CPU Fuzzing for Discovering Hardware-caused Information Leakage

Michael Schwarz
January 2022

CISPA Helmholtz Center for Information Security
Who am I

Michael Schwarz
Faculty © CISPA Helmholtz Center for Information Security
Research on CPU security and side channels
✉️ michael.schwarz@cispa.de
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
DEVELOPING STORY

COMPUTER CHIP FLAWS IMPACT BILLIONS OF DEVICES
Microarchitectural side channels are powerful attack techniques

- Attack cryptographic implementations
- Spy on user behavior
- Augment traditional software exploits
- Building blocks for transient-execution attacks
Side-channel Attacks

\[ M \equiv C^d \pmod{n} \]
### Side-channel Attacks

\[ M \equiv C^d \pmod{n} \]

<table>
<thead>
<tr>
<th>Description</th>
<th>Software</th>
</tr>
</thead>
</table>
| \( C^n = \begin{cases} 
  (C^2)^{\frac{n}{2}} & \text{if } n \equiv 0 \mod 2 \\
  C \cdot (C^2)^{\frac{n-1}{2}} & \text{if } n \equiv 1 \mod 2
\end{cases} \) | Execution time |
Side-channel Attacks

\[ M \equiv C^d \pmod{n} \]

<table>
<thead>
<tr>
<th>Description</th>
<th>Software</th>
<th>Hardware</th>
</tr>
</thead>
</table>
| \[ C^n = \begin{cases} 
(C^2)^{\frac{n}{2}} & \text{if } n \equiv 0 \pmod{2} \\
C \cdot (C^2)^{\frac{n-1}{2}} & \text{if } n \equiv 1 \pmod{2} 
\end{cases} \] | '1'      | '0'      |

Execution time  Power consumption

'1'  '0'
### Side-channel Attacks

\[ M \equiv C^d \pmod{n} \]

**Description**

\[
C^n = \begin{cases} 
(C^2)^{n/2} & \text{if } n \equiv 0 \pmod{2} \\
C \cdot (C^2)^{(n-1)/2} & \text{if } n \equiv 1 \pmod{2}
\end{cases}
\]

**Software**

- Execution time

**Hardware**

- Power consumption
- CPU caches

<table>
<thead>
<tr>
<th>'1'</th>
<th>'0'</th>
<th>'1'</th>
<th>'0'</th>
</tr>
</thead>
</table>
Flush+Reload

Attacker

flush
access

Shared Memory

Victim

access
Flush+Reload

Attacker

**flush**

**access**

Shared Memory

cached

cached

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

Shared Memory

Victim

access
Flush+Reload

Attacker

flush
access

Shared Memory

Victim

access
Flush+Reload

Attacker

\textit{flush}

\textit{access}

\textbf{Shared Memory}

Victim

\textit{access}
Flush+Reload

Attacker

flush
access

vs

Victim

access

Victim accessed (fast)

Victim did not access (slow)
Flush+Reload

Cache Hits

Access time [CPU cycles]

Number of accesses

$10^5$
Flush+Reload

Access time [CPU cycles]

Number of accesses

- Cache Hits
- Cache Misses
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

$$M = C^d \mod n$$

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

\[
\begin{array}{c}
\text{Result} = \text{Result} \times \text{Result} \times C
\end{array}
\]

- square
- multiply
M = \text{Result}^\text{d mod n}

\text{Result} = \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

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

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & \cdots \\
\end{array}
\]

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

- square
- multiply
Flush+Reload on Square-and-Multiply

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

Result = Result \times Result \times C

\[ \text{square} \]

\[ \text{multiply} \]
Flush+Reload on Square-and-Multiply

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

\[
\begin{array}{cccccccc}
1 & 1 & 0 & 0 & 1 & 1 & 0 & \cdots \\
\end{array}
\]

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

square
From Metadata to Data

Meta Data

Data
Transient Execution Attacks

- Transient-execution attacks evolved from side-channel attacks
- Side channel is a building block
- Leak data, not only metadata
- Meltdown, Spectre, ZombieLoad, Foreshadow, Fallout, LVI, ...
Motivation

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

• Apply bug-finding techniques from software
  → Fuzz for side channels
    • Input are code sequences
    • Detect timing differences
  • Start with random (dumb) fuzzing
Sequence Triples

S0

S1
Sequence Triples

S0 → S1

Reset Seq.
Sequence Triples

S0

Reset Seq.  Trigger Seq.  Trigger Seq.

