From Random Observations to Automated Leakage Discovery

Michael Schwarz
December 2022

CISPA Helmholtz Center for Information Security
8:30 AM

STUPID BUG

7 HOURS LATER...

MUST BE LINUX

THE NEXT DAY...

THE CPU IS BROKEN!

THERE IS A TYPO.
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
- Decoded (ID)
- Executed (EX) by execution units
- Memory access is performed (MEM)
- Architectural register file is updated (WB)

- Instructions are...
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

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

- 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)
Measuring Time

\[ x = y + 1 \]
Measuring Time

\[
\begin{align*}
\text{start} &= \bigcirc \\
\text{x} &= \text{y} + 1 \\
\text{end} &= \bigcirc
\end{align*}
\]
Measuring Time

\[
\text{start} = \emptyset
\]
\[
x = y + 1
\]
\[
\text{end} = \emptyset
\]
\[
\Delta = \text{end} - \text{start}
\]
Measuring Time

\[
\begin{align*}
\text{start} & = 0 \\
x &= y + 1 \\
\text{end} & = 0 \\
\Delta &= \text{end} - \text{start}
\end{align*}
\]

1. run: \( \Delta = 302 \)
Measuring Time

start = \emptyset

\begin{align*}
  x &= y + 1 \\
  \text{end} &= \emptyset \\
  \Delta &= \text{end} - \text{start}
\end{align*}

1. run: \( \Delta = 302 \)

2. run: \( \Delta = 54 \)
Measuring Time

\[ \text{start} = \bigcirc \]
\[ x = y + 1 \]
\[ \text{end} = \bigcirc \]
\[ \Delta = \text{end} - \text{start} \]

1. run: \( \Delta = 302 \)
2. run: \( \Delta = 54 \)

Determinism?

Same code with different execution time without changes
Insanity: doing the same thing over and over again and expecting different results.

Albert Einstein
Measuring Time

\[
\begin{align*}
\text{start} &= \mathbb{0} \\
\text{end} &= \mathbb{0}
\end{align*}
\]
Measuring Time

\[ \text{start} = \emptyset \]

\[ \text{end} = \emptyset \]

\[ \Delta = \text{end} - \text{start} \]
Measuring Time

start = 0

end = 0

$\Delta = \text{end} - \text{start}$

1. run: $\Delta = 12$
Measuring Time

start = ⌧

end = ⌧

$\Delta = \text{end} - \text{start}$

1. run: $\Delta = 12$

2. run: $\Delta = 12$
Measuring Time

start = 0

delta = end - start

1. run: $\Delta = 12$

2. run: $\Delta = 12$

\[
\begin{array}{c}
\text{Runtime [cycles]} \\
\text{Delta [cycles]} \\
0 & 100k & 200k & 300k
\end{array}
\]
I HAVE NO IDEA WHAT I'M DOING
Interrupts!

App

OS

\( \Delta \)
Interrupts!

App

OS

ΔØ...........
Interrupts!

App

OS

\[ \Delta \text{ HALF} \]
Interrupts!

App

OS

ΔØ
Interrupts!

App

OS

Interrupt

IRQ Handler
Interrupts!

App

OS

\( \Delta \hat{\Delta} \)

\begin{align*}
\text{Interrupt} \\
\text{IRQ Handler}
\end{align*}
Interrupts!

App

OS

\[ \Delta \]
Interrupts!

App

OS

∆Ø

Interrupt

IRQ Handler
Interrupts!

App → Interrupt → IRQ Handler

OS

ΔΩ
Why?

Why?

Why?

Oh, that's why.
Interrupt-timing Attacks

```c
int now = rdtsc();
while (true) {
    int last = now;
    now = rdtsc();
    if ((now - last) > threshold) {
        reportEvent(now, now - last);
    }
}
```
Interrupt-timing Attacks

```c
int now = rdtsc();
while (true) {
    int last = now;
    now = rdtsc();
    if ((now - last) > threshold) {
        reportEvent(now, now - last);
    }
}
```

- Continuously acquire high-resolution timestamp
Interrupt-timing Attacks

```c
int now = rdtsc();
while (true) {
    int last = now;
    now = rdtsc();
    if ((now - last) > threshold) {
        reportEvent(now, now - last);
    }
}
```

- Continuously acquire **high-resolution timestamp**
- Interrupt → **large difference** between timestamps
Interrupt-timing Attacks

![Graph showing runtime vs. delta cycles with markers at specific points.](image-url)
Is that everything?

- Explains last experiment…
- …but what about the simple calculation?
- Noise?
Data Location

- Memory operations have different runtimes
Data Location

- Memory operations have different runtimes
- Depends where the data is located

→ Loading from cache: fast
→ Loading from memory: slow
→ Loading from disk: extremely slow
Data Location

- Memory operations have different runtimes
- Depends where the data is located
  → Loading from cache: fast
Data Location

- **Memory** operations have different **runtimes**
- Depends where the data is **located**
  - Loading from **cache**: fast
  - Loading from **memory**: slow
• **Memory** operations have different runtimes
• Depends where the data is **located**
  → Loading from **cache**: fast
  → Loading from **memory**: slow
  → Loading from **disk**: extremely slow
Data Location

- Memory operations have different runtimes
- Depends where the data is located
  - Loading from cache: fast
  - Loading from memory: slow
  - Loading from disk: extremely slow
- Can also be measured
Caching Speeds-up Memory Accesses

Access time [CPU cycles]

Number of accesses

- Cache Hits
- Cache Misses
IS MY MENTAL MODEL OF THE CPU WRONG?

NO, IT MUST BE THE MEASUREMENTS THAT ARE WRONG
Reality: (Simplified) Modern CPU

Frontend

Execution Engine

Memory Subsystem
Reality: (Simplified) Modern CPU

Frontend

Fetch

Execution Engine

Memory Subsystem
Reality: (Simplified) Modern CPU

- **Fetch + Decode**
- **Execution Engine**
- **Memory Subsystem**
Reality: (Simplified) Modern CPU
Reality: (Simplified) Modern CPU

Frontend: Fetch + Decode

Execution Engine

Execute

Memory Subsystem: Write Back

Fetch + Decode

Execution Units
ALU, AES, ...
ALU, FMA, ...
ALU, Vect, ...
ALU, Branch

Load data
Store data

AGU

CDB

Write Back

Memory Subsystem
Load Buffer
Store Buffer
L1 Data Cache
DTLB
LFB
L2 Cache
L3 Cache
DRAM
Reality: (Simplified) Modern CPU

