Score-to-Leak Forwarding
There and Back Again

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
October 2, 2020

Graz University of Technology  
CISPA Helmholtz Center for Information Security
Claudio Canella
PhD student @ Graz University of Technology

@cc0x1f

claudio.canella@iaik.tugraz.at
Who am I?

Lukas Giner

PhD student @ Graz University of Technology

@redrabbyte

lukas.giner@iaik.tugraz.at
Who am I?

Michael Schwarz

Faculty @ CISPA Helmholtz Center for Information Security

@misc0110

michael.schwarz@cispa.saarland
Motivation

- How do loads handle previous stores?
Motivation

• How do loads handle previous stores?
• How is Meltdown mitigated in hardware?
Motivation

• How do loads handle previous stores?
• How is Meltdown mitigated in hardware?
• Can we abuse these mechanisms for new attacks?
Motivation

• How do loads handle previous stores?
• How is Meltdown mitigated in hardware?
• Can we abuse these mechanisms for new attacks?
• Can we mitigate such attacks efficiently?
In-Order Execution

- Mental model of CPU is simple
In-Order Execution

- Mental model of CPU is simple
- Instructions are executed in program order
In-Order Execution

- Mental model of CPU is simple
- Instructions are executed in program order
- Pipeline stalls when stages are not ready
In-Order Execution

- Mental model of CPU is simple
- Instructions are executed in program order
- Pipeline stalls when stages are not ready
- If data is not cached, we need to wait
In-Order Execution

Instructions are...
  • fetched (IF) from the L1 Instruction Cache
In-Order Execution

- Instructions are...
  - fetched (IF) from the L1 Instruction Cache
  - decoded (ID)
In-Order Execution

- Instructions are...
  - fetched (IF) from the L1 Instruction Cache
  - decoded (ID)
  - executed (EX) by execution units
In-Order Execution

- Instructions are...
  - fetched (IF) from the L1 Instruction Cache
  - decoded (ID)
  - executed (EX) by execution units
- Memory access is performed (MEM)
In-Order Execution

- Instructions are...
  - fetched (IF) from the L1 Instruction Cache
  - decoded (ID)
  - executed (EX) by execution units
- Memory access is performed (MEM)
- Architectural register file is updated (WB)
Reality: Out-of-Order Execution, Transient Execution

• No dependency between instructions? Why wait!
Reality: Out-of-Order Execution, Transient Execution

- No dependency between instructions? Why wait!
- Execute out of order
Reality: Out-of-Order Execution, Transient Execution

- No dependency between instructions? Why wait!
- Execute out of order, retire in order
No dependency between instructions? Why wait!
Execute *out of order*, retire *in order*
What happens when an *instruction faults*?
Reality: Out-of-Order Execution, Transient Execution

- No dependency between instructions? Why wait!
- Execute **out of order**, retire **in order**
- What happens when an instruction **faults**?
- Undo out-of-order effects → instructions were **transient**
Side Effects of Transient Execution

- Transient instructions are undone, what good are they?
Side Effects of Transient Execution

- Transient instructions are undone, what good are they?
- Some changes are persistent, but invisible architecturally
Side Effects of Transient Execution

- Transient instructions are undone, what good are they?
- Some changes are persistent, but invisible architecturally
  → encode data in microarchitectural state
Side Effects of Transient Execution

- Transient instructions are undone, what good are they?
- Some changes are persistent, but invisible architecturally
  → encode data in microarchitectural state
- The cache is a simple solution
Measuring Cache State

```
mov var, rax
mov var, rbx
```
Measuring Cache State

mov var, rax

mov var, rbx

Cache miss
Measuring Cache State

mov var, rax
mov var, rbx
Measuring Cache State

```
mov var, rax
mov var, rbx
```
Measuring Cache State

```
mov var, rax
mov var, rbx
```

