Transient execution attacks and defenses

Frank Piessens

Seminar for the Software Security course
21/04/2022
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
System model: a shared platform

• A platform runs programs from multiple stakeholders
  • Isolation mechanism isolates these programs
  • The platform optionally supports communication between these programs

• Many systems are such shared platforms:
  • Cloud
  • Mobile
  • Desktop

• A variety of isolation mechanisms is used to limit interference between code from different stakeholders
  • Process isolation, virtual machines, enclaved execution, software-based isolation, ...
  • You have seen examples of such isolation mechanisms in an earlier lecture
Microarchitectural attacks

• Attacker code and victim code run on the same computing platform
  • They are architecturally isolated from each other (e.g., process isolation)
  • But they share microarchitectural resources
    • **Architectural state**: state as defined in the ISA spec (memory, registers, …)
    • **Microarchitectural state**: additional state in the processor implementation, e.g., for performance improvements (caches, branch predictors, various CPU buffers, …)
Relevant for security-critical software for a long time

• Instances of microarchitectural side-channel attacks have been known for 15+ years
  • E.g., cache timing attacks that were covered in an earlier lecture
  • Ge et al., A survey of microarchitectural timing attacks and countermeasures on contemporary hardware, J. Cryptographic Engineering, 2018

• The crypto community has developed solid countermeasures
  • E.g., constant-time programming
  • Almeida et al., Verifying Constant-Time Implementations, USENIX Security 2016
Transient execution attacks changed the game

• Spectre, Meltdown and Foreshadow (all publicly disclosed in 2018) showed how speculative and out-of-order execution significantly amplified the problem of microarchitectural attacks
  • Kocher et al. Spectre Attacks: Exploiting Speculative Execution, IEEE S&P 2019
  • Lipp et al. Meltdown: Reading Kernel Memory from User Space, USENIX Security 2018
  • Van Bulck et al. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution, USENIX Security 2018

• Many other variants followed: RIDL, ZombieLoad, Fallout, LVI, ...
  • https://mdsattacks.com/
  • https://cpu.fail/

• Strong academic and industry impact
Many variants

• Classification tree from:

• Further extended and maintained at:
  • https://transient.fail/
Academic and industry impact

- 30+ CVE’s in the 2017-2021 timeframe
- Significant efforts by technology giants to mitigate the risks
Towards a principled understanding

• Both the discovery of vulnerabilities as well as the development of countermeasures has been a productive but chaotic process
  • Academics compete for finding issues first
  • Companies protect their customers through embargos and ad-hoc countermeasures

• But this class of vulnerabilities is important enough to deserve a systematic foundational study
  • How should we model processors to reason about microarchitectural vulnerabilities, attacks and countermeasures? What is the exact security objective?
  • This seminar will follow the language-based approach, of which many instances exist:
    • Mcilroy et al., Spectre is here to stay: An analysis of side-channels and speculative execution, arXiv 2019.
    • Cauligi et al., Constant-time foundations for the new Spectre era, PLDI 2020.
    • Guanciale et al., InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis, CCS 2020.
    • Guarnieri et al., Hardware/software contracts for secure speculation, IEEE S&P 2021.
    • (this list is not complete)
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
A simple Instruction Set Architecture (ISA) model

Register names $r \in \text{Regs}$
Values $v \in \mathbb{N}$
Expressions $e ::= v \mid r \mid e + e \mid e < e \mid \ldots$
Instructions $i ::= r \leftarrow e$
$\quad r \leftarrow \text{load}[e]$
$\quad \text{store}[e] \leftarrow r$
$\quad \text{jmp } e$
$\quad \text{beqz } r \ v$

Programs $p ::= \overrightarrow{i}$

(E.g., $r_0, r_1, i, \text{len}$. We assume $\text{pc} \notin \text{Regs}$)
(Also represent addresses)
(Boolean expressions return 0 or 1)
(Assign value of $e$ to $r$)
(Load value at memory address $e$ into $r$)
(Store $r$ in memory at address $e$)
(Jump to code address $e$)
(Branch to $v$ if $r$ evaluates to 0)
(Non-empty list of instructions)

Example program:

0 : $r_0 \leftarrow i < 2$ ; while ($i < 2$) {
1 : \text{beqz } r_0 \ 6$
2 : $r_0 \leftarrow \text{load}[a + i]$ ; sum = sum + $a[i]$
3 : \text{sum } \leftarrow \text{sum} + r_0$
4 : $i \leftarrow i + 1$ ; $i = i + 1$
5 : \text{jmp } 0$ ;
}
A simple Instruction Set Architecture (ISA) model

