Side Channel Mitigations

Mengjia Yan
Spring 2022
Outline

• **Non-interference**: a general security property

• Verify Non-interference for Side Channels and Transient Execution

• Hardware and Software Contract
Non-Interference Example

- Intuitively: not affecting
- Any sequence of low inputs will produce the same low outputs, regardless of what the high level inputs are.
Non-Interference Example
Use Cases of Non-interference

• Confidentiality: e.g., process isolation
  • My memory -> Confidentiality of High state
  • Other programs’ memory -> Low

• Integrity: e.g., control-flow hijacking
  • My memory -> Integrity of Low State
  • Attacker controlled input -> High

• Swirl example

• Expand High-Low to Lattice

"Lattice-Based Access Control Models; Ravi S. Sandhu; 1993"
Non-Interference Formulation

• Formulate the property as state-machine transition.
• Looking at a single-trace is ineffective

∀ S1, S2, (S, P) → S’
if S1_L = S2_L
then S1’_L = S2’_L

6.888 L6 - Side Channel Mitigations
Generality of Non-interference

- Conventionally: ISA emulation for software analysis/testing

- Can also be used for hardware security design
  - Micro-architecture: state includes caches, buffers, buses, etc.
  - Circuit level: flip-flop
Taint Analysis (also Taint Tracking)

- Goal: verify non-interference property
- Analogy
- Components:
  - Source of taint (high state)
  - Taint propagation
  - Taint check (no taint on low state)

1: \( x = \text{get\_secret}() \)
2: \( y = \text{array}[x] \)
Explicit and Implicit Information Flow

Control-Flow
1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }

Address Alias
1: x = secret;
2: array[0] = x;
3: z = array[y];
Taint Analysis Methods

• Dynamic: run-time check
  • Detect non-interference violation on-the-fly for a given input

• Static: compiler-time check
  • Verify whether a given program is secure/bug-free for arbitrary input
Dynamic Taint Analysis

- Problems:
  - Granularity
  - Run-time overhead
  - How to handle implicit flow?

<table>
<thead>
<tr>
<th>Sub-State</th>
<th>H/L</th>
<th>Tainted</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>L</td>
<td>0</td>
</tr>
</tbody>
</table>

6.888 L6 - Side Channel Mitigations
Dynamic Taint Analysis

- Problems:
  - Granularity
  - Run-time overhead
  - How to handle implicit flow?

1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }

1: BREQ x 4
2: y < 1
3: JMP 5
4: z < 2
5: ...

Taint PC.
Taint Explosion.

How to deal with it?
Static Taint Analysis

• Key differences
  • Verify whether a given program is secure for arbitrary inputs
  • Can leverage high-level program information

```plaintext
1: x = secret;
2: if (x == 0) {
3:     y = 1;
4: } else {
5:     z = 2;
6: }
```

Control Flow Graph (CFG)
Static Taint Analysis

- Problems
  - Scalability (check all possible inputs)
  - How to handle implicit flow?

```
1: x = secret;
2: array[0] = x;
3: z = array[y];
```

```
1: x = secret;
2: *ptr1 = x;
3: z = *ptr2;
```

How to deal with it?

If conservative, Taint Explosion.
Takeaways

• Non-interference property: general security property for both confidentiality and integrity

• Taint Analysis
  • Useful techniques for checking non-interference
    • Static: verification tool
    • Dynamic: online monitoring
  • Both have taint explosion problems
Non-interference for Timing Side Channels

• How to define non-interference for timing side channels?

• How to check whether a given mitigation achieves non-interference or not?