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Measuring Cache State

`mov var, rax`

`mov var, rbx`

Cache miss

Cache hit

Request

Response
Measuring Cache State

DRAM access, slow

mov var, rax  # Cache miss

mov var, rbx  # Cache hit

Request

Response

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Measuring Cache State

- `mov var, rax` (Cache miss)
- `mov var, rbx` (Cache hit)

DRAM access, slow

No DRAM access, much faster

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Memory Subsystem

Core

Execution Engine

Frontend

Memory Subsystem

Write Back
Memory Subsystem

Core

Execution Engine

Frontend

Memory Subsystem

Load Buffer

Store Buffer

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Memory Subsystem

Core

Execution Engine

Frontend

Load Buffer
Store Buffer

L1 Data Cache

DTLB
LFB

L2 Cache
L3 Cache

DRAM

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Store Buffer

\[
\text{mov data} \rightarrow [\text{address}]
\]

\[
\text{mov [address]} \rightarrow \text{reg}
\]
mov data → [ address ]

mov [ address ] → reg

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Store Buffer

\[
\text{mov data} \rightarrow \text{[ address ]}
\]

\[
\text{mov [ address ]} \rightarrow \text{reg}
\]
mov data → [ address ]

mov [ address ] → reg

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
mov data → [ address ]

mov [ address ] → reg
Store Buffer

\[ \text{mov data} \rightarrow [ \text{address} ] \]

\[ \text{mov [ address]} \rightarrow \text{reg} \]

**Execution Unit**
- Store Data
- Load Data
- Store Buffer
- Load Buffer

**Memory Subsystem**
- L1 Data Cache

**Store-to-load forwarding**

Match

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
**Store Buffer Match Logic**

**True Positive**

Addresses: $=\quad$

Forwarding: $\checkmark$

**True Negative**

Addresses: $\neq$

Forwarding: $\times$
Store Buffer Match Logic

**True Positive**
Addresses: $=\quad$
Forwarding: $\checkmark$

**True Negative**
Addresses: $\neq$
Forwarding: $\times$

Store-to-Leak
Store Buffer Match Logic

**True Positive**

- Addresses
  - =

- Forwarding
  - ✓

**False Positive**

- =

- ✓

**True Negative**

- Addresses
  - ≠

- Forwarding
  - ✗

**False Negative**

- =

- ✗

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Store Buffer Match Logic

- **True Positive**
  - Addresses: $=$
  - Forwarding: ✔️

- **True Negative**
  - Addresses: $\neq$
  - Forwarding: ✗

- **False Positive**
  - Fallout: ✔️

- **False Negative**
  - Spectre-STL: ✔️

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
• Optimization during transient execution
Store Buffer Match Logic

- Optimization during transient execution
- Failures not visible architecturally
Store Buffer Match Logic

- Optimization during transient execution
- Failures not visible architecturally
- Store-to-Leak relies on correct matching
Store Buffer Match Logic

- Optimization during transient execution
- Failures not visible architecturally
- Store-to-Leak relies on correct matching
- Only stores to valid addresses are forwarded
[The store execution phase] Fills the store buffers with linear and physical address and data.
[The store execution phase] Fills the store buffers with linear and physical address and data. Once store address and data are known, the store data can be forwarded to the following load operations that need it. [...]

**Intel Architecture Optimization Reference Manual**
Virtual Addresses

- Virtual address that is **physically backed** → valid
• Virtual address that is physically backed → valid
• Permission checks are deferred
Virtual Addresses

- Virtual address that is **physically backed** → valid
- Permission checks are **deferred**
  → Exploited in Meltdown attacks
Virtual Addresses

- Virtual address that is **physically backed** → valid
- Permission checks are **deferred**
  → Exploited in Meltdown attacks
  → Store-to-load forwarding on **inaccessible** addresses
*kernel = ’X’
*kernel = 'X'
Store-to-load Forwarding on Inaccessible Addresses

*kernel = 'X'

mem[*kernel] Fault

Out of order

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Store-to-load forwarding?

\[ *\text{kernel} = 'X' \]

\[ \text{mem}[*\text{kernel}] \]

