Software-Hardware Contract for Side Channel Defenses

Mengjia Yan
Spring 2023
**Attack Examples**

Example #1: termination time vulnerability

```python
def check_password(input):
    size = len(password);
    for i in range(0, size):
        if (input[i] == password[i]):
            return "error";
    return "success";
```

Example #2: RSA cache vulnerability

```plaintext
for i = n-1 to 0 do
    r = sqr(r)
    r = r mod n
    if e_i == 1 then
        r = mul(r, b)
        r = r mod n
    end
end
```

Example #3: Meltdown

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

**Example #2: RSA cache vulnerability**

```plaintext
for i = n-1 to 0 do
    r = sqr(r)
    r = r mod n
    if e_i == 1 then
        r = mul(r, b)
        r = r mod n
    end
end
```
Who to blame? Who to fix the problem?
Break SW and HW Contract

The contract for functional correctness.

Software

Hardware

Instruction Set Architecture (ISA)
Software Developer's Problem

Software developers:
• Need to write software for devices with unknown design details.
• How can I know whether the program is secure running on different devices?
Hardware Designer’s Problem

Hardware designer:
• Need to design processors for arbitrary programs.
• How to describe what kind of programs can run securely on my device?
Example: Termination Time Vulnerability

• How to fix it?

```python
def check_password(input):
    size = len(password);
    for i in range(0, size):
        if (input[i] != password[i]):
            return "error";
    return "success";
```

Make the computation time independent from the secret (password)
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: A Formal Definition

• The definition of noninterference for a deterministic program $P$

$$\forall M_1, M_2, P$$

$$M_{1_L} = M_{2_L} \land (M_1, P) \rightarrow^* M_1' \land (M_2, P) \rightarrow^* M_2'$$

$$\Rightarrow M_{1'_L} = M_{2'_L}$$
Non-Interference for Side Channels

• The definition of noninterference for a deterministic program $P$

$\forall M1, M2, P$

$M1_L = M2_L \land (M1, P) \xrightarrow{01}^* M1' \land (M2, P) \xrightarrow{02}^* M2'$

$\Rightarrow O1 = O2$

What should be included in the observation trace?
Understand the Property

Consider input as part of $M$

- What is $M_L$?
- What is $M_H$?
- What is $O$?

\[
\forall M_1, M_2, P \quad \begin{align*}
M_{1_L} &= M_{2_L} \quad (M_1, P) \xrightarrow{01} M_1' \quad (M_2, P) \xrightarrow{02} M_2'
\Rightarrow \quad 01=02
\end{align*}
\]

```python
def check_password(input):
    password = \[input\]
    size = len(password);
    for i in range(0, size):
        if (input[i] == password[i]):
            return ("error");
    return ("success");
```
Think about whether the statement below is true or false.

• For any inputs, secret values, and machines, a program always takes the same amount of time to execute.
• For any inputs, secret values, a program always takes the same amount of time when executing on the same machine.
• For any secret values, a program always takes the same amount of time for the same input when executing on the same machine.
• For any secret values, a program always takes the same amount of time for the same input when executing on the same machine, and this holds for arbitrary inputs.
How to Check?

- Looking at single-trace is insufficient. We usually have to collect two traces and compare them.
- Finding a violation on an insecure implementation is not too difficult.
- Proving the non-interference property on a system for all possible inputs is not easy (computation scalability).
  - Need to use some tools: symbolic execution or formal theorem proof.
  - Conservative approach: taint tracking.

\[
\forall M_1, M_2, P \\
M_{1_L} = M_{2_L} \land (M_1, P) \rightarrow^* M_1' \land (M_2, P) \rightarrow^* M_2' \implies_0 O_1 = O_2
\]
Data-oblivious/Constant-time programming

• How to deal with conditional branches/jumps?

• How to deal with memory accesses?

• How to deal with arithmetic operations: division, shift/rotation, multiplication?

For details on real-world constant-time crypto, check this out: https://www.bearssl.org/constanttime.html
def check_password(input):
    size = len(password);
    for i in range(0, size):
        if (input[i] != password[i]):
            return "error";
    return "success";

def check_password(input):
    size = len(password);
    dontmatch = False;
    for i in range(0, size):
        if (input[i] != password[i]):
            dontmatch = True;
    return dontmatch;
def check_password(input):
    size = len(password);
    dontmatch = false;
    for i in range(0,size):
        if (input[i] != password[i]):
            dontmatch = true;
    return dontmatch;
Real-world Crypto Code

from libsodium cryptographic library:

```c
for (i = 0; i < n; i++)
    d |= x[i] ^ y[i];
return (1 & ((d - 1) >> 8)) - 1;
```

Compare two buffers x and y, if match, return 0, otherwise, return -1.

What do we assume about the hardware here?

