Microarchitectural Attacks:
From the Basics to Arbitrary Read and Write Primitives without any Software Bugs

Daniel Gruss
June 19, 2018

Graz University of Technology
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

ORDER VIA PHONE: +555 12345

$1,299
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
Cache miss
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss
Request
Response

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

Cache miss
Request
Response
i
Cache hit
Request
Response
CPU Cache

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

DRAM access, slow

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

Cache hit

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

Request

Response
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
Flush+Reload

Shared Memory

ATTACKER
flush
access

VICTIM
access
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

cached

cached
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
Flush+Reload

ATTACKER
flush
access

Shared Memory

VICTIM
access
flush
access

fast if victim accessed data, slow otherwise

Flush+Reload

ATTACKER

Shared Memory

VICTIM

access

Daniel Gruss — Graz University of Technology
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits

Daniel Gruss — Graz University of Technology
Memory Access Latency

Access time [CPU cycles]

Number of accesses

- **Cache Hits**
- **Cache Misses**

Daniel Gruss — Graz University of Technology
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 00:02 26 8056
/usr/lib/x86_64-linux-gnu/gedit/libgedit.so

shark% ./spy
7. Serve with cooked and peeled potatoes
Wait for an hour
Wait for an hour

LATENCY
1. Wash and cut vegetables

2. Pick the basil leaves and set aside

3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened
1. Wash and cut vegetables

2. Pick the basil leaves and set aside

3. Heat 2 tablespoons of oil in a pan

4. Fry vegetables until golden and softened
```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
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);
```
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```c
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);
```

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

```
char data = *(char*)0xfffffffff81a000e0;
printf("%c\n", data);
```

- Kernel addresses are not accessible
char data = *(char*)0xffffffff81a000e0;
printf("%c\n", data);

Kernel addresses are not accessible

Are privilege checks also done when executing instructions out of order?
Adapted code

1. *(volatile char*)0;
2. array[84 * 4096] = 0; // unreachable
• Adapted code

1  *(volatile char*)0;
2  array[84 * 4096] = 0; // unreachable

• Static code analyzer is not happy

1  warning: Dereference of null pointer
2          *(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

<table>
<thead>
<tr>
<th>Page</th>
<th>Access time [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>150</td>
<td>3</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>250</td>
<td>5</td>
</tr>
</tbody>
</table>

- "Unreachable" code line was actually executed
- Exception was only thrown afterwards
• Combine the two things

```c
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```
• Combine the two things

```c
char data = *(char *)0xfffffffff81a0000e0;
array[data * 4096] = 0;
```

• Then check whether any part of array is cached
1. Flush+Reload over all pages of the array

2. 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
CAN YOU ENHANCE THAT
meltdown@meltdown ~/ppm2 % taskset 1 ./imgdump 0x375a00000 14919 > output.flif
Reading from 0xffff880375a00000
Leaking Passwords from your Password Manager

Daniel Gruss — Graz University of Technology
How to mitigate Meltdown?
Take the kernel addresses...

- Kernel addresses in user space are a problem
Take the kernel addresses...

- Kernel addresses in user space are a problem
- Why don’t we take the kernel addresses...
...and remove them if not needed?
...and remove them

• ...and remove them if not needed?
• User accessible check in hardware is not reliable
CAN'T LEAK DATA

IF THERE IS NO DATA
Kernel Address Isolation to have Side channels Efficiently Removed
Kernel Address Isolation to have Side channels Efficiently Removed
Without KAISER:

Shared address space

User memory \rightarrow Kernel memory

context switch

With KAISER:

User address space

User memory \rightarrow Not mapped

context switch

Kernel address space

SMAP + SMEP \rightarrow Kernel memory

Interrupt dispatcher

Daniel Gruss — Graz University of Technology
• 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)
• Microsoft implemented similar concept in Windows 10
We published KAISER in July 2017.

Intel and others improved and merged it into Linux as KPTI (Kernel Page Table Isolation).

Microsoft implemented similar concept in Windows 10.

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

MELTDOWN

SPECTRE

Daniel Gruss — Graz University of Technology
Prosciutto
Funghi
Diavolo
Diavolo
Diavolo
»A table for 6 please«
Speculative Cooking
»A table for 6 please«
What does Spectre do?

