Transient Execution Attacks

Mengjia Yan
Spring 2022

Based on slides from Christopher Fletcher
Micro-architecture Side Channels

Victim

secret-dependent execution

Attacker

A Channel
(a micro-architecture structure)

{Transient/Speculative, Non-transient/speculative} X

{Cache, DRAM, TLB, NoC, etc.}

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18

6.888 L5 - Transient Execution Attacks
Recap: 5-stage Pipeline

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

Reg.

Addr.

Inst.

Memory

0x4

Add

we

rs1

rs2

rd1

ws

wd

rd2

GPRs

Addr.

IR

ALU

Imm

Ext

we

rdata

Data

Memory

wdata

PC

IR

Addr.

Rdata

Inst.

Memory

6.888 L5 - Transient Execution Attacks
Recap: 5-stage Pipeline

- In-order execution:
  - Execute instructions according to the program order
  - What is the ideal instruction throughput? -- instruction per cycle (IPC)

<table>
<thead>
<tr>
<th>time</th>
<th>instruction1</th>
<th>instruction2</th>
<th>instruction3</th>
<th>instruction4</th>
<th>instruction5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0</td>
<td>IF1</td>
<td>IF2</td>
<td>IF3</td>
<td>IF4</td>
<td>IF5</td>
</tr>
<tr>
<td>t1</td>
<td>ID1</td>
<td>ID2</td>
<td>ID3</td>
<td>ID4</td>
<td></td>
</tr>
<tr>
<td>t2</td>
<td>EX1</td>
<td>EX2</td>
<td>EX3</td>
<td>EX4</td>
<td></td>
</tr>
<tr>
<td>t3</td>
<td>MA1</td>
<td>MA2</td>
<td>MA3</td>
<td>MA4</td>
<td></td>
</tr>
<tr>
<td>t4</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.888 L5 - Transient Execution Attacks
Data Hazard and Control Hazard

• Approaches to resolve Hazards:
  1. Stall
  2. Bypass
  3. Speculation
# Data Hazard and Control Hazard

- **Approaches to resolve Hazards:**
  1. Stall
  2. Bypass
  3. Speculation

<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>
<th>. . .</th>
</tr>
</thead>
</table>

**Loop:** ..... 

- $R2 \leftarrow \text{LD}(R1)$
  - IF$_1$ ID$_1$ EX$_1$ MA$_1$ WB$_1$

- $R3 \leftarrow \text{ADD}(R2, 10)$
  - IF$_2$ ID$_2$ EX$_2$ MA$_2$ WB$_2$

- BNE($R3$, Loop)
  - IF$_3$ ID$_3$ EX$_3$ MA$_3$ WB$_3$

<table>
<thead>
<tr>
<th>Next Iteration</th>
<th>.....</th>
</tr>
</thead>
</table>

| stall | IF$_4$ | ID$_4$ | EX$_4$ | MA$_4$ | WB$_4$ |
Data Hazard and Control Hazard

• Resolving Hazards:
  1. Stall
  2. Bypass
  3. Speculation

\[
\begin{array}{c|cccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 & \ldots \\
\hline 
\text{Loop:} & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\text{R2} & - & \text{LD(R1)} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & \text{WB}_1 & \ldots & \ldots \\
\text{R3} & - & \text{ADD(R2, 10)} & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & \text{MA}_2 & \text{WB}_2 & \ldots & \ldots \\
\text{BNE(R3, Loop)} & - & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\text{Next Iteration} & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots & \ldots \\
\end{array}
\]

**Question:** What if speculation is correct? What if it is incorrect?
Branch Predictor

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

- **Predict** target address for indirect branch
  - Branch target buffer (BTB)
  - Map <current PC, target PC>

- Use history information to setup the predictor
How to support floating point instructions which can take multiple cycles?
Complex In-Order Pipeline

- Naïve idea: Delay writeback so all operations have same latency to WB stage
  - Slow: penalize instructions that need fewer pipeline stages
Complex In-Order Pipeline

Example 1 (Fadd 3 cycles):
F3 <- Add(F1, F2) // enter EXE at cycle t
R3 <- Add(R1, R2) // when enter EXE?

Example 2 (Ld 1~50 cycles):
R4 <- Ld(R5) // enter MEM at cycle t
F3 <- Add(F1, F2) // when enter EXE?
Out-of-order Completion

• The idea: Make use of idle functional units when pipeline is stalled

Any problems?