Frontend

Execution Engine

Memory Subsystem

- Load Buffer
- Store Buffer
- L1 Data Cache
- DTLB
- LFB
- L2 Cache
- L3 Cache
- DRAM

Fetch + Decode

- Instruction Fetch & PreDecode
- Instruction Queue
- BranchPredictor
- L1 Instruction Cache
- ITLB

Allocation Queue

- 4-Way Decode
- Instruction Queue
- BranchPredictor
- L1 Instruction Cache
- ITLB
Reality: (Simplified) Modern CPU

Frontend
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode

Execution Engine

Memory Subsystem
Reality: (Simplified) Modern CPU

Frontend
- Branch Predictor
- μOP Cache
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode

Execution Engine

Memory Subsystem
- Load Buffer
- Store Buffer
- L1 Data Cache
- DTLB
- LFB
- L2 Cache
- L3 Cache
- DRAM

Fetch + Decode
- Instruction Fetch & PreDecode
- 4-Way Decode

Allocation Queue
- BranchPredictor
Reality: (Simplified) Modern CPU
Reality: (Simplified) Modern CPU
Reality: (Simplified) Modern CPU
Reality: (Simplified) Modern CPU

Frontend
- Branch Predictor
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode
- Allocation Queue

Execution Engine
- Reorder buffer
- Scheduler
- Execution Units
- ALU, AES, FMA, Vect, Branch, Load data, Store data, ALU

Memory Subsystem
- Load Buffer
- Store Buffer
- L1 Data Cache
- DTLB
- LFB
- L2 Cache
- L3 Cache
- DRAM

Fetch
- + Decode

Instruction Queue
- Instruction Fetch & PreDecode
- Branch Predictor

CDB
- Write Back

µOP Cache
- ALU, AES, FMA, Vect, Branch, Load data, Store data, ALU

MUX
- 4-Way Decode
Reality: (Simplified) Modern CPU

Frontend
- L1 Instruction Cache
- Branch Predictor
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode
- Allocation Queue
- \( \mu \text{OP} \) Cache

Execution Engine
- Reorder buffer
- Scheduler
- Execution Units
  - ALU
  - AES
  - FMA
  - Vect
  - Branch
  - Load data
  - Store data
  - \( \mu \text{OP} \)s
- CDB

Memory Subsystem
- Load Buffer
- Store Buffer
- Load data
- Store data
- ITLB
- L1 Data Cache
- DTLB
- LFB
- L2 Cache
- L3 Cache
- DRAM
Reality: (Simplified) Modern CPU

Frontend
- Branch Predictor
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode
- Allocation Queue
- μOP Cache

Execution Engine
- Reorder buffer
- Scheduler
- Execution Units
- ALU, AES, ...
- ALU, FMA, ...
- ALU, Vect, ...
- ALU, Branch
- Load data
- Store data
- Memory Subsystem
- Load Buffer
- Store Buffer
- L1 Instruction Cache
- ITLB
- L1 Data Cache
- DTLB
- LFB
Reality: (Simplified) Modern CPU

Frontend
- Branch Predictor
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode
- Allocation Queue
- µOP Cache
- MUX

Execution Engine
- Reorder buffer
- Scheduler
- Execution Units: ALU, AES, FMA, Vect, Branch, Load data, Store data, ALU

Memory Subsystem
- Load Buffer, Store Buffer
- L1 Data Cache
- DTLB
- STLB
- L2 Cache
- LFB

Fetch + Decode
- Instruction Fetch & PreDecode
- Instruction Queue
- 4-Way Decode
- Allocation Queue
- µOPs
- MUX

Execution Units
- ALU, AES, FMA, Vect, Branch, Load data, Store data, ALU
Intermezzo: CPU Architecture

- Cars all have the same interface (= architecture)
  - Steering wheel, pedals, gear stick, ...
- Some have special extensions
  - Air conditioning, cruise control, ...
- Driving skills are “compatible” with all cars
Intermezzo: CPU Microarchitecture

- Cars are implemented differently (= microarchitecture)
  - engine, fuel, motor control, ...
- Same car (“architecture”) with different engines
  - stronger or more efficient “microarchitecture”
- Drivers don’t need to know anything about internals
No thorough Description

Intel manual
(full architecture)
4778 pages

Intel optimization manual
(microarchitecture parts)
868 pages
Use of software prefetch should be limited to memory addresses that are managed or owned within the application context. Prefetching to addresses that are not mapped to physical pages can experience non-deterministic performance penalty.
Prefetch Timings on the Operating System
Prefetch Timings on the Operating System
Prefetch Timings on the Operating System
Prefetch Timings on the Operating System

![Prefetch Timings Chart](image)
I STILL HAVE NO IDEA
WHAT I'M DOING
Systematic Approach

- **Reset instruction**
- **Instruction to measure**
- **Instruction with possible side effect**
Testing A Sequence Triple

Example 1:
Seq\_measure = Seq\_trigger = Seq\_reset = INC [mem]

Example 2:
Seq\_measure = Seq\_trigger = INC [mem];
Seq\_reset = CLFLUSH [mem] INC [mem] CLFLUSH [mem]


INC [mem] Seq\_trigger INC [mem] Seq\_measure

Cold path S0
Hot path S1
Testing A Sequence Triple

Example 1:
\[\text{Seq} \text{measure} = \text{Seq trigger} = \text{Seq reset} = \text{INC } [\text{mem}]\]

Example 2:
\[\text{Seq} \text{measure} = \text{Seq trigger} = \text{INC } [\text{mem}];\]
\[\text{Seq reset} = \text{CLFLUSH } [\text{mem}];\]
\[\text{INC } [\text{mem}];\]
\[\text{CLFLUSH } [\text{mem}];\]

Cold path S0

Hot path S1
Testing A Sequence Triple

Example 1:
Seq_{reset} = Seq_{trigger} = Seq_{reset} = INC [mem]

Example 2:
Seq_{reset} = INC [mem];
Seq_{measure} = CLFLUSH [mem]

Cold path S0
Testing A Sequence Triple

Example 1:
\[\text{Seq} \text{ measure} = \text{Seq} \text{ trigger} = \text{Seq} \text{ reset} = \text{INC } [\text{mem}]\]

Example 2:
\[\text{Seq} \text{ measure} = \text{Seq} \text{ trigger} = \text{INC } [\text{mem}];\]
\[\text{Seq} \text{ reset} = \text{CLFLUSH } [\text{mem}]; \text{INC } [\text{mem}]; \text{CLFLUSH } [\text{mem}];\]