- Mistrains branch prediction
What does Spectre do?

- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
What does Spectre do?

- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls
What does Spectre do?

- Mistrains branch prediction
- CPU speculatively executes code which should not be executed
- Can also mistrain indirect calls

→ Spectre “convinces” program to execute code
index = 0;

char* data = "textKEY";

if (index < 4)
    Prediction
    LUT[data[index] * 4096]
else
    0
Spectre (variant 1)

index = 0;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction

Daniel Gruss | Graz University of Technology
index = 0;

char* data = "textKEY";

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

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
    then
        Predict
        Speculate
        LUT[data[index] * 4096]
    else
        0
index = 1;

char* data = "textKEY";

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

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 2;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
    Prediction
else
    LUT[data[index] * 4096] 0
Spectre (variant 1)

```c
index = 3;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

Prediction

else

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

LUT[data[index] * 4096]

else

0

else
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)

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

char* data = "textKEY";

if (index < 4)
    then
        LUT[data[index] * 4096]
    else
        Prediction
        0
index = 5;

char* data = "textKEY";

if (index < 4)
then
LUT[data[index] * 4096]
else
Prediction
0
index = 5;

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

```c
if (index < 4) {
    Speculate
    then
    Prediction
    LUT[data[index] * 4096]
} else {
    0
}
```
index = 5;

char* data = "textKEY";

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

else
0

Prediction

Execute

Daniel Gruss — Graz University of Technology
index = 6;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 6;

char* data = "textKEY";

if (index < 4)

then

Prediction

LUT[data[index] * 4096]

else

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

a->move();
```

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

a->move()

fly()

swim()

LUT[data[index] * 4096]

swim()

Prediction

Speculate

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

```plaintext
a->move()
```

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

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

a->move()

fly()

fly()

swim()

LUT[data[index] * 4096]

Prediction

0
Animal* a = bird;

Speculate

LUT[data[index] * 4096]

a->move()

fly()

Prediction

fly()

swim()

0
```cpp
Animal* a = bird;

a->move();

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

a->move()

fly()

fly()

Prediction

LUT[data[index] * 4096]

0
Animal* a = fish;

Speculate

fly()

LUT[data[index] * 4096]

Prediction

fly()

swim()

0
Animal* a = fish;

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

a->move()

fly()

swim()

swim()

Prediction

LUT[data[index] * 4096]

0
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]
Spectre (variant 4)

```c
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Consider:

- Prediction

Ignore:

- Speculate

LUT[data[index] * 4096]

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore
index = 1;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = 1;

index = index & 0x3;  // sanitization

char* data = "textKEY";

consider

Prediction

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";

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

index = index & 0x3; \textit{\# sanitization}

\texttt{char* data = "textKEY";}

\texttt{LUT[data[index] \times 4096]}

\texttt{LUT[data[index] \times 4096]}

\textbf{Prediction}

\textit{consider} \quad \textit{ignore}
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

```c
index = 2;

index = index & 0x3;  // sanitization

char* data = "textKEY";
```

Consider:

- Prediction

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

Ignore:

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

index = index & 0x3; // sanitization

char* data = "textKEY";

Spectre (variant 4)

index = 3;

index = index & 0x3;  // sanitization

char* data = "textKEY";

Predictation

LUT[data[index] * 4096]

consider

ignore
Spectre (variant 4)

```c
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Diagram:
- Consider Prediction
  - LUT[data[index] * 4096]
- Ignore
  - LUT[data[index] * 4096]
Spectre (variant 4)

```c
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Consider

Prediction

LUT[data[index] * 4096]

Ignore

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

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

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

char* data = "textKEY";
```

Consideration:
- LUT[data[index] * 4096]
- LUT[data[index] * 4096]

Prediction:
- Consider
- Ignore
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
```c
index = 4;
index = index & 0x3; // sanitization
char* data = "textKEY";
```

**Diagram:**
- **Execute**
  - `index = 0`
  - `LUT[data[index] * 4096]`
- **Prediction**
- **Ignore**
  - `LUT[data[index] * 4096]`
index = 5;

index = index & 0x3;  // sanitization

char* data = "textKEY";

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

consider

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

consider

Prediction

ignore

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

```c
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Prediction

consider

ignore

LUT[data[index] * 4096]  

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  

Prediction

Speculate

consider

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]
- 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!
Mitigating Spectre

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

- 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!