• How to coordinate software and hardware mitigations? How to reason security about software-hardware co-design?
  • Given SW x, running on HW y can protect all data containing secret z?
    \{SW x, HW y, sec z\}
Non-interference at Micro-arch Level

\[ \forall S_1, S_2, \quad (S, P) \rightarrow S' \]

\[
\text{if } S_{1L} = S_{2L} \\
\text{then } S'_{1L} = S'_{2L}
\]

<table>
<thead>
<tr>
<th></th>
<th>State</th>
<th>State Transition (Program Execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Analysis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Micro-arch Side Channel</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Non-interference at Micro-arch Level

### Program State Transition Table

<table>
<thead>
<tr>
<th>Software Analysis</th>
<th>State</th>
<th>State Transition (Program Execution)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micro-arch Side Channel</td>
<td>Register, Memory (Physical)</td>
<td>ISA Emulation</td>
</tr>
<tr>
<td></td>
<td>Cache, BTB, Bus Busy Bits,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Pipeline ROB status etc</td>
<td></td>
</tr>
</tbody>
</table>

∀ \( S_1, S_2, (S, P) \rightarrow S' \)

if \( S_{1L} = S_{2L} \)

then \( S'_{1L} = S'_{2L} \)

---

6.888 L6 - Side Channel Mitigations
Verify HW Design Using Static IFT

Annotate variables (registers and wires) with security labels.

```plaintext
1  reg {L} v, {L} l, {H} h;
2  // LH (0) = L, LH (1) = H
3  wire {LH(v)} shared;
4  // l=h is forbidden
5  if (v == 0) l = shared;
6  else h = shared;
7  // implicit flow, not allowed
8  if (h == 0) l = 0;
9  else l = 1;
```

**NS:** a bit to indicate normal world or secure world

**Figure 1.** TrustZone prototype implementation.

Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis; Ferraiuolo et al; ASPLOS’17

HyperFlow: A Processor Architecture for Nonmalleable, Timing-Safe Information Flow Security; Ferraiuolo et al; CCS’18
Non-Interference at Gate Level

- Dynamic taint tracking

```
Logic Truth Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

2-input AND gate

2-input OR gate

Shadow Taint Logic

Sound, yet Conservative

Complete Information Flow Tracking from the Gates Up; Tiwari et al; ASPLOS’09

6.888 L6 - Side Channel Mitigations
Non-Interference at Gate Level

• Dynamic taint tracking

2-input AND gate

Shadow Taint Logic OR gate

Sound, yet Conservative

Precise Taint Logic

Complete Information Flow Tracking from the Gates Up; Tiwari et al; ASPLOS’09

6.888 L6 - Side Channel Mitigations
Non-interference at Micro-arch Level

<table>
<thead>
<tr>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program

<table>
<thead>
<tr>
<th>High'</th>
<th>Low'</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Transition (Program Execution)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Analysis</th>
<th>Register, Memory (virtual)</th>
<th>ISA Emulation</th>
</tr>
</thead>
</table>

| Micro-arch Side Channel | Register, Memory (Physical) Cache, BTB, Bus Busy Bits, Pipeline ROB status etc | Detailed Execution |

6.888 L6 - Side Channel Mitigations
Non-interference at Micro-arch Level

<table>
<thead>
<tr>
<th>Program</th>
<th>Program Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>High’</td>
</tr>
<tr>
<td>Low</td>
<td>Low’</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Analysis</td>
</tr>
<tr>
<td>Register, Memory (virtual)</td>
</tr>
<tr>
<td>Micro-arch Side Channel</td>
</tr>
<tr>
<td>Register, Memory (Physical), Cache, BTB, Bus Busy Bits, Pipeline ROB status etc</td>
</tr>
<tr>
<td>ISA Emulation</td>
</tr>
<tr>
<td>Detailed Execution</td>
</tr>
</tbody>
</table>

Yes, we can. But ...

Can we use the same definition to reason about software mitigations?
“Constant-time” Programming

• Write program w/o data-dependent behavior
• Verify non-interference of timing side channels by simulating micro-arch state machine.
• Problems?

Original:

```plaintext
bool secret;
x <- pub[secret*64];
```

Data Oblivious:

```plaintext
bool secret;
a <- pub[0];
b <- pub[64];
cmov x <- (secret) ? b : a;
```
Observation Model

• Motivation:
  • Avoid verifying SW against specific implementations

• Observations:
  • *Program counters, Memory access addresses, Memory access data, Register data*
  • **Dependent on hardware implementation**
Verify “Constant-time” Programming

• Using Observations

Original:

```c
bool secret;
x <- pub[secret*64];
```

Data Oblivious:

```c
bool secret;
a <- pub[0];
b <- pub[64];
cmov x <- (secret) ? b : a;
```

Memory access sequence:

- Original: H
- Data Oblivious: 0 (L), 64 (L)
Takeaways

• How to verify non-interference of timing side channels?
  
  • To check HW: state transition at micro-arch and gate level
  
  • To check SW: define observation model
    • Observation model can be served as a contract between HW and SW

Shall we always assume memory access sequence as the observation?

No. It is hardware dependent. Think about Silent Store and Cache compression.
“Constant-time Programming” Fails in the Spectre Era

Original:

```c
if (x < limit){ //limit=4
    y <- pub1[x];
    z <- pub2[y*64];
}
```

Memory access sequence:

x (L), y (L)

Memory Layout:

```
pub1[0]
pub1[1]
pub1[2]
pub1[3]
secret
```
Execution Model

• Motivation:
  • Incorporate speculative execution in an execution model

• Add Execution Model:
  • Sequential, Branch mis-speculation, etc.

∀ S₁, S₂, (S, P) → S', O

if S₁ᴸ = S₂ᴸ

then S₁'L = S₂'L and O₁ = O₂

Hardware-Software Contracts for Secure Speculation; Guarnieri et al; S&P’20
“Constant-time Programming” Fails in the Spectre Era

Original:

```c
if (x < limit){
    //limit=4
    y <- pub1[x];
    z <- pub2[y*64];
}
```

Memory access sequence (no mispredict):

- x (L), y (L)

Memory access sequence (with mispredict):

- x (L), y (H)

Memory Layout:

- pub1[0]
- pub1[1]
- pub1[2]
- pub1[3]
- secret
SW Mitigations Against Spectre

**fence:**

```c
if (x < limit){
    lfence();
    y <- pub1[x];
    z <- pub2[y*64];
}
```

**SLH:**