Out of order

Fault
Kernel

0xffff ffff 8000 0000

Modules

0xffff ffff efff ffff

0xffff ffff bfff ffff

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
KASLR

Kernel

Modules

0xffff ffff 8000 0000

0xffff ffff bfff ffff

0xffff ffff efff ffff

Kernel

Modules

0xffff ffff 8000 0000

0xffff ffff bfff ffff

0xffff ffff efff ffff

Kernel

Modules

0xffff ffff 8000 0000

0xffff ffff bfff ffff

0xffff ffff efff ffff

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Data Bounce

\[ *0xffff\ ffff\ 8000\ 0000 = 'X' \]
\[ \text{mem}[*0xffff\ ffff\ 8000\ 0000] \]

Memory Subsystem
- Store Buffer
- Load Buffer

\text{mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Data Bounce

*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Memory Subsystem

Store Buffer

Load Buffer

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Data Bounce

*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Memory Subsystem
Store Buffer
Load Buffer

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
*0xffffffff 8040 0000 = 'X'
mem[*0xffffffff 8040 0000]

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Store Buffer
Load Buffer
*0xffffffff 8060 0000 = 'X'

Memory Subsystem

Store Buffer

Load Buffer

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Data Bounce

*0xffff ffff 8080 0000 = 'X'

mem[*0xffff ffff 8080 0000]

Memory Subsystem

Store Buffer

Load Buffer

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Data Bounce

\[ \text{mem}[\text{**0xffff ffff 80a0 0000**}] = 'X' \]

---

**Memory Subsystem**

Store Buffer

Load Buffer

---

\text{mem:} ABCDEFGHIJKLMNOPQRSTUVWXYZ

---

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Data Bounce

*0xffff ffff 80a0 0000 = 'X'

mem[*0xffff ffff 80a0 0000]

Memory Subsystem

Store Buffer

Load Buffer

Store-to-load forwarding

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Data Bounce

\[ \text{mem}\left[ \text{0xffff ffff 80a0 0000} \right] = 'X' \]

Memory Subsystem

Store Buffer

Load Buffer

Store-to-load forwarding

mem: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
What if an address is valid but unknown?
• Forward on **first** try → in TLB
Repeating Data Bounce

- Forward on first try $\rightarrow$ in TLB
- On second try $\rightarrow$ valid, not in TLB
Repeating Data Bounce

- Forward on first try → in TLB
- On second try → valid, not in TLB
- Not forwarded → unmapped
*0xffff ffff 8000 0000 = 'X'
mem[*0xffff ffff 8000 0000]
*0xffff ffff 8000 0000 = ’X’

mem[*0xffff ffff 8000 0000]

TLB

0xffff ffff 8040 0000

Memory Subsystem

Store Buffer

Load Buffer

Minute 21
*0xffff ffff 8000 0000 = 'X'

mem[*0xffff ffff 8000 0000]
*0xffff ffff 8000 0000 = 'X'

mem[*0xffff ffff 8000 0000]
*0xffff ffff 8000 0000 = 'X'

\[ \text{mem}[*0xffff ffff 8000 0000] \]
*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
*0xffff ffff 8020 0000 = 'X'

mem[*0xffff ffff 8020 0000]
*0xffff ffff 8020 0000 = ‘X’
mem[*0xffff ffff 8020 0000]
*0xffff ffff 8020 0000 = 'X'
mem[*0xffff ffff 8020 0000]
*0xffff ffff 8040 0000 = 'X'*

<table>
<thead>
<tr>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xffff ffff 8040 0000</td>
</tr>
<tr>
<td>0xffff ffff 8020 0000</td>
</tr>
<tr>
<td>0xffff ffff 8000 0000</td>
</tr>
</tbody>
</table>