Register names $r \in \text{Regs}$

Values $v \in \mathbb{N}$

Expressions $e ::= v | r | e + e | e < e | \ldots$ (boolean expressions return 0 or 1)

Instructions $i ::= r \leftarrow e \quad \text{(assign value of } e \text{ to } r)$

$\quad r \leftarrow \text{load}[e] \quad \text{(load value at memory address } e \text{ into } r)$

$\quad \text{store}[e] \leftarrow r \quad \text{(store } r \text{ in memory at address } e)$

$\quad \text{jmp } e \quad \text{(jump to code address } e)$

$\quad \text{beqz } r \ v \quad \text{(branch to } v \text{ if } r \text{ evaluates to } 0)$

Programs $p ::= i$

(E.g., $r_0, r_1, i, \text{len}$. We assume $pc \notin \text{Regs}$)

(Also represent addresses)

Example program:

0 : $r_0 \leftarrow i < 2$ ; while $(i < 2)$ {
1 : $\text{beqz } r_0 \ 6$
2 : $r_0 \leftarrow \text{load}[a + i] \quad \text{sum} = \text{sum} + a[i]$
3 : $\text{sum} \leftarrow \text{sum} + r_0$
4 : $i \leftarrow i + 1 \quad i = i + 1$
5 : $\text{jmp } 0$

Registers: pc=0

Memory:
Base semantics

Register state $\rho \in \text{Regs} \rightarrow \text{Values}$
Memory state $m ::= \vec{v}$
Program counter $pc ::= v$
Program state $\sigma ::= (m, \rho, pc)$

Program:
0: $r_0 \leftarrow i < 2$
1: beqz $r_0 6$
2: $r_0 \leftarrow \text{load}[a + i]$
3: $\text{sum} \leftarrow \text{sum} + r_0$
4: $i \leftarrow i + 1$
5: jmp 0

Memory:

<table>
<thead>
<tr>
<th></th>
<th>0: 5</th>
<th>1: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sum</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>r0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Architectural isolation

• We think of architectural state as securely partitioned
  • Programs by different stakeholders are (architecturally) isolated from one another
  • At the level of abstraction of the ISA, no information leaks between ISA programs of different stakeholders
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
Out-of-order and speculative execution

• Transient execution attacks exploit processor features called out-of-order and speculative execution

• The basic idea is:
  • Rather than executing one instruction at a time, fetch many instructions into a buffer of in-flight instructions
  • Execute instructions from this buffer, possibly out-of-order
    • This avoids having to wait while, for instance a slow memory load is happening
  • Commit the effect of the instructions to the architectural state in order

• Prediction and speculation are used to speed things up
  • For instance, fetching instructions beyond a branch requires prediction
Out-of-order and speculative execution

In-flight instructions

\[ f ::= r \leftarrow e \]
\[ r \leftarrow load[e] \]
\[ store[e] \leftarrow r \]
\[ pc \leftarrow v \]
\[ @v : pc \leftarrow v \]
\[ @v : r \leftarrow v \]

(non-speculated jump becomes pc assignment)
(speculated jump, v is address of original instruction)
(speculated load, v is address of original instruction)

Reorder buffer

\[ rob ::= \]
Program state

\[ \sigma ::= (m, \rho, pc, rob) \]

<table>
<thead>
<tr>
<th>pc=0</th>
<th>pc=0</th>
<th>pc=0</th>
<th>pc=1</th>
<th>pc=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 : (i \leftarrow 2 + 2)</td>
<td>1 : (n \leftarrow load[12])</td>
<td>2 : (r_0 \leftarrow i &lt; n)</td>
<td>3 : beqz (r_0 5)</td>
<td>4 : (i \leftarrow 4 \times i)</td>
</tr>
<tr>
<td>(i)</td>
<td>0</td>
<td>(i)</td>
<td>0</td>
<td>(i)</td>
</tr>
<tr>
<td>(n)</td>
<td>0</td>
<td>(n)</td>
<td>0</td>
<td>(n)</td>
</tr>
<tr>
<td>(r_0)</td>
<td>0</td>
<td>(r_0)</td>
<td>0</td>
<td>(r_0)</td>
</tr>
<tr>
<td>(\text{rob})</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pc=1</td>
<td>pc=1</td>
<td>pc=1</td>
<td>pc=3</td>
<td>pc=4</td>
</tr>
<tr>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
<td>i</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>rob</td>
<td>rob</td>
<td>rob</td>
<td>rob</td>
<td>rob</td>
</tr>
<tr>
<td>(n \leftarrow \text{load}[12]) [r0 \leftarrow i &lt; n] [@3: pc \leftarrow 5] [i \leftarrow i + 1]</td>
<td>(n \leftarrow \text{load}[12]) [r0 \leftarrow i &lt; n] [@3: pc \leftarrow 5] [i \leftarrow 5]</td>
<td>(n \leftarrow 7) [r0 \leftarrow 1] [@3: pc \leftarrow 5] [i \leftarrow 5]</td>
<td>()</td>
<td>()</td>
</tr>
</tbody>
</table>