Cold path S0
Testing A Sequence Triple

Example 1:
Seq\_measure = Seq\_trigger = Seq\_reset = INC [mem]

Example 2:
Seq\_measure = Seq\_trigger = INC [mem];
Seq\_reset = CLFLUSH [mem]
INC [mem] CLFLUSH [mem]

Seq\_reset

Cold path S0

Seq\_reset

Seq\_trigger

Seq\_measure

Cold path S0
Testing A Sequence Triple

Example 1:
Seq_{measure} = Seq_{trigger} = Seq_{reset} = INC [mem]

Example 2:
Seq_{measure} = Seq_{trigger} = INC [mem];
Seq_{reset} = CLFLUSH [mem] INC [mem] CLFLUSH [mem]

Seq_{reset}  Seq_{trigger}  Seq_{measure}

Cold path S0

Hot path S1
Testing A Sequence Triple

Example 1:
Seq\_measure = Seq\_trigger = Seq\_reset = INC [mem]

Example 2:
Seq\_measure = Seq\_trigger = INC [mem];
Seq\_reset = CLFLUSH [mem] INC [mem] CLFLUSH [mem]

Seq\_reset

Cold path S0

!=

Hot path S1
Testing A Sequence Triple

Example 1: $\text{Seq}_{\text{measure}} = \text{Seq}_{\text{trigger}} = \text{Seq}_{\text{reset}} = \text{INC} \ [\text{mem}]$

Cold path S0

Hot path S1
Testing A Sequence Triple

Example 1: $Seq_{measure} = Seq_{trigger} = Seq_{reset} = \text{INC [mem]}$

- Cold path S0
  - $Seq_{reset}$
  - $Seq_{measure}$
  - $Seq_{trigger}$

- Hot path S1
  - $Seq_{reset}$
  - $Seq_{measure}$
  - $Seq_{trigger}$
Testing A Sequence Triple

Example 1: \(Seq_{measure} = Seq_{trigger} = Seq_{reset} = INC \ [mem]\)
Testing A Sequence Triple

Example 1: \( Seq_{measure} = Seq_{trigger} = Seq_{reset} = \text{INC [mem]} \)

Cold path S0

Hot path S1
Example 1: $Seq_{measure} = Seq_{trigger} = Seq_{reset} = INC \ [mem]$
Example 2: \( Seq_{measure} = Seq_{trigger} = \text{INC [mem]} \);  
\( Seq_{reset} = \text{CLFLUSH [mem]} \)
Example 2: $Seq_{\text{measure}} = Seq_{\text{trigger}} = \text{INC \ [mem]}$;
$Seq_{\text{reset}} = \text{CLFLUSH \ [mem]}$
Testing A Sequence Triple

Example 2: $Seq_{measure} = Seq_{trigger} = INC \ [mem]$;
$Seq_{reset} = CLFLUSH \ [mem]$
Example 2: $\text{Seq}_{\text{measure}} = \text{Seq}_{\text{trigger}} = \text{INC} \ [\text{mem}]$;
$\text{Seq}_{\text{reset}} = \text{CLFLUSH} \ [\text{mem}]$

- Cold path $S0$
- Hot path $S1$
Recap: Measuring Time

start = Ⓐ

x = y + 1

end = Ⓛ

Δ = end - start
Recap: Measuring Time

\[
\begin{align*}
\text{start} & = \emptyset \\
x & = y + 1 \\
\text{end} & = \emptyset \\
\Delta & = \text{end} - \text{start}
\end{align*}
\]

1. run: \( \Delta = 302 \rightarrow \text{cache miss} \)
2. run: \( \Delta = 54 \rightarrow \text{cache hit} \)
Recap: Measuring Time

clflush [y]
start = ⊙
x = y + 1
end = ⊙

Δ = end - start

1. run: Δ = 302 → cache miss
2. run: Δ = 302 → cache miss
Recap: Measuring Time

\[ \text{clflush } [y] \]
\[ \text{start} = \varnothing \]
\[ x = y + 1 \]
\[ \text{end} = \varnothing \]
\[ \Delta = \text{end} - \text{start} \]

1. run: \( \Delta = 302 \) → cache miss
2. run: \( \Delta = 302 \) → cache miss

Determinism!

No randomness or non-determinism – just behavior we did not understand
Flush+Reload

Attacker

**flush**

**access**

Shared Memory

Victim

**access**
Flush+Reload

Attacker
flush
access

Shared Memory

cached

Victim
access
Flush+Reload

Attacker

**flush**

**access**

Shared Memory

Victim

**access**
Flush+Reload

Attacker

**flush**

**access**

Victim

**access**

Shared Memory
Flush+Reload

Attacker

flush
access

Shared Memory

Victim

access
Flush+Reload

Attacker

flush
access

Shared Memory

Victim

access
Flush+Reload

Attacker

flush
access

Shared Memory

Victim

access
Flush+Reload

Attacker

 flush

access

Victim

access

Shared Memory vs
Victim accessed (fast)
Victim did not access (slow)
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

Result = C
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

\[
egin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & \ldots \\
\end{array}
\]

\[
\text{Result} = \text{Result} \times \text{Result} \times C
\]

square

multiply
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

\[
\begin{array}{ccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & \ldots
\end{array}
\]

\[
\text{Result} = \underbrace{\text{Result} \times \text{Result}}_{\text{square}}
\]
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

Result = Result \times Result

\text{square}
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

1 1 0 0 1 1 0 \ldots

Result = Result \times Result \times C

\text{square} \quad \text{multiply}
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

1 1 0 0 1 1 0 \ldots

\[
\text{Result} = \text{Result} \times \text{Result} \times C
\]

square multiply
Flush+Reload on Square-and-Multiply

\[ M = C^d \mod n \]

Result = \( \text{Result} \times \text{Result} \)

square
Motivation

Finding side channels is a complex and time-consuming process.
MEASURE

ALL THE THINGS
Osiris – Fuzzing x86 CPUs for Side Channels

Offline

ISA  Instructions
Osiris – Fuzzing x86 CPUs for Side Channels

- Offline
- Generation
- Timing Measurement
- Randomized Execution
- Leaking Triples
- Clustering

- Fuzzed on 5 different CPUs
  - AMD and Intel
Osiris – Fuzzing x86 CPUs for Side Channels

- ISA
- Instructions
- Triple Generation
- Timing Measurement

1. **Generation**
2. **Execution**

- Offline

• Fuzzed on 5 different CPUs
  - AMD and Intel
Osiris – Fuzzing x86 CPUs for Side Channels

[Diagram showing the process]