S1

Reset Seq.
Testing A Sequence Triple

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

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

Cold path S0
Hot path S1
Testing A Sequence Triple

Example 1:

\[ \text{Seq}_{\text{measure}} = \text{Seq}_{\text{trigger}} = \text{Seq}_{\text{reset}} = \text{INC [mem]} \]

Example 2:

\[ \text{Seq}_{\text{measure}} = \text{Seq}_{\text{trigger}} = \text{INC [mem]} \]
\[ ; \]
\[ \text{Seq}_{\text{reset}} = \text{CLFLUSH [mem]} \]

\[ \text{INC [mem]} \]
\[ \text{CLFLUSH [mem]} \]

\[ \text{Seq}_{\text{reset}} = \text{INC [mem]} \]
\[ \text{Seq}_{\text{measure}} = \text{INC [mem]} \]

\[ \text{Seq}_{\text{reset}} = \text{INC [mem]} \]
\[ \text{Seq}_{\text{trigger}} = \text{INC [mem]} \]
\[ \text{Seq}_{\text{measure}} = \]
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]

\[\text{Cold path S0}\]

\[\text{Seq}_{\text{reset}} \quad \text{Seq}_{\text{measure}} \quad \text{Cold path S0}\]
Testing A Sequence Triple

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

Example 2:
\[
\begin{align*}
\text{Seq}_{\text{reset}} &= \text{INC} [\text{mem}] \\
\text{Seq}_{\text{trigger}} &= \text{INC} [\text{mem}] ; \\
\text{Seq}_{\text{reset}} &= \text{CLFLUSH} [\text{mem}] \\
&\quad \text{INC} [\text{mem}] \\
&\quad \text{CLFLUSH} [\text{mem}] \\
\end{align*}
\]

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:
\[
\text{Seq}_{\text{measure}} = \text{Seq}_{\text{trigger}} = \text{Seq}_{\text{reset}} = \text{INC} \left[ \text{mem} \right]
\]

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

Cold path S0

Hot path S1
Testing A Sequence Triple

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

Example 2:
- \( Seq_{\text{measure}} = \) \( Seq_{\text{trigger}} = \) \( \text{INC} \ [\text{mem}] \);
- \( Seq_{\text{reset}} = \) \( \text{CLFLUSH} \ [\text{mem}] \) \( \text{INC} \ [\text{mem}] \) \( \text{CLFLUSH} \ [\text{mem}] \)

Cold path S0

Hot path S1
Testing A Sequence Triple

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

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

INC [mem] \hspace{1cm} INC [mem]

Seq\_reset \hspace{2cm} Seq\_measure

Cold path S0

\[\begin{array}{c}
\text{Seq}_{\text{reset}} \\
\text{Seq}_{\text{trigger}} \\
\text{Seq}_{\text{measure}}
\end{array}\]
Testing A Sequence Triple

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

Example 2:

$Seq_{reset} = \text{INC [mem]}$

$Seq_{trigger} = \text{INC [mem]}$

$Seq_{measure} = \text{CLFLUSH [mem]}$

Cold path S0

Hot path S1
Example 1: $Seq_{measure} = Seq_{trigger} = Seq_{reset} = INC \ [mem]$
Example 1: $Seq_{\text{measure}} = Seq_{\text{trigger}} = Seq_{\text{reset}} = \text{INC [mem]}$

Cold path S0

Hot path S1
Testing A Sequence Triple

Example 2: $Seq_{measure} = Seq_{trigger} = INC\ [mem]$;

$Seq_{reset} = CLFLUSH\ [mem]$

Cold path S0

Hot path S1
Testing A Sequence Triple

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

$Seq_{\text{reset}} = \text{CLFLUSH \ [mem]}$
Example 2: \( Seq_{measure} = Seq_{trigger} = \text{INC [mem]}; \)
\( Seq_{reset} = \text{CLFLUSH [mem]} \)
Osiris – Fuzzing x86 CPUs for Side Channels

Offline

ISA Instructions
Osiris – Fuzzing x86 CPUs for Side Channels

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

- ISA
- Instructions
- Triple Generation

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

Offline

1. Generation

2. Execution

ISA

Instructions

Triple Generation

Timing Measurement

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

- Fuzzed on 5 different CPUs
- AMD and Intel

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

ISA → XML

Δ XML → Δ XML

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA

Δ ISA → Δ ISA
Osiris – Fuzzing x86 CPUs for Side Channels

Offline

1. Generation

2. Execution

3. Confirmation

4. Clustering

ISA

Instructions

Triple Generation

Timing Measurement

Randomized Execution

Leaking Triples

Clustering

Fuzzed on 5 different CPUs

AMD and Intel
Osiris – Fuzzing x86 CPUs for Side Channels

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

- Fuzzed on 5 different CPUs
- AMD and Intel
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
RDRAND Covert Channel

