Transient Side Channels

Mengjia Yan
Fall 2020

Based on slides from Christopher W. Fletcher
Reminder

• 1st 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>Notes</th>
</tr>
</thead>
</table>
Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
Micro-architecture Side Channels

{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. Fetch (ID)

Execute (EX)

Memory (MA)

Write-Back (WB)

6.888 L6-Transient Side Channels
5-stage Pipeline

<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>
<tbody>
<tr>
<td>instruction1</td>
<td>IF₁</td>
<td>ID₁</td>
<td>EX₁</td>
<td>MA₁</td>
<td>WB₁</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction2</td>
<td>IF₂</td>
<td>ID₂</td>
<td>EX₂</td>
<td>MA₂</td>
<td>WB₂</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction3</td>
<td>IF₃</td>
<td>ID₃</td>
<td>EX₃</td>
<td>MA₃</td>
<td>WB₃</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction4</td>
<td>IF₄</td>
<td>ID₄</td>
<td>EX₄</td>
<td>MA₄</td>
<td>WB₄</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction5</td>
<td>IF₅</td>
<td>ID₅</td>
<td>EX₅</td>
<td>MA₅</td>
<td>WB₅</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.888 L6-Transient Side Channels
5-stage Pipeline

<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>
<tbody>
<tr>
<td>instruction1</td>
<td>IF1</td>
<td>ID1</td>
<td>EX1</td>
<td>MA1</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction2</td>
<td>IF2</td>
<td>ID2</td>
<td>EX2</td>
<td>MA2</td>
<td>WB2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction3</td>
<td>IF3</td>
<td>ID3</td>
<td>EX3</td>
<td>MA3</td>
<td>WB3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction4</td>
<td>IF4</td>
<td>ID4</td>
<td>EX4</td>
<td>MA4</td>
<td>WB4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction5</td>
<td>IF5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.888 L6-Transient Side Channels
5-stage Pipeline

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

<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></td>
<td>IF1</td>
<td>ID1</td>
<td>EX1</td>
<td>MA1</td>
<td>WB1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF2</td>
<td>ID2</td>
<td>EX2</td>
<td>MA2</td>
<td>WB2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF3</td>
<td>ID3</td>
<td>EX3</td>
<td>MA3</td>
<td>WB3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF4</td>
<td>ID4</td>
<td>EX4</td>
<td>MA4</td>
<td>WB4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF5</td>
<td>ID5</td>
<td>EX5</td>
<td>MA5</td>
<td>WB5</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Data Hazard and Control Hazard

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

Loop:  

\[
\begin{align*}
\text{LD}(R1, 0, R2) & \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{WB}_1 \\
\text{ADD}(R2, 10, R3) & \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad \text{MA}_2 \quad \text{WB}_2 \\
\text{BNE}(R3, \text{Loop}) & \quad \text{IF}_3 \quad \text{ID}_3 \quad \text{EX}_3 \quad \text{MA}_3 \quad \text{WB}_3 \\
\end{align*}
\]

......

6.888 L6-Transient Side Channels
Resolving Hazards

• Stall or Bypass

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

Loop: ......

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
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>
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
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
Out-of-order Execution

• When the pipeline is stalled, find something else to do

![Diagram of out-of-order execution](image)

<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
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

```
LD(R1, 1, R2)  IF1  ID1  EX1  MA1  MA1  MA1  MA1  WB1
ADD(R3, 10, R4)  IF2  ID2  EX2  MA2  MA2  MA2  MA2  WB2
SUB(R4, 10, R5)  IF3  ID3  EX3  MA3  MA3  MA3  MA3  WB3
......
```

6.888 L6-Transient Side Channels
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

```
LD(R1, 1, R2)
ADD(R3, 10, R4)
SUB(R4, 10, R5)
......
```

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

Branch Prediction

PC → Fetch → Decode & Rename

In-Order

Commit (head of ROB)

Update predictors

In-Order
Speculative & Out-of-Order Execution

Branch Prediction

PC → Fetch → Decode & Rename → Execute

Physical Reg. File

ALU MEM FALU

Update predictors

Commit (head of ROB)

In-Order
Speculative & Out-of-Order Execution

Branch Prediction

In-Order

Fetch → Decode & Rename

ALU MEM FALU

Physical Reg. File

Execute

Out-of-Order

Update predictors

Reorder Buffer (ROB)

In-Order

Commit (head of ROB)
Speculative & Out-of-Order Execution

- **Branch Prediction**
- **Fetch**
- **Decode & Rename**
- **In-Order**
- **Execute**
  - **Physical Reg. File**
  - **ALU**
  - **MEM**
  - **FALU**
- **Reorder Buffer (ROB)**
- **Out-of-Order**
- **Commit (head of ROB)**
- **Dispatch logic:**
  - Detect data dependency, issue instructions to execute
- **Update predictors**

Diagram showing the flow of instructions through the pipeline stages of fetch, decode, rename, execute, reorder buffer, commit, and branch prediction.
Speculative & Out-of-Order Execution

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

Dispatch logic: Detect data dependency, issue instructions to execute
Speculative & Out-of-Order Execution

In-Order

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

A *speculative* instruction may squash.