1. Offline
   - ISA
   - Instructions

2. Generation
   - Triple Generation
   - Timing Measurement
   - Randomized Execution

3. Execution
   - Leaking Triples

4. Confirmation

- Fuzzed on 5 different CPUs
- AMD and Intel
Osiris – Fuzzing x86 CPUs for Side Channels

- Fuzzed on 5 different CPUs
- AMD and Intel
Osiris – Fuzzing x86 CPUs for Side Channels

1. Generation
2. Execution
3. Confirmation
4. Clustering

- Fuzzed on 5 different CPUs
- AMD and Intel

ISA

Instructions

Triple Generation

Timing Measurement

Randomized Execution

Leaking Triples

Clustering

Report
Osiris – Fuzzing x86 CPUs for Side Channels

- Fuzzed on 5 different CPUs
- AMD and Intel
Osiris Results

- ~4 days per CPU
- 2 side channels rediscovered
- 4 new side channels
- 2 new attacks
Cross-VM Interference

- Random-number generator RDRAND
- VM in the cloud (e.g., AWS) sees usage of other VMs
  \[\rightarrow\] Breaks VM isolation
RDRAND Covert Channel - Properties

- AMD and Intel
- VM and native
- 1000 bit/s
- No memory
- No detection
- No mitigation
KASLR Break

Kernel

CLFLUSH

MOVNT

Load

Cached?

exec.

yes
KASLR Break

CLFLUSH

MOVNT

Load

Cached?

Kernel

no

abort

Cached?
KASLR Break

Kernel

CLFLUSH

MOVNT

Load

Cached?

exec.

yes
HEY! GET BACK TO WORK!

MEASURING!

OH. CARRY ON.
Continue Measuring

\[ x = y + 1 \]

\[ \text{start} = \emptyset \]

\[ x = y + 1 \]

\[ \text{end} = \emptyset \]

\[ \Delta = \text{end} - \text{start} \]
Continue Measuring

\[ x = y + 1 \]

\[ \text{start} = \emptyset \]
\[ x = y + 1 \]
\[ \text{end} = \emptyset \]
\[ \Delta = \text{end} - \text{start} \]

1. run: \( \Delta = 54 \rightarrow \text{cache hit} \)
2. run: \( \Delta = 54 \rightarrow \text{cache hit} \)
Continue Measuring

<crash>

\[ x = y + 1 \] (never executed architecturally)

<restart>

\[
\begin{align*}
\text{start} & = \emptyset \\
x & = y + 1 \\
\text{end} & = \emptyset \\
\Delta & = \text{end} - \text{start}
\end{align*}
\]

1. run:

2. run:
Continue Measuring

\(<\text{crash}>\)

\[ x = y + 1 \] (never executed architecturally)

\(<\text{restart}>\)

\[
\begin{align*}
\text{start} & = \emptyset \\
\quad x & = y + 1 \\
\text{end} & = \emptyset \\
\Delta & = \text{end} - \text{start}
\end{align*}
\]

1. run: \( \Delta = 54 \rightarrow \text{cache hit} \)

2. run: \( \Delta = 54 \rightarrow \text{cache hit} \)

(Im)possible

Microarchitecture can do things impossible for the architecture
Meltdown

User Memory

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D E</td>
</tr>
<tr>
<td>F</td>
<td>G H</td>
</tr>
<tr>
<td>I</td>
<td>J K</td>
</tr>
<tr>
<td>L M N</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>P Q</td>
</tr>
<tr>
<td>R S T</td>
<td></td>
</tr>
<tr>
<td>U V W</td>
<td></td>
</tr>
<tr>
<td>X Y Z</td>
<td></td>
</tr>
</tbody>
</table>

```c
char value = kernel[0]
```
### User Memory

|   | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

```c
char value = kernel[0]
```

Page fault (Exception)
Meltdown

User Memory

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>Q</td>
<td>R</td>
</tr>
<tr>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>V</td>
</tr>
<tr>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

```
char value = kernel[0]
```

**Page fault (Exception)**

```
mem[value]
```

**Out of order**
User Memory

```
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

```cpp
char value = kernel[0]
mem[value]
```

Page fault (Exception)
Out of order
### Meltdown Experiment

<table>
<thead>
<tr>
<th>L1 Cacheline</th>
<th>XXXXXXXXXXXXXXXX</th>
<th>...</th>
<th>Kernel Memory</th>
</tr>
</thead>
</table>

...
Meltdown Experiment

L1 Cacheline

··· XXXXXXXXXXXX ···

Kernel Memory

Leak (Meltdown)

X
Meltdown Experiment

L1 Cacheline

· · · X· · ·

 leak (Meltdown)

X X

Kernel Memory
Meltdown Experiment

L1 Cacheline

XXX...

Leak (Meltdown)

X X X

Kernel Memory
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

 Kernel Memory

Leak (Meltdown)

X X X X X X
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

XXXXXXXXXXXXXXXX

Kernel Memory

Leak (Meltdown)

X X X X X X X X X
Meltdown Experiment

L1 Cacheline

XXX

Kernel Memory

Leak (Meltdown)

XX XP
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Leak (Meltdown)

Kernel Memory

X X X X X X X X X X X P X X X X
Meltdown Experiment

L1 Cacheline

 Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

X X X X X X X P X X X X X P X

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Meltdown Experiment

L1 Cacheline

Kernel Memory

Leak (Meltdown)
Complex Load Situations

Execution Engine
- Reorder buffer
  - Scheduler
    - Execution Units
      - ALU, AES, ...
      - ALU, FMA, ...
      - ALU, Vect, ...
      - ALU, Branch
      - Load data
      - Store data
    - AGU

Core Memory
- Load Buffer
- Store Buffer
- L1 Data Cache
- DTLB
- LFB

CDB
Complex Load Situations

... mov al, byte [rcx] ...
Complex Load Situations

...mov al, byte [rcx]...

Diagram:
- Complex Load Situations
- Execution Engine
  - Reorder buffer
  - Scheduler
  - Execution Units: ALU, AES, ALU, FMA, ALU, Vect, ALU, Branch
  - Load data
  - Store data
  - AGU
- Core Memory
  - Load Buffer
  - Store Buffer
  - L1 Data Cache
  - DTLB
  - LFB
- Memory Model: 
  - CDB
  - Load Buffer
  - Store Buffer
Complex Load Situations

- **Execution Engine**
  - Reorder buffer
  - Scheduler
  - Execution Units
    - ALU, AES, ...
    - ALU, FMA, ...
    - ALU, Vect, ...
    - ALU, Branch
  - Load data
  - Store data
  - AGU

- **Core Memory**
  - Load Buffer
  - Store Buffer
  - L1 Data Cache
  - DTLB
  - LFB

- **CDB**

- **mov al, byte [rcx]**
Complex Load Situations

... mov al, byte [rcx] ...

Diagram showing the execution engine withScheduler, Execution Units (ALU, AES, ALU, FMA, ALU, Vect, ALU, Branch), Load load, Store data, AGU, Load Buffer, Store Buffer, CDB, Reorder buffer, Execution Engine, Core Memory, L1 Data Cache, DTLB, LFB.
Complex Load Situations