Memory Subsystem:
- Store Buffer
- Load Buffer

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
*0xffffffff 8040 0000 = 'X'
mem[*0xffffffff 8040 0000]
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
*0xffff ffff 8040 0000 = 'X'
mem[*0xffff ffff 8040 0000]
*0xffff ffff 8060 0000 = ‘X’
mem[*0xffff ffff 8060 0000]
*0xffff ffff 8060 0000 = 'X'

mem[*0xffff ffff 8060 0000]
*0xffff ffff 8060 0000 = ’X’

`mem[*0xffff ffff 8060 0000]`
*0xffff ffff 8060 0000 = 'X'
mem[*0xffff ffff 8060 0000]
Observing Kernel Activity
Observing Kernel Activity
Observing Kernel Activity

Kernel

Modules

Bluetooth

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Observing Kernel Activity

Kernel Modules

Claudio Canella (@cc0xf), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Observing Kernel Activity
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>
if <access in bounds>

predicted

true
false
false

true
false
false
if <access in bounds>
if <access in bounds>
if <access in bounds>
index = 0

if (index < 4)
  glyph[data[index]]
else
  {}
index = 0

if (index < 4)
    then
glyph[data[index]]
    else
    Speculate

\{ \}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 0

if (index < 4)
    glyph[data[index]]
else
    {}
index = 0

if (index < 4)
    glyph[data[index]]
else
    {}
index = 0

if (index < 4)
  glyph[data[index]]
else
  {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbit), Michael Schwarz (@misc0110)
index = 1

if (index < 4)
    glyph[data[index]]
else
    {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 1

if (index < 4) then
  glyph[data[index]]
else
  {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 1

if (index < 4)

then

glyph[data[index]]

else

{}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 1

if (index < 4)

then

glyph[data[index]]

else

{}
index = 1

if (index < 4)
then

glyph[data[index]]

else


index = 1

if (index < 4)
    glyph[data[index]]
else
    {}
```
if (index < 4)
glyph[data[index]]
```
index = 2

\[ \text{if (index < 4)} \]

\[ \text{then} \]

\[ \text{glyph[data[index]]} \]

\[ \text{else} \]

\[ \text{\{} \]

\[ \text{\}} \]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 2

\[ \text{if (index < 4)} \]

\[ \text{then} \]

\[ \text{glyph}[\text{data}[\text{index}]] \]

\[ \text{else} \]

\[ \{ \} \]

\[ \text{DATA} \]

\[ \text{KEY} \]

\[ \text{data[0]} \]

\[ \text{data[1]} \]

\[ \text{data[2]} \]

\[ \text{data[3]} \]
index = 2

if (index < 4)

glyph[data[index]]
index = 2

if (index < 4)
    glyph[data[index]]
else
    {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Spectre

index = 3

if (index < 4)
  glyph[data[index]]
else
  {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 3

if (index < 4) then
   glyph[data[index]]
else
   
{}
index = 3

if (index < 4)
    glyph[data[index]]
else
    {}
index = 3

\[
\text{if (index < 4) then}\]

\[
\text{glyph}[\text{data[index]}]\]

\[
\text{else}\}

\[
\text{data[0]}
\[
\text{data[1]}
\[
\text{data[2]}
\[
\text{data[3]}
\]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
\[
\text{index} = 3
\]

The diagram illustrates a memory layout and a process flow. The memory contains the following data:

- **DATA[0]**
- **DATA[1]**
- **DATA[2]**
- **DATA[3]**

The process involves:

1. **Check the index:** If \( \text{index} < 4 \), then execute:
   
   \[
   \text{glyph[DATA[index]]}
   \]

2. Else, execute:
   
   \[
   \text{{} if (index < 4) then glyph[DATA[index]] else {} else }
   \]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 4

if (index < 4)
    `glyph[data[index]]`
else
    {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 4

if (index < 4)
then
  glyph[data[index]]
else
  {}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 4

if (index < 4)
then
glyph[data[index]]

else
{}

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
index = 4

if (index < 4)

then

glyph[data[index]]

else

{}
index = 4

if (index < 4)
    glyph[data[index]]
else
    Execute

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Speculative Fetch+Bounce

```c
if (x < len(array))
y = kernel[array[x] * 4096]
```

