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

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

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

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

DRAM access, slow

printf("%d", i);

Cache miss

printf("%d", i);

Cache hit

DRAM access,
CPU Cache

#define _GNU_SOURCE
#include <stdio.h>

int main()
{
    int i;
    printf("%d", i);
    return 0;
}

1. Cache miss
2. Request
3. Response
4. Cache hit
5. No DRAM access, much faster

DRAM access, slow

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

Daniel Gruss — Graz University of Technology
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
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

Shared Memory

access
Flush + Reload

Attacker

Flush
Access

Shared Memory

Fast if victim accessed data, slow otherwise

Victim

Daniel Gruss — Graz University of Technology
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits

Cache Misses

Daniel Gruss — Graz University of Technology
Cache Template Attack Demo

```bash
% sleep 2; ./spy 300 7f05140a4000-7f051417b000 r-xp 0x20000 00:02 26
/usr/lib/x86_64-linux-gnu/gedit/libgdesk.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
int width = 10, height = 5;

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

int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
int width = 10, height = 5;

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

int area = width * height;

printf("Area %d x %d = %d\n", width, height, area);
```
char data = *(char*)0xfffffffff81a000e0;
printf("%c\n", data);
```
Building Meltdown

```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);
```

```
segfault at ffffffffff81a000e0 ip 0000000000400535
   sp 00007ffce4a80610 error 5 in reader
```

- Kernel addresses are not accessible
Building Meltdown

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

segfault at ffffffff81a000e0 ip 00000000000400535
sp 00007ffce4a80610 error 5 in reader

- Kernel addresses are not accessible
- Are privilege checks also done when executing instructions out of order?
Adapted code

```c
*(volatile char*) 0;
array[84 * 4096] = 0; // unreachable
```
• Adapted code

```c
*(volatile char*)0;
array[84 * 4096] = 0; // unreachable
```

• Static code analyzer is not happy

```c
warning: Dereference of null pointer
       *(volatile char*)0;
```
- Flush+Reload over all pages of the array

- “Unreachable” code line was actually executed
- Flush+Reload over all pages of the array

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

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

```c
char data = *(char*)0xfffffffff81a000e0;
array[data * 4096] = 0;
```

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

• Index of cache hit reveals data
- Flush+Reload over all pages of the array

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

Terminal

mschwarz@lab06:~/Documents$
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

- ...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

Daniel Gruss — Graz University of Technology
KAISER /ˈkʌɪzə/
1. [german] Emperor, ruler of an empire
2. largest penguin, emperor penguin

Kernel Address Isolation to have Side channels Efficiently Removed
Without KAISER:

- Shared address space
  - User memory
  - Kernel memory
  - Context switch

With KAISER:

- User address space
  - User memory
  - Not mapped
  - SMAP + SMEP
  - Kernel memory
  - Context switch
  - Interrupt dispatcher
  - Kernel address space
  - Address space switch
- 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
Meltdown and Spectre

MELTDOWN

SPECTRE
Prosciutto
Funghi
Diavolo
Diavolo
Diavolo
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)
then
LUT[data[index] * 4096]
else
0
index = 0;

char* data = "textKEY";

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

char* data = "textKEY";

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

Prediction

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

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

char* data = "textKEY";

if (index < 4)
	then
	Prediction
	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)

Speculate

LUT[data[index] * 4096]

then

Prediction

else

0

Daniel Gruss | Graz University of Technology
index = 1;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 2;

char* data = "textKEY";

if (index < 4)

    then

        LUT[data[index] * 4096]

    else

        0
index = 2;

char* data = "textKEY";

if (index < 4)
    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)

    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

Speculate

then

LUT[data[index] * 4096]

else

Prediction

else

0
index = 3;

char* data = "textKEY";

if (index < 4)

Prediction

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 4;

char* data = "textKEY";

if (index < 4)

Speculate

then

LUT[data[index] * 4096]

else

Prediction

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

char* data = "textKEY";

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

char* data = "textKEY";

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

Execute

0
index = 6;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096] 0

else
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
    LUT[data[index] * 4096]
else
    Prediction
    0
index = 6;

char* data = "textKEY";

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

a->move()

fly()

swim()

swim()

LUT[data[index] * 4096]

Prediction

0
Animal* a = bird;

LUT[data[index] * 4096]

a->move()

fly() -> swim()

swim() -> Speculate

Prediction

0
Animal* a = bird;

a->move();

fly();

swim();

LUT[data[index] * 4096]

0

swim();

Prediction
Animal* a = bird;

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

a->move()

fly()  fly()

LUT[data[index] * 4096]

Prediction

0
```cpp
Animal* a = bird;

a->move();

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

a->move();

fly()

fly()

swim()

LUT[data[index] * 4096]

Prediction

0
Spectre (variant 2)

```cpp
Animal* a = fish;
```

LUT[data[index] * 4096]

0
Animal* a = fish;

Speculate

LUT[data[index] * 4096]

0

fly()

Prediction

fly()

a->move()

swim()
Animal* a = fish;

Prediction

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

a->move();

LUT[data[index] * 4096]
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]

consider

Prediction

ignore

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

index = index & 0x3; // sanitization

`char* data = "textKEY";`

consider

Prediction

`LUT[data[index] * 4096]`

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]


```c
index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

Prediction

consider
go

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]

Prediction

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Speculate

consider

ignore

Prediction

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  

consider  
Prediction  
ignore 

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

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

index = 3;

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";

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

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";
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Speculate

consider

Prediction

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore
Spectre (variant 4)