```
... mov al, byte [rcx] ...
```
Complex Load Situations

... mov al, byte [rcx] ...

...
Complex Load Situations

... mov al, byte [rcx] ...

not used for L1/SB/LFB
Complex Load Situations

- Execution Engine
  - Reorder buffer
  - Scheduler
  - Execution Units:
    - ALU, AES, ...
    - ALU, FMA, ...
    - ALU, Vect, ...
    - ALU, Branch
  - Load, Store data
- AGU

- CDB
- Core Memory:
  - Load Buffer
  - Store Buffer
  - L1 Data Cache
  - DTLB
- Load, Store data

- CDB
- #n+1 ...
- #n  ppn  vpn  offset  reg.no.
- #n-1 ...

- mov al, byte [rcx]...

Data can go to register.
There is no noise.

Noise is just someone else’s data.
Systematic Analysis

1. preface
2. trigger instruction
3. transient instructions
4. fixup
5. reconstruct

Time

architectural

transient execution

architectural
Intel Zombieload bug fix to slow data centre computers

ZombieLoad attack lets hackers steal data from Intel chips

'Zombieload' Flaw Lets Hackers Crack Almost Every Intel Chip Back to 2011. Why's It Being Downplayed?

Only New CPUs Can Truly Fix ZombieLoad and Spectre
That Escalated Quickly

How it started
That Escalated Quickly

How it started

How it’s going
I have some sort
Of idea what I am
doing
WITH GREAT INSIGHT COMES GREAT AUTOMATION
• Many **Microarchitectural Data Sampling (MDS)** attacks → ZombieLoad, RIDL, Fallout, Meltdown-UC

• Different **variants** and leakage targets

• **Complex** to reproduce and test all variations

• **Common**: require a fault or microcode assist
MDS

User Memory

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
</tr>
<tr>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>R</td>
<td>S</td>
</tr>
<tr>
<td>U</td>
<td>V</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
</tr>
</tbody>
</table>

```python
char value = faulting[0]
```
char value = faulting[0]

Fault
```markdown
# MDS

## User Memory

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>K</td>
</tr>
<tr>
<td>L</td>
<td>M</td>
<td>N</td>
</tr>
<tr>
<td>O</td>
<td>P</td>
<td>Q</td>
</tr>
<tr>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>Y</td>
<td>Z</td>
</tr>
</tbody>
</table>

```char` value = faulting[0]`  

```mem[value]`  

**Fault**  

**Out of order**
### User Memory

|   | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

```c
char value = faulting[0]
```

```c
mem[value]
```

Out of order

Fault

```
```
Memory Access Checks (simplified)

- Many possibilities for faults
Memory Access Checks (simplified)

- Many possibilities for faults

- Idea: mutation fuzzing for new variants
Transynther

P1: Synthetisation

- Meltdown
- RIDL
- Fallout
- ZombieLoad

Random Instruction
P1: Synthesisation

- Meltdown
- RIDL
- Fallout
- ZombieLoad

Mutate

Random Instruction
Transynther

P1: Synthetisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad

P2: Evaluation
- Random Instruction
- Mutate
- Potential Meltdown Code Sequence
- Execute Code
- Leakage

0
Transynther

P1: Synthetisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad

Random Instruction → Mutate
- Potential Meltdown Code Sequence

P2: Evaluation
- Execute Code
- Leakage

Send to Classification

P3: Classification

1
0
Transynther Results

- 26 hours runtime
- 100 unique leakage patterns
- 7 attacks reproduced
- 1 new vulnerability
- 1 regression
● Medusa: new variant of ZombieLoad
- **Medusa**: new variant of ZombieLoad
- Leaks from write-combining buffer, i.e., REP MOV
- Used for fast memory copy, e.g., in OpenSSL or kernel
  → Leaked RSA key while decoding in OpenSSL
Ice Lake Regression

• Ice Lake microarchitecture reported no vulnerabilities
• Transynther found a regression via a small mutation
  → Re-enabled a “mitigated” variant
• Fixed via microcode update
After all... why not?

WHY SHOULDN'T I LOOK AT ARCHITECTURAL INTERFACES?
Playing with MSRs

- Goal: disable a prefetcher for an experiment

% sudo wrmsr -a 320 0x2

% sudo wrmsr -a 420 0x2

Segmentation fault (core dumped)
% top
Segmentation fault (core dumped)
% sudo reboot
Segmentation fault (core dumped)

→ Every application crashed on startup

• No information about this MSR in the Intel manual
Playing with MSRs

- Goal: disable a prefetcher for an experiment

```
% sudo wrmsr -a 320 0x2
```

Typo: should be 420, not 320

```
% sudo wrmsr -a 420 0x2
Segmentation fault (core dumped)
```

```
% top
Segmentation fault (core dumped)
```

```
% sudo reboot
Segmentation fault (core dumped)
```

→ Every application crashed on startup

- No information about this MSR in the Intel manual
Playing with MSRs

- Goal: disable a prefetcher for an experiment

  ```
  % sudo wrmsr -a 320 0x2
  ```

- Typo: should be 420, not 320
Playing with MSRs

- Goal: disable a prefetcher for an experiment
  
  ```
  % sudo wrmsr -a 320 0x2
  ```

- Typo: should be 420, not 320
  
  ```
  % sudo wrmsr -a 420 0x2
  Segmentation fault (core dumped)
  % top
  Segmentation fault (core dumped)
  % sudo reboot
  Segmentation fault (core dumped)
  ```
Playing with MSRs

- **Goal:** disable a prefetcher for an experiment

  ```
  % sudo wrmsr -a 320 0x2
  ```

- **Typo:** should be 420, not 320

  ```
  % sudo wrmsr -a 420 0x2
  Segmentation fault (core dumped)
  % top
  Segmentation fault (core dumped)
  % sudo reboot
  Segmentation fault (core dumped)
  ```

→ Every application **crashed on startup**
Playing with MSRs

- **Goal:** disable a prefetcher for an experiment
  
  \[
  \% \text{ sudo wrmsr -a 320 0x2}
  \]