256 pages kernel memory (kernel)
if \ (x < \text{len}(\text{array}))

\begin{equation}
y = \text{kernel}[\text{array}[x] * 4096]
\end{equation}
if \( x < \text{len}(\text{array}) \)

\[ y = \text{kernel}[\text{array}[x] \times 4096] \]

256 pages kernel memory (kernel)

Store in TLB

Fetch+Bounce
Speculative Fetch+Bounce

if \( x < \text{len(array)} \)

\[ y = \text{kernel}[\text{array}[x] \times 4096] \]

256 pages kernel memory (kernel)

Store in TLB

TLB Hit

Fetch+Bounce
Speculative Fetch+Bounce

if (x < len(array))
y = kernel[array[x] * 4096]

256 pages kernel memory (kernel)

Store in TLB

Kernel

User

Fetch+Bounce
Meltdown

- Exploits deferred permission check in out-of-order execution
- Exploits deferred permission check in \textit{out-of-order} execution
- Can read \textit{any kernel address}
Meltdown

- Exploits deferred permission check in out-of-order execution
- Can read any kernel address
- Kernel maps all physical memory
Meltdown

- Exploits deferred permission check in out-of-order execution
- Can read any kernel address
- Kernel maps all physical memory
  - Read arbitrary memory by encoding in cache
Assumptions

1. Stalling CPU might be too costly
Assumptions

1. Stalling CPU might be too costly
2. Stalling might require redesigning parts of CPU pipeline
Reverse-Engineering Meltdown Hardware Mitigations

Assumptions

1. Stalling CPU might be too costly
2. Stalling might require redesigning parts of CPU pipeline

Hypothesis
Load is executed, returned value is zeroed out on faults

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Verifying the Hypothesis

Two tests:
1. Perform Meltdown attack
Verifying the Hypothesis

Two tests:
1. Perform Meltdown attack
2. Use Performance Counters
Verifying the Hypothesis

- Intel: CYCLE_ACTIVITY.STALLS_MEM_ANY
- AMD: Dispatch Stalls

![Graph showing vulnerability mitigation for Intel and AMD.]
Verifying the Hypothesis

- Track number of *issued* \( \mu \)OPs on the load ports
- UOPS\_DISPATCHED\_PORT.\_PORT\_2, UOPS\_DISPATCHED\_PORT.\_PORT\_3

![Bar chart](image)

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Verifying the Hypothesis

- L1D_PEND_MISS.PENDING_CYCLES

![Bar chart showing L1D_PEND_MISS.PENDING_CYCLES for user and kernel with and without mitigations.]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
• Can we use the hardware-based mitigations for an attack?
Can we use the hardware-based mitigations for an attack?

EchoLoad: fast and reliable KASLR break
• Can we use the hardware-based mitigations for an attack?
• EchoLoad: fast and reliable KASLR break
• **Encodes** the returned value in the cache
EchoLoad

```
mem[0xffff ffff 8000 0000]
```
EchoLoad

mem[*0xffff ffff 8000 0000]

stall
EchoLoad

```
mem[0xffff ffff 8020 0000]
```

stall
```
mem[*0xffffff ffff 8040 0000]
```

`stall`
EchoLoad

\[\text{mem}[\ast 0xffff\ ffff\ 8060\ 0000]\]

stall
EchoLoad

```
mem[*0xffff ffff 8080 0000]
```

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
EchoLoad

mem[*0xffff ffff 80a0 0000]

0
### Performance of EchoLoad

<table>
<thead>
<tr>
<th>CPU</th>
<th>Speculation</th>
<th>TSX</th>
<th>Segfault</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-6700K</td>
<td>Time (F-Score)</td>
<td>63 µs (0.999)</td>
<td>48 µs (1.000)</td>
</tr>
<tr>
<td>i9-9900K</td>
<td>Time (F-Score)</td>
<td>33 µs (1.000)</td>
<td>29 µs (1.000)</td>
</tr>
<tr>
<td>Xeon Silver 4208</td>
<td>Time (F-Score)</td>
<td>51 µs (0.994)</td>
<td>40 µs (1.000)</td>
</tr>
</tbody>
</table>