**Incorrect prediction**

**Correct prediction**
Predictions and scheduling

• The semantics requires the processor to make choices, for instance for predicted values
  • These happen based on heuristics and observing past behavior
  • Hence, they can also be influenced by an attacker
    • E.g., “training the branch-predictor”

• How should we model this influence of the attacker?
Overview

- Introduction
- A simple Instruction Set Architecture (ISA) model
- Out-of-order and speculative execution
- Modeling the attacker
- Transient execution attacks by example
- Towards defenses
- Conclusions
Attacker model

• Transient execution attacks build on:
  • Classic microarchitectural side-channel and covert-channel attacks,
    • For instance, cache attacks, but there are many more
  • The fact that the attacker can influence the speculative and out-of-order execution
    • For instance, by “training” the branch predictor, but there are many other ways

• To make reasoning about these attacks manageable yet secure, we **overapproximate and simplify**
  • The “constant-time leakage model” models what an attacker can learn through classic side-channels
  • We give the attacker full control over predictions and scheduling
  • Note that this significantly simplifies attack examples!
    • Doing the example attacks we will discuss on a real system can be very labor-intensive
The constant time leakage model

- Extend base semantics to specify what leaks at each step:

Program:

<table>
<thead>
<tr>
<th>PC</th>
<th>a</th>
<th>i</th>
<th>sum</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>i+1</td>
<td>sum</td>
<td>r0</td>
</tr>
<tr>
<td>4</td>
<td>i+1</td>
<td>1</td>
<td>sum</td>
<td>r0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>sum</td>
<td>r0</td>
</tr>
</tbody>
</table>

Memory:

<table>
<thead>
<tr>
<th>PC</th>
<th>a</th>
<th>i</th>
<th>sum</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

access @0

<table>
<thead>
<tr>
<th>PC</th>
<th>a</th>
<th>i</th>
<th>sum</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

access @1

<table>
<thead>
<tr>
<th>PC</th>
<th>a</th>
<th>i</th>
<th>sum</th>
<th>r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Leak gadgets

• In later attacks, we rely on code snippets that leak secrets through a microarchitectural side-channel
  • A wide variety of such snippets exist
  • In some scenarios, the attacker can construct them, in other scenarios the attacker has to find them in victim code

• For simplicity, we will define:

```
leak secret ::= dummy ← load[secret]
```

(where secret is the name of a register containing the secret to be leaked, and dummy is an otherwise unused register)
Attacker influence on the execution

- Prediction and scheduling choices can be done by the attacker within constraints defined in the semantics, e.g.:
  - Fetch is only possible if the reorder buffer has room
  - Executing an instruction in the reorder buffer is only possible if its dependencies are satisfied
  - Commit is only possible for the oldest instruction in the reorder buffer, and only after it has fully executed
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
Modeling transient execution attacks

• We have seen that instructions can execute transiently
• This impacts security in two ways:
  • Transiently executed instructions can also leak information to the attacker
    • On rollback, architectural effects are discarded, but microarchitectural effects remain
  • Transiently executed instructions can **access** information expected to be inaccessible
    • Because the information is protected by software -> “Spectre”-style attacks
    • Because it is in another hardware protection domain -> “Meltdown”-style attacks
Spectre examples

• We will discuss a couple of Spectre examples

• In each example:
  • There is code operating in a program state containing secrets
  • According to the base ISA semantics, the code does not leak these secrets
    • Even taking into account “classic” side-channels
    • For instance, all the examples satisfy the constant-time coding discipline
  • Yet, because of speculation and out-of-order execution, the secrets do leak
Example 1: Spectre v1 (Spectre-PHT)