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

Workaround after every bounds check:
- x86: LFENCE
- ARM: CSDB

Available on all Intel CPUs, 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
- 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
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 load no speculate
```
Spectre Variant 1 Mitigations

- Speculation barrier requires compiler supported

Daniel Gruss — Graz University of Technology
Spectre Variant 1 Mitigations

- Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC

Daniel Gruss — Graz University of Technology
Speculation barrier requires compiler supported
- Already implemented in GCC, LLVM, and MSVC
- Can be automated (MSVC) $\rightarrow$ not really reliable
Speculation barrier requires compiler supported
Already implemented in GCC, LLVM, and MSVC
Can be automated (MSVC) → not really reliable
Explicit use by programmer: \_\_builtin\_load\_no\_speculate
// Unprotected

int array[N];

int get_value(unsigned int n) {
    int tmp;
    if (n < N) {
        tmp = array[n]
    } else {
        tmp = FAIL;
    }
    return tmp;
}
// Unprotected

int array[N];

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

// Protected

int array[N];

int get_value(unsigned int n) {
    int tmp;
    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
Spectre Variant 1 Mitigations

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

Daniel Gruss — Graz University of Technology
Speculation barrier works if affected code constructs are known

Programmer has to fully understand vulnerability

Automatic detection is not reliable
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

Daniel Gruss — Graz University of Technology
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):

![Diagram](https://example.com/diagram.png)
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)
Retpoline (compiler extension)

1. `push <call_target>`
2. `call 1f`
3. `2: ; speculation will continue here`
4. `lfence ; speculation barrier`
5. `jmp 2b ; endless loop`
6. `1:`
7. `lea 8(%rsp), %rsp ; restore stack pointer`
8. `ret ; the actual call to <call_target>`

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

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

→ always predict to enter an endless loop

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

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

→ always predict to enter an endless loop

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

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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:
Spectre Variant 2 Mitigations (Software)

Retpoline (compiler extension)