- Works in SGX and JavaScript

---

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Performance of EchoLoad

<table>
<thead>
<tr>
<th>CPU</th>
<th>Speculation</th>
<th>TSX</th>
<th>Segfault</th>
</tr>
</thead>
<tbody>
<tr>
<td>i7-6700K</td>
<td>Time (F-Score)</td>
<td>63 µs (0.999)</td>
<td>48 µs (1.000)</td>
</tr>
<tr>
<td>i9-9900K</td>
<td>Time (F-Score)</td>
<td>33 µs (1.000)</td>
<td>29 µs (1.000)</td>
</tr>
<tr>
<td>Xeon Silver 4208</td>
<td>Time (F-Score)</td>
<td>51 µs (0.994)</td>
<td>40 µs (1.000)</td>
</tr>
</tbody>
</table>

- Works in SGX and JavaScript
Commonalities of Previous Microarchitectural Attacks

Timing difference
Commonalities of Previous Microarchitectural Attacks

Timing difference

- between mapped and unmapped pages
Commonalities of Previous Microarchitectural Attacks

- Timing difference
  - between mapped and unmapped pages
  - for different page sizes
Commonalities of Previous Microarchitectural Attacks

Timing difference

- between mapped and unmapped pages
- for different page sizes
- between executable and non-executable pages
Fake Load Address REsponse (FLARE)

- **Executable code & data**
  - 0xffffffff 8000 0000
- **Non-Executable**
  - 0xffffffff a000 0000
- **Current Linux design**
  - 0xffffffff bfff ffff

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Step 1: Mitigating difference between mapped and unmapped pages
Fake Load Address REsponse (FLARE)

Step 2: Mitigating difference between executable and non-executable pages
mem[*0xffff ffff a000 0000]
mem[*0xffff ffff a000 0000]
mem[*0xffff ffff a000 0000]
mem[*0xffff ffff a020 0000]
\texttt{mem[*0xffffff ffff a040 0000]}
mem[*0xffffffff a060 0000]
mem[*0xffffffff a080 0000]
\texttt{mem[\ast0xffff\ ffff\ a0a0\ 0000]}
mem[*0xffffff ffff a0c0 0000]
mem[*0xffffffff a0e0 0000]
mem[*0xffff ffff a100 0000]
mem[*0xffff ffff a120 0000]
mem[*0xffff ffff a140 0000]
mem[*0xffff ffff a160 0000]
FLARE

Kernel offset [MB]

Stalls [%]

with FLARE  without FLARE

EchoLoad

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
FLARE

Kernel offset [MB]

Prefetch time

with FLARE
without FLARE

Prefetch [Gru+16]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
Kernel offset [MB]

Repetitions

with FLARE

without FLARE

Data Bounce [Sch+19]
Double page fault [HWH13]

Kernel offset [MB]

Page-fault time

with FLARE

without FLARE

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
FLARE

Kernel offset [MB]

TSX time

with FLARE vs without FLARE

DrK [JLK16]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
FLARE

Kernel offset [MB]

WTF success [%]

with FLARE  without FLARE

Fallout [Can+19]

Claudio Canella (@cc0x1f), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)
You can find our proof-of-concept implementation of FLARE on:

- https://github.com/IAIK/FLARE
• **Optimizations** introduce security problems
Conclusion

- Optimizations introduce security problems
- Mitigations can overlook edge-cases
• **Optimizations** introduce security problems
• **Mitigations** can overlook edge-cases
• Once again requires **software workaround**
Store-to-Leak Forwarding

There and Back Again

Claudio Canella (@cc0xlf), Lukas Giner (@redrabbyte), Michael Schwarz (@misc0110)

October 2, 2020

Graz University of Technology  
CISPA Helmholtz Center for Information Security