- **Typo:** should be 420, not 320
  
  \[
  \% \text{ sudo wrmsr -a 420 0x2}
  \]
  
  Segmentation fault (core dumped)
  
  \[
  \% \text{ top}
  \]
  
  Segmentation fault (core dumped)
  
  \[
  \% \text{ sudo reboot}
  \]
  
  Segmentation fault (core dumped)

→ Every application **crashed on startup**

- **No information** about this MSR in the Intel manual
Model-Specific Registers and their Hidden Secrets

- Model-Specific Registers (MSRs): CPU control registers
Model-Specific Registers and their Hidden Secrets

- Model-Specific Registers (MSRs): CPU control registers
- AMD K8: Undocumented debug mode via MSR
Model-Specific Registers and their Hidden Secrets

- Model-Specific Registers (MSRs): CPU control registers
- AMD K8: Undocumented debug mode via MSR
- VIA C3 CPU: backdoor access via undocumented MSR bit
MSRevelio – Overview
MSRevelio – Overview

- BIOS Templating
- MSR Scanning & Difference Detection
- MSR Detection

MSR Detection

Documented MSRs

Undocumented MSRs

Dynamic MSRs

Static MSRs

Analyze Correlation

Analyze Bit Effects
MSRevelio – Overview

- BIOS Templating
- MSR Scanning & Difference Detection
- Documented MSRs
- Undocumented MSRs

MSR Detection

- Dynamic MSRs
- Static MSRs

Analyze

Correlation

Bit Effects
MSRevelio – Overview

- BIOS Templating
- MSR Scanning & Difference Detection
- Documented MSRs
- Dynamic MSRs
- Undocumented MSRs
- Static MSRs

MSR Detection
MSRevelio – Overview

BIOS Templating

MSR Scanning & Difference Detection

Documented MSRs

Dynamic MSRs

Analyze Correlation

Undocumented MSRs

Static MSRs

Analyze Bit Effects

MSR Detection

MSR Classification
MSRevelio – Results Overview

5890 undocumented MSRs
MSRevelio – Results Overview

5890 undocumented MSRs

3 MSRs as attack mitigation
MSRevelio – Results Overview

5890 undocumented MSRs

3 MSRs as attack mitigation

1 MSR allows TOCTOU vulnerability
MSRevelio – Results Overview

5890 undocumented MSRs

3 MSRs as attack mitigation

1 MSR allows TOCTOU vulnerability

New MSRs hint towards vulnerabilities
Findings – Undisclosed Attacks

• MSRs often introduce vulnerability fixes
Findings – Undisclosed Attacks

- MSRs often introduce vulnerability fixes
- MSRs exist before public disclosure
  → Useful for 1-Day Exploits
Findings – Attack Mitigation

- Mitigate prefetch side-channel attacks
Findings – Attack Mitigation

- Mitigate prefetch side-channel attacks
- Reduce CrossTalk leakage
Findings – Attack Mitigation

- Mitigate prefetch side-channel attacks
- Reduce CrossTalk leakage
- Reduce Medusa leakage
• Searching for unknown behavior is hard
MSRevelio – Lessons Learned

- Searching for unknown behavior is hard
- We can automate the search for undocumented MSR behavior
• Searching for unknown behavior is hard
• We can automate the search for undocumented MSR behavior
• Automation allows tracing changes between releases
• Searching for unknown behavior is hard
• We can automate the search for undocumented MSR behavior
• Automation allows tracing changes between releases
• Architectural interfaces not fully explored
Loads are Complex

mov al, byte [rcx]

L1 Instruction Cache

Instruction Fetch & PreDecode

Instruction Queue

4-Way Decode

Branch Predictor

μOP Cache

Allocation Queue

Reorder buffer

Scheduler

Execution Units

ALU, AES, . . .

ALU, FMA, . . .

ALU, Vect, . . .

ALU, Branch

Load data

Store data

CDB

Memory Subsystem

Load Buffer

Store Buffer

L1 Data Cache

DTLB

L2 Cache

L3 Cache

DRAM

51
Loads are Complex

mov al, byte [rcx]

Frontend
- Branch Predictor
- μOP Cache
- Instruction Fetch & PreDecode
- 4-Way Decode
- Allocation Queue

Execution Engine
- Reorder buffer
- Scheduler
- Execution Units
  - ALU, AES, ...
  - ALU, FMA, ...
  - ALU, Vect, ...
  - ALU, Branch
  - Load data
  - Store data
  - ALU

Memory Subsystem
- Load Buffer
- Store Buffer
- L1 Data Cache
- DTLB
- STLB
- L2 Cache
- L3 Cache
- DRAM
Loads are Complex
Loads are Complex

mov al, byte [rcx]

L1 Instruction Cache

Branch Predictor

Instruction Fetch & PreDecode

Instruction Queue

4-Way Decode

Allocation Queue

Reorder buffer

Scheduler

Execution Units

ALU, AES, . . .

Execution Engine

ALU, FMA, . . .

ALU, Vect, . . .

ALU, Branch

Load data

Store data

ALU

CDB

Memory Subsystem

Load Buffer

Store Buffer

L1 Data Cache

DTLB

STLB

L2 Cache

L3 Cache

DRAM
Loads are Complex

mov al, byte [rcx]
Loads are Complex

mov al, byte [rcx]
Loads are Complex

mov al, byte [rcx]
Loads are Complex

mov al, byte [rcx]

L1 Instruction Cache

Branch Predictor

Instruction Fetch & PreDecode

Instruction Queue

4-Way Decode

Allocation Queue

Reorder buffer

Scheduler

Execution Units

Execution Engine

Memory Subsystem

Load Buffer

Store Buffer

L1 Data Cache

DTLB

STLB

L2 Cache

L3 Cache

DRAM

CDB

ALU, AES, . . .

ALU, FMA, . . .

ALU, Vect, . . .