- When executed, can change uArch state
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.
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

Victim

Access secret

transmit (secret)

Channel

Attacker

recv()

6.888 L6-Transient Side Channels
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
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?
• 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
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];
```
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];
```
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];
```

6.888 L6-Transient Side Channels
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

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

ROB head

6.888 L6-Transient Side Channels
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];
```

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

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

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

```c
... 
Ld1: uint8_t byte = *kernel_address;
Ld2: uint8_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 → recovers byte.
Meltdown Type Attacks

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

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

• Mitigations:
  • Stall speculation
  • Register poisoning
Meltdown Type Attacks

• Can be used to read arbitrary memory
•Leaks across privilege levels
  • OS ↔ Application
  • SGX ↔ 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

```c
Br:  if (x < size_array1) {
    Ld1:  secret = array1[x]*64
    Ld2:  y = array2[secret]
}
```

ROB head
Spectre Variant 1 – Exploit Branch Condition

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

```c
Br: if (x < size_array1) {
Ld1: secret = array1[x]*64
Ld2: y = array2[secret]
}
```

Attacker to read arbitrary memory:
1. Setup: Train branch predictor
Spectre Variant 1 – Exploit Branch Condition

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

```c
if (x < size_array1) {
    secret = array1[x] * 64
    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
Spectre Variant 1 – Exploit Branch Condition

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

```c
Br: if (x < size_array1) {
    Ld1: secret = array1[x]*64
    Ld2: 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
Spectre Variant 1 – Exploit Branch Condition

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

```c
Br:  if (x < size_array1) {
    Ld1: secret = array1[x]*64
    Ld2: 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
Spectre Variant 1 – Exploit Branch Condition

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

```c
if (x < size_array1) {
    secret = array1[x]*64
    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?
Spectre Variant 1 – Exploit Branch Condition

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

```
Br:  if (x < size_array1) {
Ld1:  secret = array1[x]*64
Ld2:  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.

6.888 L6-Transient Side Channels
Spectre Variant 1 – Exploit Branch Condition

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

```c
Br: if (x < size_array1) {
    Ld1: secret = array1[x]*64
    Ld2: 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>

```plaintext
Br: if (...) {
  ...
}
...
Ld1: secret = array1[x]*4096
Ld2: y = array2[secret]
```

0xffff110 0xffff234
Spectre Variant 2 – Exploit Branch Target

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

\[
\begin{align*}
\text{Br: if } (...) & \{ \\
& \ldots \} \\
& \ldots \\
\text{Ld1: secret} & = \text{array1}[x] \times 4096 \\
\text{Ld2: y} & = \text{array2}[\text{secret}]
\end{align*}
\]

Branch target buffer (BTB)

BTB predicts ... Ld1, Ld2

6.888 L6-Transient Side Channels
Spectre Variant 2 – Exploit Branch Target

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

\[
\text{Br: if (\ldots) { }
\ldots 
\text{ld1: secret = array1[x] \times 4096}
\text{ld2: y = array2[secret]}
\]

Train BTB properly \textbf{→} Execute arbitrary gadgets speculatively
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
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()
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

“Easy” to fix

“Hard to fix”
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

Attacker

Access secret transmit (secret)

Channel

recv()
Takeaways

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

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

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

<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.
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)
```
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)
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
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

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

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

if (false)
  Dummy<- load(secret)

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

Non-transient secret + Non-transient transmitter:
- secret does not leak
- secret leaks
- secret does not leak

Non-transient secret + Transient secret:

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

```r
dummy <- load(secret)
if (false)
    dummy <- load(secret)
```
**Non-transient secret + \{Transient, Non-transient\} transmitter**

<table>
<thead>
<tr>
<th>secret &lt;- load(0x5)</th>
<th>secret &lt;- load(0x5)</th>
<th>secret &lt;- load(0x5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>secret &lt;- secret + 1</td>
<td>Dummy &lt;- load(secret)</td>
<td>if (false) Dummy &lt;- load(secret)</td>
</tr>
<tr>
<td>secret -&gt; store(0x5)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
Non-transient secret + \{\textbf{Transient}, Non-transient\} transmitter

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 (!)

6.888 L6-Transient Side Channels
Leakage Summary

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

- Transient + Transient
- Non-transient + Transient

6.888 L6-Transient Side Channels
Leakage Summary

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

Subset of committed arch state
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.

6.888 L6-Transient Side Channels
Leakage Summary

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

- Transient + Transient
- Non-transient + Transient
- Non-transient + Non-transient

Subset of committed arch state

All of program memory

\text{6.888 L6-Transient Side Channels}

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