Example 1 (Fadd 3 cycles):
F3 <- Add(F1, F2) // enter EXE at cycle t
R3 <- Add(R1, R2) // when enter EXE?

Example 2 (Ld 1~50 cycles):
R4 <- Ld(R5) // enter MEM at cycle t
R3 <- Add(R1, R2) // when enter EXE?
Problem of Out-of-order Completion

Consider executing a sequence of

\[ r_k \leftarrow (r_i) \text{ op } (r_j) \]

type of instructions

**Data-dependence**

\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \quad \text{Read-after-Write} \\
    r_5 & \leftarrow (r_3) \text{ op } (r_4) \quad \text{(RAW) hazard}
\end{align*}
\]

**Anti-dependence**

\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Read} \\
    r_1 & \leftarrow (r_4) \text{ op } (r_5) \quad \text{(WAR) hazard}
\end{align*}
\]

**Output-dependence**

\[
\begin{align*}
    r_3 & \leftarrow (r_1) \text{ op } (r_2) \quad \text{Write-after-Write} \\
    r_3 & \leftarrow (r_6) \text{ op } (r_7) \quad \text{(WAW) hazard}
\end{align*}
\]
Scoreboard: Detect Hazards Dynamically

• Approach: Stall issue until sure that issuing will cause no dependence problems...

• What to check before the Issue stage can dispatch an instruction?
  • Is the required function unit available?
  • Is the input data available? ⇒ RAW?
  • Is it safe to write the destination? ⇒ WAR? WAW?
  • Is there a structural conflict at the WB stage?
A Data Structure for Correct In-order Issues

<table>
<thead>
<tr>
<th>Name</th>
<th>Busy</th>
<th>Op Dest</th>
<th>Src1</th>
<th>Src2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mult2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Div</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The instruction \( i \) at the Issue stage consults this table:

- FU available? check the busy column
- RAW? search the dest column for \( i \)'s sources
- WAR? search the source columns for \( i \)'s destination
- WAW? search the dest column for \( i \)'s destination

An entry is added to the table if no hazard is detected;
An entry is removed from the table after Write-Back
Superscalar Processors in 6.823

Dispatch logic:
Detect data dependency, issue instructions to execute
Precise Exception

• Exceptions: Event that needs to be processed by the OS kernel.
  • The event is usually unexpected or rare.
  • divide by zero, page fault, etc.

![Diagram showing process and exception handler](image-url)
Handling Exceptions in OoO Processors

• Exceptions create a control-flow dependence
• Options for handling this dependence:
  • Stall
  • Bypass
  • Find something else to do
  • Speculate!

  Stall
  Bypass
  Find something else to do
  Speculate!

  No
  No
  No
  Most common approach!

• How can we handle rollback on mis-speculation?

  Delay state update until commit on speculated instructions
Handling Exceptions in OoO Processors

In-Order

Branch Prediction

Fetch → Decode & Rename

Out-of-Order

Update predictors

PC

Reorder Buffer (ROB)

In-Order

Commit (head of ROB)

Need to wait until the faulting instruction reaches the head of ROB.

Physical Reg. File

ALU, MEM, FALU, ...

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
Meltdown & Spectre
Kernel/User Pages

- 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 secret = *kernel_address;
      Ld2: unit8_t dummy = probe_array[secret*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
Meltdown Type Attacks

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

• Mitigations:
  • HW: Stall speculation; Register poisoning
  • SW: Do not let user and kernel share address space (KPTI) -> broken by several groups (talks at BlackHat)
• 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:

\[
\begin{align*}
\text{Br: } & \quad \text{if } (x < \text{size_array1}) \{ \\
\text{Ld1: } & \quad \text{secret} = \text{array1}[x] \\
\text{Ld2: } & \quad y = \text{array2}[\text{secret}\times64] \\
\} 
\end{align*}
\]

Attacker to read arbitrary memory:
1. Setup: Train branch predictor
2. Transmit: Trigger branch misprediction; \&\text{array1}[x] maps to some desired kernel address
3. Receive: Attacker probes cache to infer which line of \text{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...
  • \(<\text{last } n \text{ bits of current PC, target PC}>\)

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

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

Train BTB properly ⇒ Execute arbitrary gadgets speculatively
Mitigations?
General Attack Schema

- Traditional (non-transient) attacks
  - Data in-use

- Transient attacks: can leak data-at-rest
  - Meltdown = transient execution + deferred exception handling
  - Spectre = transient execution on wrong paths
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**.
Next Paper Discussion:

An Analysis of Speculative Type Confusion Vulnerabilities in the Wild