ALU, Branch

Load data

Load data

Store data

AGU
Loads are Complex

mov al, byte [rcx]
Loads are Complex

mov al, byte [rcx]
Loads are Complex

mov al, byte [rcx]
Bunch of Stuff Mapped

00001000-0009efff : System RAM
0009f000-000ffffff : Reserved
  000a0000-000bffff : PCI Bus 0000:00
00100000-412f6017 : System RAM
45c0a000-45cba000 : ACPI Non-volatile Storage
47f00000-64bfffff : System RAM
  61000000-64bfffff : Graphics Stolen Memory
64c00000-bfffffff : PCI Bus 0000:00
  66000000-721ffffff : PCI Bus 0000:01
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
ff000000-fffffffff : Reserved
  ff000000-fffffffff : pnp 00:04
100000000-49b3fffff : System RAM
  1ad600000-1ae400df0 : Kernel code
  1ae400df1-1af25687f : Kernel data
  1af52a000-1af9fffff : Kernel bss
49b400000-49bfffff : RAM buffer
[...]
Scanning Memory

Physical Memory

...........

repeat

read: SECRET
write: SECRET
flush: SECRET
Scanning Memory

Physical Memory

........

repeat

read: SECRET
write: SECRET
flush: SECRET
Scanning Memory

Physical Memory

........

read: SECRET
write: SECRET
flush: SECRET

repeat
Scanning Memory

Physical Memory

repeat

read: SECRET
write: SECRET
flush: SECRET
Scanning Memory

Physical Memory

read: SECRET
write: SECRET
flush: SECRET

repeat
Scanning Memory

Physical Memory

read: SECRET
write: SECRET
flush: SECRET

repeat
Scanning Memory

Physical Memory

read: SECRET
write: SECRET
flush: SECRET

repeat
Scanning Memory

Physical Memory

repeat

read: SECRET
write: SECRET
flush: SECRET
Leaking Address

00010000-0009efff : System RAM
0009f000-000ffffff : Reserved
  000a0000-000bffff : PCI Bus 0000:00
00100000-412f6017 : System RAM
45c90000-45c9ffffff : ACPI Non-volatile Storage
47f00000-64bffffff : Reserved
  61000000-64bffffff : Graphics Stolen Memory
64c00000-bfffffff : PCI Bus 0000:00
  66000000-721ffffff : PCI Bus 0000:01
fee00000-fee00fff : Local APIC
  fee00000-fee00fff : Reserved
ff000000-fffffff : Reserved
  ff000000-fffffff : pnp 00:04
100000000-49b3fffff : System RAM
1ad600000-1aee400df0 : Kernel code
1ae400df1-1af25687f : Kernel data
1af52a000-1af9ffffff : Kernel bss
49b400000-49bffffff : RAM buffer
[...]
Advanced Programmable Interrupt Controller (APIC)

Generate, receive and forward interrupts in modern CPUs.

- Local APIC for each CPU
- I/O APIC towards external devices
- Exposes registers
### APIC MMIO

- **Memory-mapped** APIC registers

<table>
<thead>
<tr>
<th>Address</th>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFEE00000</td>
<td>Timer</td>
<td>0x00</td>
</tr>
<tr>
<td>0xFEE00010</td>
<td>Thermal</td>
<td>0x10</td>
</tr>
<tr>
<td>0xFEE00020</td>
<td>ICR bits 0-31</td>
<td>0x20</td>
</tr>
<tr>
<td>0xFEE00030</td>
<td>ICR bits 32-63</td>
<td>0x30</td>
</tr>
</tbody>
</table>

- Memory-mapped APIC registers are controlled by MSR IA32_APIC_BASE (default 0xFEE00000).
- Mapped as 32-bit values, aligned to 16 bytes.
- Should not be accessed at bytes 4 through 15.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
APIC MMIO

- **Memory-mapped** APIC registers
  - Controlled by MSR IA32_APIC_BASE (default 0xFEE00000)

| 0xFEE00000: | Timer | | | | 0x00 |
|---|---|---|---|---|
| | Thermal | | | 0x10 |
| ICR bits 0-31 | | | | 0x20 |
| ICR bits 32-63 | | | | 0x30 |
| 0 | 4 | 8 | 12 |
APIC MMIO

- **Memory-mapped** APIC registers
  - Controlled by MSR IA32_APIC_BASE (default 0xFEE00000)
  - Mapped as **32bit** values, **aligned** to **16 bytes**

0xFEE00000:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td></td>
<td></td>
<td></td>
<td>0x00</td>
</tr>
<tr>
<td>Thermal</td>
<td></td>
<td></td>
<td></td>
<td>0x10</td>
</tr>
<tr>
<td>ICR bits 0-31</td>
<td></td>
<td></td>
<td></td>
<td>0x20</td>
</tr>
<tr>
<td>ICR bits 32-63</td>
<td></td>
<td></td>
<td></td>
<td>0x30</td>
</tr>
</tbody>
</table>
**APIC MMIO**

- **Memory-mapped** APIC registers
  - Controlled by MSR IA32_APIC_BASE (default 0xFEE00000)
  - Mapped as **32bit** values, **aligned** to **16 bytes**
  - **Should not** be accessed at bytes 4 through 15.

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timer</td>
<td>0x00</td>
<td>0x00</td>
</tr>
<tr>
<td>Thermal</td>
<td>0x10</td>
<td>0x00</td>
</tr>
<tr>
<td>ICR bits 0-31</td>
<td>0x20</td>
<td>0x00</td>
</tr>
<tr>
<td>ICR bits 32-63</td>
<td>0x30</td>
<td>0x00</td>
</tr>
</tbody>
</table>
Any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results, or unexpected exceptions.
Any access that touches bytes 4 through 15 of an APIC register may cause undefined behavior and must not be executed. This undefined behavior could include hangs, incorrect results, or unexpected exceptions.
Ruling out Microarchitectural Elements

Core

Thread

- Registers
- Execution Engine
- MOB
- L1
- TLB
- L2
- Superqueue
- LLC
- Memory Controller
- RAM
Ruling out Microarchitectural Elements

Core

Thread

Execution Engine

L1

TLB

L2

Superqueue

LLC

Memory Controller

RAM

Thread

Execution Engine

MOB

L1

TLB

L2

Superqueue

LLC

Memory Controller

RAM
Ruling out Microarchitectural Elements

Core

Thread

Execution Engine

MOB

Superqueue

L1

L2

LLC

Memory Controller

RAM

Thread

Registers

Registers

Thread Registers

Execution Engine

L1

MOB

Superqueue

L2

LLC

Memory Controller

RAM
Ruling out Microarchitectural Elements

Core

Thread

Registers

Execution Engine

MOB

L1

TLB

L2

Superqueue

LLC

Memory Controller

RAM

Thread

Registers
Ruling out Microarchitectural Elements
The Superqueue

- It’s the **decoupling buffer** between L2 and LLC
The Superqueue

- It’s the **decoupling buffer** between L2 and LLC
- Contains **data** passed between L2 and LLC
The Superqueue

- It’s the **decoupling buffer** between L2 and LLC
- Contains **data** passed between L2 and LLC
- Like **Line Fill Buffers** for L1 and L2
Leaking from the Superqueue