```
1 push <call_target>
2 call 1f
3 jmp 2b ; endless loop
4 ...
5 lea 8(%rsp), %rsp ; restore stack pointer
6 ret ; the actual call to <call_target>
```

→ always predict to enter an endless loop

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

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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- On Broadwell or newer:
  - `ret` may fall-back to the BTB for prediction
  → microcode patches to prevent that
ARM provides hardened Linux kernel
ARM provides hardened Linux kernel
Clears branch-predictor state on context switch
ARM provides hardened Linux kernel
Clears branch-predictor state on context switch
Either via instruction (BPIALL)...
ARM provides hardened Linux kernel
Clears branch-predictor state on context switch
Either via instruction (BPIALL)...
...or workaround (disable/enable MMU)
ARM provides hardened Linux kernel
- Clears branch-predictor state on context switch
  - Either via instruction (BPIALL)...
  - ...or workaround (disable/enable MMU)
- Non-negligible performance overhead ($\approx$ 200-300 ns)
Intel released microcode updates
Intel released microcode updates

- Disable store-to-load-forward speculation
- Performance impact of 2–8%
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
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 branch prediction
turning off speculative execution entirely
has no effect on Meltdown!
melts down the isolation provided by the
user-accessible 38-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 branch (mis)prediction
turning off speculative execution entirely would work
has nothing to do with the user
KAISER has no effect on Spectre at all
<table>
<thead>
<tr>
<th>Meltdown</th>
<th>Spectre</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Out-of-Order Execution</td>
<td>• Speculative Execution (subset of Out-of-Order Execution)</td>
</tr>
</tbody>
</table>
Meltdown vs. Spectre

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

Spectre
- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
Meltdown

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

Spectre

- Speculative Execution (subset of Out-of-Order Execution)
- fundamentally builds on branch (mis)prediction
- turning off speculative execution entirely would work
Meltdown

- Out-of-Order Execution
- has nothing to do with branch 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 branch (mis)prediction
- turning off speculative execution entirely would work
- has nothing to do with the user_accessible-bit
Meltdown

- Out-of-Order Execution
- has nothing to do with branch 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 branch (mis)prediction
- 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 branch 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 branch (mis)prediction
- 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

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
Meltdown
- performs illegal memory accesses → we need to take care of processor exceptions

Spectre
- performs only legal memory accesses
Meltdown vs. Spectre

Meltdown
- performs illegal memory accesses → 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

Daniel Gruss — Graz University of Technology
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
DRAM organization

channel 0

channel 1
DRAM organization

channel 0

back of DIMM: rank 1

front of DIMM: rank 0

channel 1
DRAM organization

- **chip**
- **channel 0**
- **channel 1**
- **front of DIMM: rank 0**
- **back of DIMM: rank 1**
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 → repetitive refresh necessary

Maximum interval between refreshes to guarantee data integrity

Cells leak faster upon proximate accesses → 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 → repetitive refresh necessary
- Maximum interval between refreshes to guarantee data integrity
- Cells leak faster upon proximate accesses → Rowhammer
• Cells leak → repetitive refresh necessary
• Maximum interval between refreshes to guarantee data integrity
• Cells leak faster upon proximate accesses → Rowhammer
Rowhammer

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

Row buffer

DRAM bank

bit flips in row 2!
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
Hammering techniques

- 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

![DRAM bank diagram with activate signal highlighted]
#1 - Single-sided hammering

DRAM bank

activate
#1 - Single-sided hammering

![DRAM bank diagram]

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

![Diagram of a DRAM bank with activation process]
#1 - Single-sided hammering

![Diagram of DRAM bank with rows and columns labeled with binary values and an arrow labeled 'activate'.]
#1 - Single-sided hammering

![Diagram of DRAM bank with bit flips](image)
#2 - Double-sided hammering

![Diagram of DRAM bank with 'activate' highlighted row]
#2 - Double-sided hammering

![DRAM bank diagram]

- Activate

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

![DRAM bank diagram]

activate

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

[Diagram of a DRAM bank with binary values and an 'activate' label pointing to a specific row]
#2 - Double-sided hammering

DRAM bank

activate
#2 - Double-sided hammering

DRAM bank

activate

bit flips
#3 - One-location hammering

![DRAM bank diagram](image)

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
#3 - One-location hammering

![Diagram of a DRAM bank with rows labeled 111111111111111 and one row highlighted in yellow labeled 111111111111111 with the word "activate" pointing to it.](image-url)
#3 - One-location hammering

![Diagram of a DRAM bank with a highlighted block of memory.]
#3 - One-location hammering

DRAM bank

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

1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 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]

Daniel Gruss — Graz University of Technology
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?

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 reproduciblenip pattern!

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

- They are not random → highly reproducible flip pattern!
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
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
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
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
  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.
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?
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

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

JE

XORB

0 1 1 1 0 1 0 0

0 0 1 1 0 1 0 0

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

JE
0 1 1 1 0 1 0 0

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

JE

<prefix>
Opcode Flipping - Conditional Jump

JE
0 1 1 1 0 1 0 0

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

JE

0 1 1 1 0 1 0 0

JO

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

JE

0 1 1 1 0 1 0 0

JBE

0 1 1 1 0 1 1 0

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

JE
0 1 1 1 0 1 0 0

JNE
0 1 1 1 0 1 0 1
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

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

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
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
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”
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
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
When you read the manuals...

After learning about a side channel you realize:
When you read the manuals...

After learning about a side channel you realize:

- the side channels were documented in the 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

Seabelts
More Seabelts
Airbags
More Airbags
ABS

Daniel Gruss — Graz University of Technology
Attacks vs. Defenses

54

Daniel Gruss — Graz University of Technology
Attacks vs. Defenses

- 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?
• 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

www.tugraz.at

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?

Daniel Gruss | Graz University of Technology
A unique chance to
- rethink processor design
A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
Conclusions

A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
- dedicate more time into identifying problems and not solely in mitigating known problems
Microarchitectural Attacks:
From the Basics to Arbitrary Read and Write Primitives without any Software Bugs

Daniel Gruss
June 19, 2018
Graz University of Technology