Transient Side Channels

Mengjia Yan
Fall 2020

Based on slides from Christopher W. Fletcher
Reminder

- 1\textsuperscript{st} paper review due midnight on 09/27 (before the next lecture)

- You will receive an invitation from HotCRP

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Presenter</th>
<th>Paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>9/28 (Mon)</td>
<td>Hardware to Enforce Non-interference</td>
<td>Mengjia</td>
<td>Tiwari et al. Complete information flow tracking from the gates up. ASPLOS. 2009.</td>
</tr>
</tbody>
</table>
Micro-architecture Side Channels

Victim

A Channel
(a micro-architecture structure)

{Transient, Non-transient} × {Cache, DRAM, TLB, NoC, etc.}

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
Recap: 5-stage Pipeline

I-Fetch (IF)  Decode, Reg. (ID)  Fetch (EX)  Memory (MA)  Write-Back (WB)

addr  rdata  Inst. Memory

0x4  Addr

PC  IR

ALU

Imm Ext

we rs1 rs2 rd1 we
rs
rd2  wdata

GPRs

we

rd1

GPRs

6.888 L6-Transient Side Channels
5-stage Pipeline

- In-order execution:
  - Execute instructions according to the program order

```
t0  t1  t2  t3  t4  t5  t6  t7  . . .
IF₁  ID₁  EX₁  MA₁  WB₁  IF₂  ID₂  EX₂  MA₂  WB₂
IF₂  ID₂  EX₂  MA₂  WB₂  IF₃  ID₃  EX₃  MA₃  WB₃
IF₃  ID₃  EX₃  MA₃  WB₃  IF₄  ID₄  EX₄  MA₄  WB₄
IF₄  ID₄  EX₄  MA₄  WB₄  IF₅  ID₅  EX₅  MA₅  WB₅
```

6.888 L6-Transient Side Channels
Data Hazard and Control Hazard

\[ time \quad t0 \quad t1 \quad t2 \quad t3 \quad t4 \quad t5 \quad t6 \quad t7 \quad \ldots \]

Loop: ......

\[ \text{LD}(R1, 0, R2) \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \]

\[ \text{ADD}(R2, 10, R3) \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \]

\[ \text{BNE}(R3, \text{Loop}) \quad \text{IF}_3 \quad \text{ID}_3 \quad \text{EX}_3 \quad \text{MA}_3 \quad \text{WB}_3 \]

......
Resolving Hazards

• Stall or Bypass

\[
\text{time} \quad t_0 \quad t_1 \quad t_2 \quad t_3 \quad t_4 \quad t_5 \quad t_6 \quad t_7 \quad \ldots
\]

Loop: \ldots

LD(R1, 1, R2) \quad IF_1 \quad ID_1 \quad EX_1 \quad MA_1 \quad WB_1

ADD(R2, 10, R3) \quad IF_2 \quad ID_2 \quad EX_2 \quad MA_2 \quad WB_2

BNE(R3, Loop) \quad IF_3 \quad ID_3 \quad EX_3 \quad MA_3 \quad WB_3

\ldots

• Speculation (e.g., branch predictor)
  • Guess a value and continue executing anyway
  • When actual value is available, two cases
    • Guessed correctly \( \rightarrow \) do nothing
    • Guessed incorrectly \( \rightarrow \) restart with correct value (roll back)
Branch Predictor

• Predict Taken/Not taken
  • Not taken: PC+4
  • Taken: need to know target address

• Predict target address
  • Branch target buffer (BTB)
  • Map <current PC, target PC>

• Use history information to setup the predictor
Complex In-order Pipeline

- Need complex bypass/stall/kill paths
- In real systems, EX/MA can take multiple cycles
Out-of-order Execution

• When the pipeline is stalled, find something else to do
• When we do out-of-order execution, we are speculating that previous instructions do not cause exception
• If instruction $n$ is speculative instruction, instruction $n+i$ is also speculative

### Example

<table>
<thead>
<tr>
<th>time</th>
<th>t0</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
<th>t5</th>
<th>t6</th>
<th>t7</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD(R1, 1, R2)</td>
<td>IF₁</td>
<td>ID₁</td>
<td>EX₁</td>
<td>MA₁</td>
<td>MA₁</td>
<td>MA₁</td>
<td>MA₁</td>
<td>WB₁</td>
</tr>
<tr>
<td>ADD(R₃, 10, R4)</td>
<td>IF₂</td>
<td>ID₂</td>
<td>EX₂</td>
<td>MA₂</td>
<td>MA₂</td>
<td>MA₂</td>
<td>MA₂</td>
<td>WB₂</td>
</tr>
<tr>
<td>SUB(R₄, 10, R5)</td>
<td>IF₃</td>
<td>ID₃</td>
<td>EX₃</td>
<td>MA₃</td>
<td>MA₃</td>
<td>MA₃</td>
<td>MA₃</td>
<td>WB₃</td>
</tr>
</tbody>
</table>

6.888 L6-Transient Side Channels
Speculative & Out-of-Order Execution

- **Fetch Decode & Rename**
- **Branch Prediction**
- **Branch Resolution**
- **Out-of-Order**
- **In-Order**
- **Commit (head of ROB)**
- **Update predictors**
- **Physical Reg. File**
  - ALU
  - MEM
  - FALU
  - ……

Dispatch logic:
Detect data dependency,
issue instructions to execute
Terminology

A **speculative** instruction may squash.
- When executed, can change uArch state

A **Transient** instruction *will* squash, i.e., will not commit.