0: $len \leftarrow \text{load}[a - 1]$ ; assume length field stored before array
1: $r_0 \leftarrow i < len$
2: $\text{beqz } r_0 5$ ; if($i < len$) {
3: $r_0 \leftarrow \text{load}[a + i]$ ; $r_0 = a[i]$
4: $\text{leak } r_0$ ; $\text{leak}(r_0)$
5: ... ; }

Memory:

<table>
<thead>
<tr>
<th>addr</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>1234</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

pc=0

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>rob</td>
<td></td>
</tr>
</tbody>
</table>

len: 0

access @3

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>rob</td>
<td></td>
</tr>
</tbody>
</table>

access @1234

<table>
<thead>
<tr>
<th>a</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
</tr>
<tr>
<td>rob</td>
<td></td>
</tr>
</tbody>
</table>
Example 2: Spectre v2 (Spectre-BTB)

```
0: r0 ← load[7] ; load a secret into r0
1: fptr ← load[8] ; load a "function pointer" to a trusted function
2: jmp fptr ; call trusted function that safely accesses secret
... ...
20: r0 ← 0 ; trusted function just clears secret.
21: jmp 3
... ...
31: leak r0
```

Memory:

<table>
<thead>
<tr>
<th>1234</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>1234</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

```plaintext
pc=0
fptr  0
r0    0
rob   

access @3 pc=0
fptr  0
r0    0
rob   

pc=1
fptr  0
r0    1234
rob   

access @1234 pc=1
fptr  0
r0    0
rob   
```

```
@2 : pc ← 31
dummy ← load[r0]
```
Example 3: Spectre v4 (Spectre-STL)

Suppose memory address 7 contains the secret 1234, that is currently cached.

0: \( r_0 \leftarrow 0 \)
1: \( \text{store}[4 + 3] \leftarrow r_0 \)
   ; overwrite the secret with 0
2: \( r_0 \leftarrow \text{load}[7] \)
   ; load address 7 into \( r_0 \), should read 0
3: \( \text{leak} \ r_0 \)

...
Transient execution attacks

• These were a couple of simplified Spectre attacks
  • See https://transient.fail/ for more variants and more details
• Note the devastating nature of this kind of attack on software-enforced confidentiality properties

References

- A Systematic Evaluation of Transient Execution Attacks and Defenses
  Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, Daniel Gruss (USENIX Security 2019)
- Spectre Attacks: Exploiting Speculative Execution
- BranchScope: A New Side-Channel Attack on Directional Branch Predictor
  Dmitry Evtushkin, Ryan Riley, Nael Abu-Ghazaleh, Dmitry Pomorarev (ASPLOS 2018)
- The microarchitecture of Intel, AMD and VIA CPUs
  Agner Fog
Meltdown example: faulting loads

• Surprisingly, memory loads that raise a fault (e.g., page fault or protection fault) still execute transiently (on some processors)

• The essence of Meltdown:

\[
\begin{align*}
  r_0 & \leftarrow \text{load}[\text{kernel\_address}] \quad ; \text{this load will raise a fault} \\
  \text{leak } r_0
\end{align*}
\]

• Later papers have shown that faulting loads (or loads that receive “microcode assists”) compute transiently on all kinds of potentially sensitive data
See again transient.fail for an overview
General transient execution attack structure

1. Prime the micro-architectural state
2. Trigger transient execution (misprediction or fault)
3. Send on the covert channel
4. CPU flushes architectural effects of transient execution
5. Read from the covert channel

Source: Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, Daniel Gruss, A systematic evaluation of transient execution attacks and defenses, Usenix Security 2019
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
Defenses

• Defenses are being investigated at multiple levels:
  • Hardware mitigations
    • For instance, do not forward values from faulting loads to subsequent instructions
  • Mitigations in the Operating System
    • For instance, do not place the kernel in the same virtual address space as user code
  • Mitigations in the compiler
    • For instance, insert instructions to stop out-of-order execution, or rewrite code to remove the vulnerability

• Meltdown-style vulnerabilities are being addressed in hardware

• For Spectre-style vulnerabilities, good defenses are still the subject of ongoing research
Security objective of defenses

• Transient execution attacks cause unexpected information flows, and hence the security of a program against these attacks can be defined using techniques from information flow security

• We define a policy as an equivalence relation over program states
  • The intuition is that the policy relates states that should be indistinguishable to an attacker. Typically, one defines a policy by marking secrets, and two states are equivalent if they only differ in secrets.

