Cache Timing Side-Channel Vulnerability Checking with Computation Tree Logic

Shuwen Deng, Wenjie Xiong and Jakub Szefer
Yale University

HASP
June 2, 2018
Typical set-associative cache

cache enables fast access to the data
Cache State Machine

timing latency ↔ cache hit & miss

fast access

slow access
Cache Timing Side-Channel Attacks

- For load/store instruction, time differs between hits and misses
- For flush instruction, time depends on data existence

- Attacker’s Goal: get information of the address of victim’s sensitive data by observing the timing difference

- Threat Model:
  - An attacker (A) shares the same cache with a victim (V)
  - The attacker cannot directly access the cache state machine
  - The attacker can observe the timing of the victim or itself
  - The attacker can combine timing observation with some other knowledge
    - The attacker knows some source code of the victim
    - The attacker can force victim to execute a specific function

- E.g. Flush + Reload Attack
E.g. Prime + Probe Attack

1- Attacker primes each cache set
2- Victim accesses critical data
3- Attacker probes each cache set (measure time)
E.g. Flush + Reload Attack

1- Flush each line in the cache

2- Victim accesses critical data

3- Attacker reloads critical data by running specific process (measure time)
Spectre & Meltdown Attack

- Speculative executions
  - Variant 1: Bounds Check
  - Variant 2: Branch Target Injection
  - Variant 3: Rogue Data Cache
    - Variant 3a: Rogue System Register
  - Variant 4: Speculative Store

- Timing side-channel in the cache
Spectre & Meltdown Attack

- Uses speculative executions
- Leverages timing side-channel in the cache
Contribution

Develop Cache Access Model
Three-step single-cache-block-access model construction

Analyze Timing Vulnerabilities
Exhaustive search for possible attacks based on three-step model

Use Computation Tree Logic (CTL)
Model execution paths of the processor cache focusing on side-channel attacks
We use three steps to model all possible cache side channel attacks:

- **Step 0**: The initial state of the cache block
  - $V_1/A_1$, $V_x$, $V_R/A_R$, *

- **Step 1**: Actions of victim or attacker
  - $V_1/A_1$, $V_x$, $V_R/A_R$

- **Step 2**: Interference & final observation
  - $V_1/A_1$, $V_x$, $V_R/A_R$

<table>
<thead>
<tr>
<th>condition</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1/A_1$</td>
<td>A specific known memory location.</td>
</tr>
<tr>
<td>$V_x$</td>
<td>A piece of memory containing data from a range of victim’s memory addresses is accessed.</td>
</tr>
<tr>
<td>$V_R/A_R$</td>
<td>single-cache-block access to “remove” the cache block contents</td>
</tr>
<tr>
<td>*</td>
<td>Attacker has no knowledge about memory location</td>
</tr>
</tbody>
</table>
Vulnerability Examples

• Prime + Probe Attack

Set-associative cache

1- Attacker Primes each cache set

2- Victim accesses critical data

3- Attacker Probes each cache set (measure time)

\[ EF(E(E(A_1 U V_x) U A_1)) \]
Vulnerability Examples

- **Flush + Reload Attack**

  - $A_R \rightsquigarrow V_x \rightsquigarrow A_1$
  - $V_R \rightsquigarrow V_x \rightsquigarrow A_1$

<table>
<thead>
<tr>
<th>condition</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_1/A_1$</td>
<td>A specific known memory location.</td>
</tr>
<tr>
<td>$V_x$</td>
<td>A piece of memory containing data from a range of victim’s memory addresses is accessed.</td>
</tr>
<tr>
<td>$V_R/A_R$</td>
<td>single-cache-block access to “remove” the cache block contents</td>
</tr>
</tbody>
</table>
Why three-step model can cover all?

