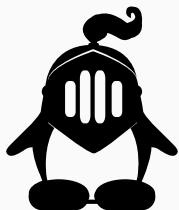
- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch



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



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



- ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
- Either via instruction (`BPIALL`)...
- ...or workaround (disable/enable MMU)
- Non-negligible performance overhead ($\approx 200\text{-}300\text{ ns}$)



- Prevent access to high-resolution timer



- Prevent access to high-resolution timer
- Own timer using timing thread



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged



- Prevent access to high-resolution timer
- Own timer using timing thread
- Flush instruction only privileged
- Cache eviction through memory accesses



- 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



- 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

What to do now?





- Is the used hardware even affected?



- Is the used hardware even affected?
- Can untrusted users run code on affected hardware?



- Is the used hardware even affected?
- Can untrusted users run code on affected hardware?
- Is a software attack even in the threat model?



- Is the used hardware even affected?
- Can untrusted users run code on affected hardware?
- Is a software attack even in the threat model?
- Is confidentiality required on the hardware?



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



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

- attacks on crypto



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

- attacks on crypto → “software should be fixed”



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

- attacks on crypto → “software should be fixed”
- attacks on ASLR



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

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



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



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”



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



After learning about a side channel you realize:



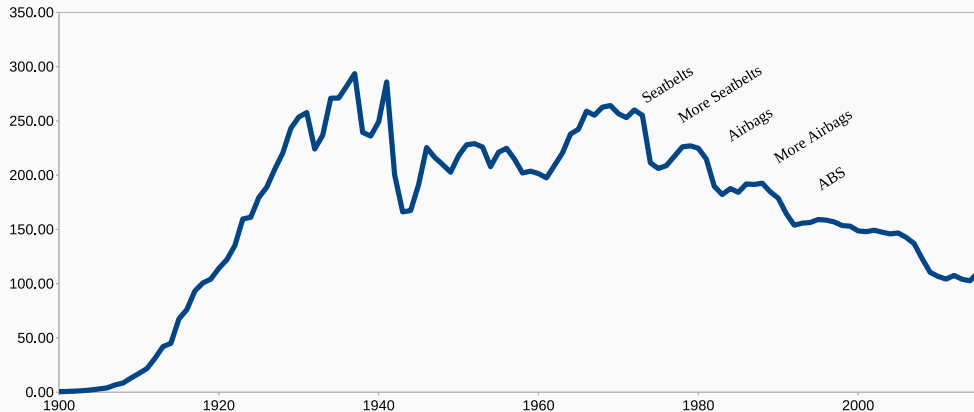
After learning about a side channel you realize:

- the side channels were documented in the Intel manual



After learning about a side channel you realize:

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



Motor Vehicle Deaths in U.S. by Year





A unique chance to

- rethink processor design
 - 705.005 System-On-Chip
- find good trade-offs between security and performance



- 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

Embedded Security

Chapter 3: Meltdown and Spectre

Michael Schwarz, Moritz Lipp

20.03.2018

www.iaik.tugraz.at