Meltdown & Spectre
Side-channels considered hARMful

Qualcomm Mobile Security Summit 2018
17 May, 2018 - San Diego, CA

Moritz Lipp (@mlqxyz)  Michael Schwarz (@misc0110)
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
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
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!
• Find something human readable, e.g., the Linux version

```bash
# sudo grep linux_banner /proc/kallsyms
fffffffff81a000e0 R linux_banner
```
char data = *(char*) 0xffffffffffff81a000e0;
printf("%c\n", data);
• Compile and run

```
segfault at fffffff81a000e0 ip 0000000000400535
sp 00007ffce4a80610 error 5 in reader
```
• Compile and run

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

• Kernel addresses are of course not accessible
• 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
• Just catch the segmentation fault!
• Just catch the segmentation fault!
• We can simply install a signal handler
• Just catch the segmentation fault!
• We can simply install a signal handler
• And if an exception occurs, just jump back and continue
• 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
• 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
Building the Code

- 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 straightforward
• 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);
Cache miss
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss

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

Cache miss

Request
Response

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
printf("%d", i);
printf("%d", i);
CPU Cache

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

Cache miss

Cache hit
CPU Cache

printf("%d", i);

DRAM access, slow

Cache miss

printf("%d", i);

Cache hit

DRAM access,
slow

Request

Response

i

printf("%d", i);
CPU Cache

printf("%d", i);

Cache miss

DRAM access,
slow

Cache hit

No DRAM access,
much faster

printf("%d", i);

Request

Response

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
Memory Access Latency

![Graph showing memory access latency with measured access time on the x-axis and number of accesses on the y-axis. The graph has data points for cache hits and cache misses.]

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
Flush + Reload

ATTACKER

Shared Memory

VICTIM

flush

access

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush

access

Shared Memory

access

VICTIM
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush

access

fast if victim accessed data,
slow otherwise

Shared Memory

VICTIM

access

fast if victim accessed data,
slow otherwise
Operating Systems 101
• 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
```c
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);
```
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
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

\[ *(\text{volatile char}*)0; \]
\[ \text{array}[84 \times 4096] = 0; \]
• Adapted code

\[
\ast (\text{volatile char} \ast) 0;
\text{array}[84 \ast 4096] = 0;
\]

• \texttt{volatile} because compiler was not happy

\textit{warning: statement with no effect \([-\text{Wunused-value}\]}  
\ast (\text{char} \ast) 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
• Out-of-order instructions leave microarchitectural traces
• We can see them for example in the cache
• Out-of-order instructions leave microarchitectural traces
• We can see them for example in the cache
• Give such instructions a name: transient instructions
• 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 committing them
• Maybe there is no permission check in transient instructions...
• ...or it is only done when committing them
• Add another layer of indirection to test

```c
char data = *(char*) 0xfffffffff81a000e0;
array[data * 4096] = 0;
```
• Maybe there is no permission check in transient instructions...
• ...or it is only done when committing them
• Add another layer of indirection to test

```c
char data = *(char*) 0xfffffffff81a000e0;
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?
- 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

- ...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
• Let’s just unmap the kernel in user space
• Kernel addresses are then no longer present
• 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
• We published KAISER in July 2017
• Intel and others improved and merged it into Linux as KPTI (Kernel Page Table 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
• 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?
• ARM found a closely related Meltdown variant
ARM found a closely related Meltdown variant

Read of system registers that are not accessible from current exception level
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
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
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
• Solution? Substitute registers with dummy values on context switch
• Only necessary for virtual addresses and secrets
→ only a few registers affected
Solution? Substitute registers with dummy values on context switch
• Only necessary for virtual addresses and secrets
→ 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>
if <access in bounds>

true

true

predicted
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>
if \(<\text{access in bounds}>\)
if <access in bounds>
if <access in bounds>

predicted

true

false

true

false

true

false
if <access in bounds>

true → true
false → false
false → true
true → false
index = 0;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 0;

char* data = "textKEY";

if (index < 4)
    then
    Prediction
    LUT[data[index] * 4096]
else
    0
```c
index = 0;

char* data = "textKEY";

if (index < 4)
  Speculate
  LUT[data[index] * 4096]
else
  Prediction
  0
```
```c
index = 0;

char* data = "textKEY";