- One cache access
  - Interference does not exist
- Two cache accesses
  - Same as three-step model with Step0 to be “∗”
- More than three cache accesses
  - \{⋯ \xrightarrow{⋆} \cdots \} can be divided into two parts
  - \{⋯ \xrightarrow{A_R} \xrightarrow{A_R} \cdots \}, \{⋯ \xrightarrow{A_R} \xrightarrow{V_R} \cdots \}, \{⋯ \xrightarrow{A_1} \xrightarrow{V_1} \cdots \}, \{⋯ \xrightarrow{V_x} \xrightarrow{V_x} \cdots \}, \cdots \ can be reduced to \{⋯ \xrightarrow{A_R} \cdots \}, \{⋯ \xrightarrow{V_R} \cdots \}, \{⋯ \xrightarrow{V_1} \cdots \}, \{⋯ \xrightarrow{V_x} \cdots \}, \cdots , respectively
  - \{⋯ \xrightarrow{(A_R/V_R / A_1 / V_1)} \xrightarrow{V_x} \xrightarrow{(A_R/V_R / A_1 / V_1)} \cdots \} maps to effective vulnerabilities represented by three-step model
• More than three cache accesses
  - \{\ldots \rightsquigarrow \ast \rightarrow \ldots \} can be divided into two parts
  - \{\ldots \rightarrow A_R \rightarrow A_R \rightarrow \ldots \}, \{\ldots \rightarrow A_R \rightarrow V_R \rightarrow \ldots \}, \ldots , \{\ldots \rightarrow A_1 \rightarrow V_1 \rightarrow \ldots \}, \ldots , \{\ldots \rightarrow V_x \rightarrow V_x \rightarrow \ldots \} can be reduced to \{\ldots \rightarrow A_R \rightarrow \ldots \}, \{\ldots \rightarrow V_R \rightarrow \ldots \}, \ldots , \{\ldots \rightarrow V_1 \rightarrow \ldots \}, \ldots , \{\ldots \rightarrow V_x \rightarrow \ldots \}, \text{respectively}
  - \{\ldots \rightarrow (A_R/V_R / A_1 / V_1) \rightarrow V_x \rightarrow (A_R/V_R / A_1 / V_1) \rightarrow \ldots \} maps to known vulnerabilities represented by three-step model
• Explicit enumeration of all the possible three steps (6x5x5=150)
• Identify 28 types of cache attacks
  – 20 types already known or categorized
  – 8 types previously not in literature
• Can be applied to evaluate any cache architecture with CTL logic
Vulnerability Exhaustive List

<table>
<thead>
<tr>
<th>Vx</th>
<th>Ax</th>
<th>Vx</th>
<th>Recognized name</th>
<th>Categorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vx</td>
<td>Ax</td>
<td>Vx</td>
<td>Type A</td>
<td></td>
</tr>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>Vx</td>
<td>Type B</td>
<td></td>
</tr>
<tr>
<td>AR</td>
<td>A1</td>
<td>Vx</td>
<td>Type C</td>
<td></td>
</tr>
<tr>
<td>VR</td>
<td>A1</td>
<td>Vx</td>
<td>Type D</td>
<td></td>
</tr>
<tr>
<td>A1</td>
<td>A1</td>
<td>Vx</td>
<td>Type E</td>
<td></td>
</tr>
<tr>
<td>V1</td>
<td>A1</td>
<td>Vx</td>
<td>Type F</td>
<td></td>
</tr>
<tr>
<td>Vx</td>
<td>A1</td>
<td>Vx</td>
<td>Evict+Time</td>
<td>Type G</td>
</tr>
<tr>
<td>AR</td>
<td>V1</td>
<td>Vx</td>
<td>Cache Collision</td>
<td>Type H</td>
</tr>
<tr>
<td>VR</td>
<td>V1</td>
<td>Vx</td>
<td>Cache Collision</td>
<td>Type I</td>
</tr>
<tr>
<td>A1</td>
<td>V1</td>
<td>Vx</td>
<td>Cache Collision</td>
<td>Type J</td>
</tr>
<tr>
<td>V1</td>
<td>V1</td>
<td>Vx</td>
<td>Cache Collision</td>
<td>Type K</td>
</tr>
<tr>
<td>Vx</td>
<td>V1</td>
<td>Vx</td>
<td>Bernstein's attack</td>
<td>Type L</td>
</tr>
<tr>
<td>AR</td>
<td>Vx</td>
<td>AR</td>
<td>Flush+Flush</td>
<td>Type M</td>
</tr>
<tr>
<td>VR</td>
<td>Vx</td>
<td>AR</td>
<td>Flush+Flush</td>
<td>Type N</td>
</tr>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>V1</td>
<td>Bernstein's attack</td>
<td>Type AA</td>
</tr>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>V1</td>
<td>Cache Collision</td>
<td>Type AB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vx</th>
<th>Vx</th>
<th>Ax</th>
<th>Recognized name</th>
<th>Categorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>Ax</td>
<td>Flush+Flush</td>
<td>Type O</td>
</tr>
<tr>
<td>AR</td>
<td>Vx</td>
<td>VR</td>
<td>Flush+Flush</td>
<td>Type P</td>
</tr>
<tr>
<td>VR</td>
<td>Vx</td>
<td>VR</td>
<td>Flush+Flush</td>
<td>Type Q</td>
</tr>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>VR</td>
<td>Flush+Flush</td>
<td>Type R</td>
</tr>
<tr>
<td>AR</td>
<td>Vx</td>
<td>A1</td>
<td>Flush(Evict)+Reload</td>
<td>Type S</td>
</tr>
<tr>
<td>VR</td>
<td>Vx</td>
<td>A1</td>
<td>Flush(Evict)+Reload</td>
<td>Type T</td>
</tr>
<tr>
<td>A1</td>
<td>Vx</td>
<td>A1</td>
<td>Prime+Probe</td>
<td>Type U</td>
</tr>
<tr>
<td>V1</td>
<td>Vx</td>
<td>A1</td>
<td></td>
<td>Type V</td>
</tr>
<tr>
<td>Vx</td>
<td>Vx</td>
<td>A1</td>
<td>Flush(Evict)+Reload</td>
<td>Type W</td>
</tr>
<tr>
<td>AR</td>
<td>Vx</td>
<td>V1</td>
<td>Cache Collision</td>
<td>Type X</td>
</tr>
<tr>
<td>VR</td>
<td>Vx</td>
<td>V1</td>
<td>Cache Collision</td>
<td>Type Y</td>
</tr>
<tr>
<td>A1</td>
<td>Vx</td>
<td>V1</td>
<td></td>
<td>Type Z</td>
</tr>
</tbody>
</table>