A **Non-Transient** instruction will not squash, i.e., will eventually retire.

That is, **transient instructions** are unreachable on a non-speculative microarchitecture.
General Attack Schema

- The difference between transient and non-transient side channels
  - Whether the secret access or transmitter execution is transient

Victim

Access secret transmit (secret)

Attacker

recv()
Meltdown & Spectre
In x86, a process’s virtual address space includes kernel pages, but kernel pages are only accessible in kernel mode.

- For performance purpose
  - Avoids switching page tables on context switches

What will happen if accessing kernel addresses in user mode?

- Protection fault
Meltdown

• Problem: Speculative instructions can change uArch state, e.g., cache

• Attack procedure
1. Setup: Attacker allocates probe_array, with 256 cache lines. Flushes all its cache lines
2. Transmit: Attacker executes

```
......
Ld1: uint8_t byte = *kernel_address;
Ld2: unit8_t dummy = probe_array[byte*64];
```

3. Receive: After handling protection fault, attacker performs cache side channel attack to figure out which line of probe_array is accessed \(\rightarrow\) recovers byte

Exception handling is deferred when the instruction reaches the head of ROB.
Meltdown Type Attacks

• Can be used to read arbitrary memory
•Leaks across privilege levels
  • OS $\leftrightarrow$ Application
  • SGX $\leftrightarrow$ Application (e.g., Foreshadow)
  • Etc

• Mitigations:
  • Stall speculation
  • Register poisoning
• We generally consider it as a design bug
Spectre Variant 1 – Exploit Branch Condition

• Consider the following kernel code, e.g., in a system call:

Br: if (x < size_array1) {
    secret = array1[x]*64
Ld1: y = array2[secret]
}

Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; \&array1[x] maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of array2 was fetched

Always malicious?
No. It may be a benign misprediction. We do not consider Spectre as a bug.
Spectre Variant 2 – Exploit Branch Target

• Most BTBs store partial tags and targets...
  • <last n bits of current PC, target PC>

Br: if (...) {
  ...
}

Ld1: secret = array1[x]*4096
Ld2: y = array2[secret]

Train BTB properly → Execute arbitrary gadgets speculatively

BTB predicts ... Ld1, Ld2
General Attack Schema

- Traditional (non-transient) attacks
  - Data-dependent program behavior

- Transient attacks
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths

Victim

Access `secret`

transmit `(secret)`

Channel

Attacker

recv()

“Easy” to fix

Hard to fix

6.888 L6-Transient Side Channels
Takeaways

Transient execution attacks *use* (not “are”) side/covert channels.

“Spectre” (wrong-path execution) is **fundamental**.
Speculation/prediction is not perfect.

“Meltdown” (deferred exceptions) is **not fundamental**.
Transient v.s. Non-transient
### Classification

**Access secret**  \( \rightarrow \) **transmit (secret)**  \( \rightarrow \) **recv()**

\{**Transient**, **Non-transient**\} secret \( \times \) \{**Transient**, **Non-transient**\} transmitter

<table>
<thead>
<tr>
<th>Secret accessed</th>
<th>Transmitter</th>
<th>Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-transient</td>
<td>Non-transient</td>
<td>Traditional side channels</td>
</tr>
<tr>
<td>Transient</td>
<td>Non-transient</td>
<td>Not possible on today’s machines?</td>
</tr>
<tr>
<td>Non-transient</td>
<td>Transient</td>
<td>Spectre</td>
</tr>
<tr>
<td>Transient</td>
<td>Transient</td>
<td>Spectre</td>
</tr>
</tbody>
</table>
Non-transient secret + Non-transient transmitter

What can leak?
A subset of committed architectural state, at each point in the program’s dynamic execution.

secret <- load(0x5)
secret <- secret + 1
secret -> store(0x5)

secret does not leak
(assume ‘+’ data independent)

secret <- load(0x5)
Dummy <- load(secret)

secret leaks

secret <- load(0x5)
if (false)
  Dummy <- load(secret)

secret does not leak
Non-transient secret + \{Transient, Non-transient\} transmitter

\[
\begin{align*}
\text{secret} & \leftarrow \text{load}(0x5) \\
\text{secret} & \leftarrow \text{secret} + 1 \\
\text{secret} & \rightarrow \text{store}(0x5)
\end{align*}
\]

Non-transient secret + Non-transient transmitter:

- secret does not leak
- secret leaks
- secret does not leak

Non-transient secret + Transient secret:

- secret does not leak
- secret leaks
- secret leaks (!)
Leakage Summary

\{\text{Transient}, \text{Non-transient}\} \text{ secret} \times \{\text{Transient}, \text{Non-transient}\} \text{ transmitter}

- Subset of committed arch state
- (Larger?) Subset of committed arch state. Depends on what speculation.
Next Lecture:
Tiwari et al. Complete information flow tracking from the gates up. ASPLOS. 2009.