• A program $P$ is secure on hardware $H$ if executing $P$ on $H$ starting from any two equivalent initial states will produce identical observations for the attacker
  • Security can be achieved by software mitigations, or by hardware mitigations, or by a combination of both
Reconsider the Spectre v1 example:

```plaintext
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 5 ; if(i < len) {
3 : r0 ← load[a + i] ; r0 = a[i]
4 : leak r0 ; leak(r0)
5 : ...
}
```

Memory:

<table>
<thead>
<tr>
<th>pc=0</th>
<th>a</th>
<th>r0</th>
<th>rob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>len</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pc=0</th>
<th>a</th>
<th>r0</th>
<th>rob</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>len</td>
<td>i</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>access @3 pc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>access @1234 pc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc=0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234: 0</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>3: 1234</td>
</tr>
<tr>
<td>2: 5</td>
</tr>
<tr>
<td>1: 3</td>
</tr>
<tr>
<td>len: 0</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

A hardened version of the program is secure:

```plaintext
0: len ← load[a - 1] ; assume length field stored before array
1: r0 ← i < len
2: beqz r0 6 ; if(i < len) {
  3: i ← i % len ; i ← i % len
  4: r0 ← load[a + i] ; r0 ← a[i]
  5: leak r0 ; leak(r0)
  6: ...
}
```

<table>
<thead>
<tr>
<th>pc=0</th>
<th>access @0 pc=0</th>
<th>access @1 pc=0</th>
<th>access @3 pc=0</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>len</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>i</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
| rob  | len ← load[a - 1]  
      | r0 ← i < len    
      | @2 : pc ← 3     
      | i ← i % len      
      | r0 ← load[a + i]  
      | dummy ← load[r0] |
|      | *              | *              | *              |
|      | len ← 2        
      | r0 ← i < len    
      | @2 : pc ← 3     
      | i ← 0           
      | r0 ← load[a + i]  
      | dummy ← load[r0] |
|      | *              | *              | *              |
|      | len ← 2        
      | r0 ← i < len    
      | @2 : pc ← 3     
      | i ← 0           
      | r0 ← 3          
      | dummy ← load[r0] |
|      | *              | *              | *              |
|      | len ← 2        
      | r0 ← i < len    
      | @2 : pc ← 3     
      | i ← 0           
      | r0 ← 3          
      | dummy ← 1234    |
|      | *              | *              | *              |
```

Memory:

- 1234: 0
- 3: 1234
- a: 2
- 1: 3
- len: 0: 2
- ...
Hardening with speculation barriers

```
0 : len ← load[a - 1] ; assume length field stored before array
1 : r0 ← i < len
2 : beqz r0 6 ; if(i < len) {
3 : fence ;
4 : r0 ← load[a + i] ; r0 = a[i]
5 : leak r0 ; leak(r0)
6 : ...
```

Memory:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1234:</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>1234</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a:</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>len:</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
Different scenarios for defenses

• Attacks can be decomposed in two steps:
  1. **Accessing** the secret
  2. **Leaking** it through a microarchitectural channel

• Three scenarios:
  • Access and leak are both non-transient
    • This is a classic side-channel attack, and it can be defended, e.g., with CT programming
  • Access is non-transient, leak is transient
    • The “secure programming” scenario, for instance cryptographic code
    • Can we extend the CT programming discipline to also close these leaks?
  • Both access and leak are transient
    • The “sandboxing” scenario: victim host program executes attacker code using software-based isolation
      • Typical example: a browser running WebAssembly code
    • Sufficient condition for security: hardware ensures that transiently accessed data does not leak
Overview

• Introduction
• A simple Instruction Set Architecture (ISA) model
• Out-of-order and speculative execution
• Modeling the attacker
• Transient execution attacks by example
• Towards defenses
• Conclusions
Conclusions

• Transient execution attacks are a fundamentally new class of attacks:
  • That break many important security mechanisms
  • That are not easy to defend against

• Short-term defenses have been useful but ad-hoc

• Long-term defenses are the subject of current research
  • Pure software defenses against Spectre will remain important for the foreseeable future and are the subject of active research.
    • For a recent survey, see:
  • Hardware/software co-designs can offer better security/performance trade-offs
    • Excellent starting point for reading more:
      • Guarnieri et al., Hardware/software contracts for secure speculation, IEEE S&P 2021