if (index < 4)
{
    LUT[data[index] * 4096]
}
else
{
    0
}
```
index = 1;

char* data = "textKEY";

if (index < 4)
    then
    LUT[data[index] * 4096]
    else
    0
index = 1;
char* data = "textKEY";

if (index < 4)

then

Prediction

LUT[data[index] * 4096]

else

0
index = 1;

char* data = "textKEY";

if (index < 4) then

LUT[data[index] * 4096]

else

0

Speculate

then

Prediction

else
index = 1;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096] 0

else

Prediction

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
index = 2;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]
index = 2;

cchar* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
index = 2;

char* data = "textKEY";

if (index < 4)

Speculate
then
LUT[data[index] * 4096]

Prediction

else
0
index = 2;

char* data = "textKEY";

if (index < 4)

then

Prediction

LUT[data[index] * 4096]

else

0
index = 3;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 3;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0
index = 3;

char* data = "textKEY";

if (index < 4)
    Speculate
else
    Prediction

LUT[data[index] * 4096]
spectre (variant 1: bounds-check bypass)

\[
\text{index} = 3;
\]

\[
\text{char* data} = "\text{te}xt\text{KEY}";
\]

\[
\text{if (index} < 4)\]

\[
\text{LUT[data[index] \times 4096]}\quad 0
\]
index = 4;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

if (index < 4)

then

else

Prediction

LUT[data[index] * 4096] 0
**Spectre (Variant 1: Bounds-check bypass)**

```c
index = 4;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0

```
Spectre (Variant 1: Bounds-check bypass)

```c
index = 4;

char* data = "textKEY";

if (index < 4)
    then

    LUT[data[index] * 4096]

    else
        Prediction
            Execute
            0

else
```
index = 5;

char* data = "textKEY";

if (index < 4)

then

Prediction

else

LUT[data[index] * 4096] 0
index = 5;

char* data = "textKEY";

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

Spectre (Variant 1: Bounds-check bypass)

```c
index = 5;

char* data = "textKEY";

if (index < 4)
{
    Speculate
    then
    LUT[data[index] * 4096]
}
else
{
    Prediction
    0
}
```
Spectre (Variant 1: Bounds-check bypass)

```c
index = 5;

char* data = "textKEY";

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

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
index = 6;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0
index = 6;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
Spectre (Variant 1: Bounds-check bypass)

```c
index = 6;

char* data = "textKEY";

if (index < 4)
    Speculate
    then
    Prediction
    LUT[data[index] * 4096]

else
    0
```
index = 6;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
0
Demo
```cpp
Animal* a = bird;

a->move();
```

![Diagram](image-url)
Animal* a = bird;

a->move();

LUT[data[index] * 4096]
Animal* a = bird;

a->move();

fly()

swim()  swim()

Prediction

LUT[data[index] * 4096]  0
Animal* a = bird;

Execute

a->move()

LUT[data[index] * 4096]

fly()

Prediction

swim()

swim()
```cpp
Animal* a = bird;
```

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

Prediction

```
0
```
Animal* a = bird;

Speculate

LUT[data[index] * 4096]

fly() — Prediction

swim()
Animal* a = bird;

a->move()

fly()

fly()

swim()

LUT[data[index] * 4096] 0
Animal* a = fish;

a->move();

fly();
fly();

swim();

LUT[data[index] * 4096] 0
Spectre (Variant 2: Branch target injection)

```cpp
Animal* a = fish;
a->move();
```

Speculate

LUT[data[index] * 4096]

fly()

Prediction

fly()

swim()

0
Animal* a = fish;

a->move();

fly()

fly()

swim()

LUT[data[index] * 4096] 0
Animal* a = fish;

a->move();

LUT[data[index] * 4096]
Animal* a = fish;

a->move();

fly()

LUT[data[index] * 4096] 0

swim()

swim()

Prediction
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
• 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
Spectre

- Demonstrated on Intel, AMD and ARM CPUs

Moritz Lipp (@mlqxyz), Michael Schwarz (@misc0110) — www.iaik.tugraz.at
Spectre

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

- 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 (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)
• 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
Spectre Variant 1 Mitigations

• 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

Retrofitted to existing ARMv7 and ARMv8
• Workaround: insert instructions stopping speculation

ARM: Conditional select or conditional move and new barrier CSDB

Alternative: DSB SYS + ISB → greater performance hit

Retrofitted to existing ARMv7 and ARMv8
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation
  → insert after every bounds check
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation → insert after every bounds check
- ARM: Conditional select or conditional move and new barrier CSDB
Spectre Variant 1 Mitigations

- 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

Already implemented in GCC, LLVM, and MSVC

Can be automated (MSVC) → not really reliable

Explicit use by programmer: `builtin load no speculate`
• Speculation barrier requires compiler supported
Spectre Variant 1 Mitigations

- 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;
}
Spectre Variant 1 Mitigations

```c
// 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);
}
```
Spectre Variant 1 Mitigations

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

- Clears branch-predictor state on context switch
- Either via instruction (BPIALL)...
• 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)
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
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 $\rightarrow$ “software should be fixed”
- attacks on ASLR $\rightarrow$ “ASLR is broken anyway”
- attacks on SGX and TrustZone $\rightarrow$ “not part of the threat model”

$\rightarrow$ for years we solely optimized for performance
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
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

Seabets
More Seabets
Airbags
More Airbags
ABS
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 hARMful

Qualcomm Mobile Security Summit 2018
17 May, 2018 - San Diego, CA

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