Note: The table represents various vulnerabilities and their categorization based on the recognized names and actions.
Vulnerability Examples

- **Flush + Reload Attack** (Type S, T Attack)
  
  \[ A_R \leftrightarrow V_x \leftrightarrow A_1 \]
  
  \[ V_R \leftrightarrow V_x \leftrightarrow A_1 \]

- **New Type V Attack**
  
  \[ V_1 \leftrightarrow V_x \leftrightarrow A_1 \]

---

Set-associative cache

<table>
<thead>
<tr>
<th>sets</th>
<th>ways</th>
</tr>
</thead>
</table>

1. Flush each line in the cache
2. Victim accesses critical data
3. Attacker reloads critical data by running specific process (measure time)

Set-associative cache

| sets | ways |

1. Victim probes each cache set (measure time)
Computation Tree Logic

Treats time as discrete and branching
Can explore different execution paths

- Atomic propositions: $\bullet, \bullet, \ldots$
- Boolean operators: $\neg \varphi, \varphi \lor \psi, \varphi \land \psi, \ldots$
- Temporal modalities:
  - $\mathsf{X} \varphi$ 
  - $\varphi \mathsf{U} \psi$
  - $\mathsf{F} \varphi$
  - $\mathsf{G} \varphi$
  - “next $\psi$”
  - “$\varphi$ until $\psi$”
  - “eventually $\varphi$”
  - “always $\varphi$”
- Path quantifiers:
  - $\mathsf{E} \psi$
  - $\mathsf{A} \psi$
For a single cache block, model execution paths that represent vulnerabilities to attacks:

\[(M, s) \models EF(E(E(Step0 U Step1) U Step2))\]

Eventually there exists a path that corresponds to the vulnerability:

\[Step0 \leadsto Step1 \leadsto Step2\]

\[E.g. A_R \leadsto V_x \leadsto A_1 \leftrightarrow EF(E(E(A_R UV_x) UA_1))\]
Bounded Computation Tree

three-step model:
Future Work

- hardware design of secure caches
- cache state machine modeling
- checking of vulnerability in CTL logic
- improve CTL modeling
Summary

- Develop Cache Access Model
  - Three-step single-cache-block-access model construction

- Analyze Timing Vulnerabilities
  - Exhaustive search for possible attacks based on three-step model

- Use Computation Tree Logic (CTL)
  - Model execution paths of the processor cache focusing on side-channel attacks

- Cache state machine modeling
- Checking of vulnerability in CTL logic
- Improve CTL modeling

Thank you!
back up slides
• One cache access
  – Interference does not exist
• Two cache accesses
  – Same as three-step model with $Step0$ to be “⋆”
  – None of them can form an attack
• Three cache accesses
  – Exhaustive vulnerability Search and effective vulnerabilities derived
More than three cache accesses

- \{\cdots \rightsquigarrow \star \rightsquigarrow \cdots \} can be divided into two parts
- \{\cdots \rightsquigarrow A_R \rightsquigarrow A_R \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow A_R \rightsquigarrow V_R \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow A_1 \rightsquigarrow V_1 \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow V_x \rightsquigarrow V_x \rightsquigarrow \cdots \}, \ldots \text{ can be reduced to} \{\cdots \rightsquigarrow A_R \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow V_R \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow V_1 \rightsquigarrow \cdots \}, \{\cdots \rightsquigarrow V_x \rightsquigarrow \cdots \}, \ldots, \text{ respectively}
- \{\cdots \rightsquigarrow (A_R/V_R / A_1 / V_1) \rightsquigarrow V_x \rightsquigarrow (A_R/V_R / A_1 / V_1) \rightsquigarrow \cdots \} maps to effective vulnerabilities represented by three-step model