```c
if (x < limit){
    cmov mask <- (i < limit) 0xFFFF:0
    y <- pub1[x] & mask;
    z <- pub2[x*64];
}
```

Memory access sequence (with mispredict):
- ∅
- 0 (L), y (L)

Chandler Carruth. Speculative Load Hardening. [https://llvm.org/docs/SpeculativeLoadHardening.html](https://llvm.org/docs/SpeculativeLoadHardening.html)
HW Solutions Targeting Many Transient Execution Attacks

- Meltdown
- Spectre-PHT/BTB
- Spectre
- Spectre-RSB
- Spectre-ROP
- NetSpectre
- SS RE
- RSRE
- LazyFP
- Forshadow-NG
- Forshadow

Year of 2018
Generalization of Transient Execution

• Different transient execution attacks create transient instructions in different ways
• Speculative attack model: an attacker can exploit any speculative insts

Speculative Attack Model

Various events, such as:
• Control-flow mispredictions → Spectre
• Virtual memory exceptions → Meltdown
• Address alias between a load and an earlier store
• Interrupts
• etc.
Lifetime of a Load Instruction

Load is issued to memory

Load is speculative

All prior branches are resolved

Load reaches head of Reorder Buffer

Spectre attack model

Unsafe

Safe

Comprehensive attack model

Unsafe

Unsafe

Visibility Point

The load becomes unsquashable

Load can be made visible without compromising security
Naïve Solution: Delay all spec Loads

Load issued to memory

Visibility Point

delay

Load reaches head of ROB
Delay-on-Miss (DoM)

Load is issued to L1

Load issued to L1

Hit

Visibility Point

Load reaches head of ROB

Visibility Point

Load issued to memory

Load is issued to L1

Miss

Visibility Point

Load reaches head of ROB

Christos Sakalis, et al. Efficient invisible speculative execution through selective delay and value prediction. ISCA’19
Performance Optimizations

// x is committed
Br: if (x < size){
// speculation starts here
Ld1: y = array1[x]
Ld2: z = array2[y]
}

**x**: Committed register state (exists in legal execution)

**y**: Transient register state (does not exist in legal execution)

**Insight**: Only need to protect the instructions that use transient states
STT, NDA, etc

// x is committed
Br: if (x < size) {
// speculation starts here
Ld1: y = array1[x]
Ld2: z = array2[y]
}

Access Instruction (brings transient state into pipeline)

Transmit Instruction (uses transient state)

Taint Tracking of Speculative Data

Only Protect Transmit Instructions

Comparing Two Approaches

// x is committed
Br: if (x < size){
// speculation starts here
Ld1: y = array1[x]
Ld2: z = array2[y]
}

Ld2 is secure:

DoM/InvisiSpec/…
STT/NDA/…

Ld2 reaches visibility point
Need protection

When Ld1 reaches visibility point
Need protection

NO protection

Speculative
Committed
Comparing Two Approaches

// x is committed

Ld1: \( y = \text{array1}[x] \)

Br: if \((x < \text{size})\){

// speculation starts here

Ld2: \( z = \text{array2}[y] \) \( \rightarrow \) Need protection

InvisiSpec/DoM/…

STT/NDA/…

NO protection

Speculative

Committed

6.888 L6 - Side Channel Mitigations
Problems

• Different HW mitigations achieve different security properties
• How to communicate this information to SW?
  • ISA?
  • List code patterns?
  • Specify execution model + observation model?

Current state-of-the-art.
May not be the final solution.
An unsolved research problem.
## Analyze Security Properties

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td>Speculative (can mispredict)</td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>+ Memory Address</td>
<td></td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>+ Memory Address</td>
<td></td>
</tr>
<tr>
<td>+ Register Content</td>
<td></td>
</tr>
</tbody>
</table>

**Enable Speculation**

**No Protection**
## Analyze Security Properties

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td>Speculative (can mispredict)</td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>Program Counter + Memory Address</td>
<td>Disable Speculation</td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td>No Protection</td>
</tr>
</tbody>
</table>
## Analyze Security Properties

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td>Speculative (can mispredict)</td>
</tr>
<tr>
<td>Program Counter</td>
<td></td>
</tr>
<tr>
<td>Program Counter + Memory Address</td>
<td>Disable Speculation</td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td></td>
</tr>
</tbody>
</table>

6.888 L6 - Side Channel Mitigations
## Analyze Security Properties

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td>Sequential</td>
<td>Speculative (can mispredict)</td>
</tr>
<tr>
<td>Program Counter + Memory Address</td>
<td>Disable Speculation</td>
<td></td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td>DoM</td>
<td></td>
</tr>
</tbody>
</table>
# Analyze Security Properties

## Observation Model

<table>
<thead>
<tr>
<th>Program Counter</th>
<th>Program Counter + Memory Address</th>
<th>Program Counter + Memory Address + Register Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoM</td>
<td>Disable Speculation</td>
<td>DoM</td>
</tr>
</tbody>
</table>

## Execution Model

<table>
<thead>
<tr>
<th>STT</th>
<th>Sequential</th>
<th>Speculative (can mispredict)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DoM</td>
</tr>
</tbody>
</table>
### Analyze Security Properties

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Sequential</th>
<th>Speculative (can mispredict)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter</td>
<td></td>
<td>DoM</td>
</tr>
<tr>
<td>Program Counter + Memory Address</td>
<td>Disable Speculation</td>
<td>STT</td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td>DoM</td>
<td>STT</td>
</tr>
</tbody>
</table>
Summary

• Non-interference
  • A general security property that can be used to reason software security and micro-arch side channels
  • Pros/Cons of static and dynamic taint analysis

• Reason about non-interference for side mitigations
  • Both observation model and execution model are hardware dependent

• Fundamental problem, timing is not defined at the contract between HW and SW (currently ISA)