```c
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

Prediction

Speculate

consider

ignore

Daniel Gruss — Graz University of Technology
index = 5;

index = index & 0x3;  // sanitization

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

index = index & 0x3; // sanitization

char* data = "textKEY";

consider
Prediction

ignore

LUT[data[index] * 4096]

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";
index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";
• 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!
• 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
• Workaround: insert instructions stopping speculation
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: `builtin load 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`
// Unprotected

int array[N];

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

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

// Protected
int array[N];

int get_value(unsigned int n) {
    int *lower = array;
    int *ptr = array + n;
    int *upper = array + N;
    return __builtin_load_no_speculate(ptr, lower, upper, FAIL);
}
```
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
Spectre Variant 1 Mitigations

- 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
Spectre Variant 1 Mitigations

- 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
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):

\[ \text{IBRS implementation diagram} \]
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)

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

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

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

→ always predict to enter an endless loop

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

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

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

→ always predict to enter an endless loop

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

```assembly
1:    push <call_target>
2:    call 1f
3:    ; speculation will continue here
4:    lfence
5:    ; speculation barrier
6:    jmp 2b
7:    ; endless loop
1:
8:    lea 8(%rsp), %rsp
9:    ; restore stack pointer
10:  ret
11:  ; 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)

```
1 | push <call_target>
2 | call 1f
3:   ; speculation will continue here
4 | lfence  ; speculation barrier
5 | jmp 2b  ; endless loop
1:   
7 | lea 8(%rsp), %rsp ; restore stack pointer
8 | 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 (≈ 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 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 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</td>
</tr>
<tr>
<td></td>
<td>Out-of-Order Execution)</td>
</tr>
</tbody>
</table>
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 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

**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

- 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 with exception suppression using TSX. Exception suppression with branch misprediction.

Spectre performs only legal memory accesses. It has nothing to do with exception handling or suppression.

Daniel Gruss — Graz University of Technology
Meltdown vs. Spectre

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
- 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
- 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
DRAM organization

channel 0

channel 1
DRAM organization

channel 0

channel 1

back of DIMM: rank 1

front of DIMM: rank 0

Daniel Gruss — Graz University of Technology
DRAM organization

- Channel 0
- Channel 1

Back of DIMM: rank 1

Front of DIMM: rank 0

Chip
## DRAM Organization

![DRAM Chip Diagram]

### Bank 0

<table>
<thead>
<tr>
<th>Row</th>
</tr>
</thead>
<tbody>
<tr>
<td>row 0</td>
</tr>
<tr>
<td>row 1</td>
</tr>
<tr>
<td>row 2</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>row 32767</td>
</tr>
</tbody>
</table>

**Row Buffer**
DRAM organization

Bank 0

- row 0
- row 1
- row 2
- ...
- row 32767

Row buffer

64k cells
1 capacitor, 1 transistor each

www.tugraz.at

Daniel Gruss — Graz University of Technology
- 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
• 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
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
Hammering techniques

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 binary data in yellow highlighted rows. The diagram shows a bank of memory with rows of binary 1s and 0s, with a highlighted row indicating activation.]
#1 - Single-sided hammering

![Diagram of a DRAM bank with binary values and an activate arrow pointing to the top row.](image)
#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 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1
#1 - Single-sided hammering

![DRAM Bank Diagram]

- Activate

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

DRAM bank

activate
#1 - Single-sided hammering

![DRAM bank diagram]

Bit flips
#2 - Double-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
#2 - Double-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
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
#2 - Double-sided hammering

[Diagram of a DRAM bank with binary numbers and an arrow labeled 'activate']
#2 - Double-sided hammering

![DRAM bank diagram]

activate
#2 - Double-sided hammering

[Diagram of a DRAM bank with binary values and an 'activate' indicator]
#2 - Double-sided hammering

![Diagram of DRAM bank with bit flips and activate notations.]

Daniel Gruss — Graz University of Technology
#3 - One-location hammering

![DRAM bank diagram with activate highlighted]

Daniel Gruss — Graz University of Technology
#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
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 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

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
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

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
#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" bits
3. Place data structure there
4. Trigger bit flip again
How to exploit random bit flips?

They are not random!

highly reproducible

bit pattern!

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

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” ips
3. Place data structure there
4. Trigger bit flip again
They are not random \(\rightarrow\) highly reproducible flip pattern!
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
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 $\rightarrow$ 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.

`sudo`
What if we cannot target kernel pages?

Many applications perform actions as root. They can be used by unprivileged users as well. 

Daniel Gruss — Graz University of Technology
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

0 1 1 1 0 1 0 0

→

HLT

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

JE 0110100  XORB 00110100

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

JE

PUSHQ

0 1 1 1 0 1 0 0

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

JE

0 1 1 1 0 1 0 0

<prefix>

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

JE

0 1 1 1 0 1 0 0

JL

0 1 1 1 1 1 0 0

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

JE

01110100

JO

011110000
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 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
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”
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”
What do we learn from it?

We have ignored microarchitectural attacks for many many years:

- attacks on crypto $\rightarrow$ “software should be fixed”
- attacks on ASLR $\rightarrow$ “ASLR is broken anyway”
- attacks on SGX and TrustZone $\rightarrow$ “not part of the threat model”
- 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
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
When you read the manuals...

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

Daniel Gruss — Graz University of Technology
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?
• 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. However, we have overlooked the dangers of attacks such as Meltdown and Spectre for decades. We do not know all problems, but are we hammering on a small subset of problems and forgetting about the bigger picture?
Conclusions

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