- RDRAND cross-core interference

→ Cross-core cross-VM covert channel

- Tested on the AWS cloud
RDRAND Covert Channel - Properties

- AMD and Intel
- VM and native
- 1000 bit/s
- No memory
- No detection
- No mitigation
MOVNT Side Channel

- MOVNT can replace CLFLUSH
- Flushes data from all cache levels
MOVNT Side Channel - Properties

- Faster reload
- Stealthy
- Not prevented by any cache design
MOVNT-based Meltdown

- Faster reload time → more leakage per transient window
- Previously: max. 3 bytes at once
- Meltdown PoC with MOVNT: 7.83 bytes at once
MOVNT KASLR Break

Kernel

CLFLUSH

MOVNT

Load

hit
MOVNT KASLR Break

Kernel

CLFLUSH

MOVNT

Load

miss
MOVNT KASLR Break

Kernel

CLFLUSH

MOVNT

Load

hit
Transient Execution Attacks

- Improves implementation of transient execution attacks
- Can we find new transient execution attacks too?
  → Analyze them like the side channels
MDS Analysis

- 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
### 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>

```c
char value = faulting[0];
```
```c
char value = faulting[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 |
| char value = faulting[0]
| mem[value]
| K

Fault

Out of order
char value = faulting[0]

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

Random Instruction

RIDL

Fallout

ZombieLoad
Transynthemer

P1: Synthetisation

- Meltdown
- RIDL
- Fallout
- ZombieLoad

Mutate

Random Instruction
Transynther

P1: Synthetisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad

P2: Evaluation
- Random Instruction
- Mutate
- Potential Meltdown Code Sequence
Transynther

P1: Synthesisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad
- Random Instruction
- Mutate

P2: Evaluation
- Potential Meltdown Code Sequence
- Execute Code
- Leakage
Transynther

P1: Synthetisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad
- Random Instruction
- Mutate
- Potential Meltdown Code Sequence

P2: Evaluation
- Execute Code
- Leakage
- Send to Classification
- 0
- 1

P3: Classification

196
328
437
Transynther

P1: Synthetisation
- Meltdown
- RIDL
- Fallout
- ZombieLoad
- Random Instruction
- Mutate
- Potential Meltdown Code Sequence

P2: Evaluation
- Execute Code
- Leakage
- Send to Classification

P3: Classification
- Performance Counters
- Evaluate
- Manual Analysis

Potential Meltdown Code Sequence

1

0
Buffer Grooming

- Fill microarchitectural buffers with known values
  → Rely on eviction sequences for buffers
- Leaked value indicates origin of leakage
Transynther Results

- 26 hours runtime
- 100 unique leakage patterns
- 7 attacks reproduced
- 1 new vulnerability
- 1 regression
• Medusa: new variant of ZombieLoad
Medusa

- **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
Small Specialized Fuzzers

- Only cover **small field** of possible vulnerabilities
Small Specialized Fuzzers

- Only cover small field of possible vulnerabilities
- Fuzzers are still simple
Small Specialized Fuzzers

- Only cover **small field** of possible vulnerabilities
- Fuzzers are still **simple**
  - Narrow scope
  - No complex sequences
  - No guidance
Small Specialized Fuzzers

- Only cover small field of possible vulnerabilities
- Fuzzers are still simple
  - Narrow scope
  - No complex sequences
  - No guidance
- Very specialized fuzzers
Some other specialized CPU fuzzers

- **Sandsifter**: undocumented x86 instructions
- **ABSynthe**: same-core contention side channels
- **FastSpec**: Spectre variants
- **CrossTalk**: cross-core transient execution attacks
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
• Simple models are sufficient to find leakage
• Dumb fuzzers find leakage within hours
  • New vulnerability variants
  • New side channels
  • Regression in new CPUs
• Prediction: smarter fuzzers $\rightarrow$ more vulnerabilities
Open Source

https://github.com/CISPA/Osiris

(use) USENIX’21
Osiris: Automated Discovery of Microarchitectural Side Channels.

https://github.com/vernamlab/Medusa

(use) USENIX’20
Daniel Moghimi, Moritz Lipp, Berk Sunar, Michael Schwarz.
Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis.
FUZZ

ALL THE THINGS
CPU Fuzzing for Discovering Hardware-caused Information Leakage

Michael Schwarz
January 2022

CISPA Helmholtz Center for Information Security
References