<table>
<thead>
<tr>
<th>APIC</th>
<th>EOI</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ISR</td>
<td>???</td>
</tr>
<tr>
<td></td>
<td>IRR</td>
<td>???</td>
</tr>
</tbody>
</table>

Attacker

Victim (SGX)

Superqueue
Leaking from the Superqueue

APIC

<table>
<thead>
<tr>
<th>IR</th>
<th>EOI</th>
<th>???</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISR</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>IRR</td>
<td>???</td>
<td></td>
</tr>
</tbody>
</table>

Victim (SGX)

L3 load/store

Attacker
Leaking from the Superqueue

APIC

<table>
<thead>
<tr>
<th></th>
<th>EOI</th>
<th>ISR</th>
<th>IRR</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>???</td>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

Attacker

Victim (SGX)

L3 load/store

Supercue
Leaking from the Superqueue

APIC

<table>
<thead>
<tr>
<th>ISR</th>
<th>EOI</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>???</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
</tr>
</tbody>
</table>

Victim (SGX)

L3 load/store

Attacker

request

response

Superqueue

X X X X X X X X X

IRR

X X X X X X X X X (stale data)
• First architectural CPU vulnerability
• Deterministically leak data from SGX
• Does not require hyperthreading
Software vs. Hardware Fuzzing

- All low-hanging fruit
- Approximately as sophisticated as software fuzzing in 1990
- Majority of fuzzers does not use any guidance
- More research on feedback necessary
Summary

- **Simple** models are **sufficient** to find leakage
- Dumb fuzzers find leakage **within hours**
  - New vulnerability variants
  - New side channels
  - Regression in new CPUs
  - New architectural vulnerabilities
- Prediction: **smarter fuzzers → more vulnerabilities**
### Open Source

- [https://github.com/CISPA/Osiris](https://github.com/CISPA/Osiris)
  - **USENIX’21**
  - Osiris: Automated Discovery of Microarchitectural Side Channels.

- [https://github.com/vernamlab/Medusa](https://github.com/vernamlab/Medusa)
  - **USENIX’20**
  - Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis.

- [https://github.com/IAIK/MSRevelio](https://github.com/IAIK/MSRevelio)
  - **USENIX’22**
  - Finding and Exploiting CPU Features using MSR Templating.
### Spectre, Meltdown, and LVI Variants

#### Spectre-type
- Spectre-PHT
  - Cross-address-space: PHT-CA-IP, PHT-CA-OP
  - Same-address-space: PHT-SA-IP, PHT-SA-OP
- Spectre-BTB
  - Cross-address-space: BTB-CA-IP, BTB-CA-OP
  - Same-address-space: BTB-SA-IP, BTB-SA-OP
- Spectre-RSB
  - Cross-address-space: RSB-CA-IP, RSB-CA-OP
  - Same-address-space: RSB-SA-IP, RSB-SA-OP
- Spectre-BHB
  - Cross-address-space: BHB-CA-IP, BHB-CA-OP
  - Same-address-space: BHB-SA-IP, BHB-SA-OP

#### Meltdown-type
- Meltdown-NM-REG
- Meltdown-AC
  - Cross-address-space: Meltdown-AC-LFB, Meltdown-AC-LP
  - Same-address-space: Meltdown-AC-REG
- Meltdown-DE
- Meltdown-PF
- Meltdown-US
- Meltdown-P
  - Cross-address-space: Meltdown-P-L1, Meltdown-P-LFB, Meltdown-P-SB, Meltdown-P-LP, Meltdown-P-SP
  - Same-address-space: Meltdown-P-L1, Meltdown-P-LFB, Meltdown-P-SB, Meltdown-P-LP, Meltdown-P-SP
- Meltdown-RW
- Meltdown-PK
  - Cross-address-space: Meltdown-PK-L1, Meltdown-PK-SB
  - Same-address-space: Meltdown-PK-L1, Meltdown-PK-SB
- Meltdown-XD
- Meltdown-SM-SB
- Meltdown-UD
- Meltdown-SS
- Meltdown-BR
- Meltdown-MPX
- Meltdown-BND
- Meltdown-GP
- Meltdown-CPL-REG
- Meltdown-NC-SB
- Meltdown-AVX
  - Cross-address-space: Meltdown-AVX-SB, Meltdown-AVX-LP
  - Same-address-space: Meltdown-AVX-SB, Meltdown-AVX-LP
- Meltdown-MCA
- Meltdown-AD
  - Cross-address-space: Meltdown-AD-L1, Meltdown-AD-LFB, Meltdown-AD-SB, Meltdown-AD-LP
  - Same-address-space: Meltdown-AD-L1, Meltdown-AD-LFB, Meltdown-AD-SB, Meltdown-AD-LP
- Meltdown-TAA
  - Cross-address-space: Meltdown-TAA-LFB, Meltdown-TAA-LP, Meltdown-TAA-SB, Meltdown-TAA-SP
  - Same-address-space: Meltdown-TAA-LFB, Meltdown-TAA-LP, Meltdown-TAA-SB, Meltdown-TAA-SP
- Meltdown-PRM-LFB
- Meltdown-UC-LFB
- Meltdown-snoop-L1

#### LVI-type
- LVI-NM-FPU
- LVI-PF
- LVI-US
- LVI-PPN
  - Cross-address-space: LVI-PPN-NULL, LVI-PPN-L1D
  - Same-address-space: LVI-PPN-NULL, LVI-PPN-L1D
- LVI-P
  - Cross-address-space: LVI-P-NULL, LVI-P-L1D, LVI-P-LFB, LVI-P-SB, LVI-P-LP
  - Same-address-space: LVI-P-NULL, LVI-P-L1D, LVI-P-LFB, LVI-P-SB, LVI-P-LP
- LVI-MCA
- LVI-AD
  - Cross-address-space: LVI-AD-LFB, LVI-AD-SB, LVI-AD-LP
  - Same-address-space: LVI-AD-LFB, LVI-AD-SB, LVI-AD-LP
- LVI-FP
- LVI-SCSB
Josh Walden @jmw1123 · 19. Nov.
Case of beer on its way/there later this week thanks Daniel! Thanks again for the partnership!

Daniel Gruss @lavados · 13. Nov.
Antwort an @Desertrold und @jmw1123
I'm in favor!
Thanks again Josh!

We already received the case a month ago but only found time this weekend to sit together and enjoy some!

We wish you a merry Christmas and look forward to continue working with Intel next year.

cc @cc0x1f @mlqxyz @misc0110 @tugraz_csbme #tugraz
FUZZ

ALL THE THINGS