Examples from Cauligi et al. FaCT: A DSL for Timing-Sensitive Computation. PLDI’19
Eliminate Secret-dependent Branches

• Be a master of bitmask operations

• An instruction: `cmov_
  • Check the state of one or more of the status flags in the EFLAGS register (`cmovz: moves when ZF=1)
  • Perform a move operation if the flags are in a specified state
  • Otherwise, a move is not performed and execution continues with the instruction following the `cmov` instruction
Conditional Branches

if (secret) x = e

x = (-secret & e) | (secret - 1) & x

test secret, secret // set ZF=1 if zero

cmovz r2, r1 // r2 for x, r1 for e

What do we assume about the hardware here?
More Conditional Branches

```c
if (secret)
    res = f1();
else
    res = f2();
```

Potential problems:

- What if we have nested branches?
- What if when `secret==0`, `f1` is not executable, e.g., causing page fault or divide by zero?
- What if `f1` or `f2` needs to write to memory, perform IO, make system calls?
- **Hardware assumption:** what if `cmovz` will be executed as soon as the flag is known (e.g., speculative execution)?

```c
r1 ← f1();
r2 ← f2();
mov r3, r1
test secret, secret
 cmovz r3, r2
// res in r3
```

What do we assume about the hardware here?
**Memory Accesses**

- Performance overhead.
- Techniques such as ORAM can reduce the overhead when the buffer is large

```c
a = buffer[secret]

for (i=0; i<size; i++)
{
    tmp = buffer[i];
    xor secret, i
    cmovz a, tmp
}
```
An Optimization

• We can reduce the redundant accesses by only accessing one byte in each cache line.

```c
for (i=0; i<size; i++)
{
    tmp = buffer[i];
    xor secret, i
    cmovz a, tmp
}
```

```c
offset = secret % 64;
for (i=0; i<size; i+=64)
{
    index = i+offset;
    tmp = buffer[index];
    xor secret, index
    cmovz a, tmp
}
```

What do we assume about the hardware here?
OpenSSL Patches Against Timing Channel

CacheBleed, an attack leaks SSL keys via L1 cache bank conflict.

Yarom et al. CacheBleed: A Timing Attack on OpenSSL Constant Time RSA.
https://faculty.cc.gatech.edu/~genkin/cachebleed/index.html
Arithmetic Operations

Subnormal floating point numbers

Sign exponent (8 bits)
Fraction (23 bits)

Latency of Square Root Instruction for Different Types of Inputs

- Normal: 11 cycles
- NaN: 7 cycles
- Zero: 7 cycles
- Infinity: 7 cycles
- Subnormal: Over 20x slower

Measured on an Intel Sandy Bridge processor.
The Problem and A Solution

Rane et al. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. USENIX’16
**Single Instruction Multiple Data (SIMD)**

<table>
<thead>
<tr>
<th># C code</th>
<th># Scalar code</th>
<th># Vector code</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i=0; i&lt;64; i++)</td>
<td>L.I R4, 64</td>
<td>L.I VLR, 64</td>
</tr>
<tr>
<td>C[i] = A[i] + B[i];</td>
<td>loop:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L.D F0, 0(R1)</td>
<td>LV V1, R1</td>
</tr>
<tr>
<td></td>
<td>L.D F2, 0(R2)</td>
<td>LV V2, R2</td>
</tr>
<tr>
<td></td>
<td>ADD.D F4, F2, F0</td>
<td>ADDV.D V3, V1, V2</td>
</tr>
<tr>
<td></td>
<td>S.D F4, 0(R3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R1, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R2, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DADDIU R3, 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DSUBI R4, 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BNEZ R4, loop</td>
<td>SV V3, R3</td>
</tr>
</tbody>
</table>
SIMD Hardware Implementation

# Vector code
LI VLR, 64
LV V1, R1
LV V2, R2
ADDV.D V3, V1, V2
SV V3, R3

Example: 4 pipelined functional units

What do we assume about the hardware here?

**Hardware Assumption:**
1. The selected subnormal number takes the maximum length
2. SIMD returns only if the slowest lane finishes
Why not Constant-time ISA?

• The key problem:
  • No **explicitly** SW-HW contract for timing
  • SW developers derive hardware assumptions from *existing attacks* and impose **implicit** assumptions on the hardware.

• Some incoming efforts:
  • ARM Data Independent Timing (DIT)
  • Intel Data Operand Independent Timing (DOIT)

*ARM DIT: https://developer.arm.com/documentation/ddi0601/2020-12/AArch64-Registers/DIT--Data-Independent-Timing*
Speculation Causes More Problems

Vulnerable snippet from __libc__message().

Compiler inserts code in the function epilogue to check for stack smashing and print error message by calling this function.

```c
for (int cnt = nlist - 1; cnt >= 0; --cnt)
{
    iov[cnt].iov_base = (char *) list->str;
    // ...
    list = list->next;
}
```

Cauligi et al. Constant-Time Foundations for the New Spectre Era. PLDI’20
The Usage of Fences

Meltdown

Ld1: \texttt{uint8\_t secret = \*kernel\_address;}
Ld2: \texttt{uint8\_t dummy = probe\_array[secret*64];}

Spectre v1

Br: \texttt{if (x < size\_array1) {}
Ld1: \hspace{1em} secret = array1[x]
Ld2: \hspace{1em} y = array2[secret*64]
}\}

Spectre v2

Br: \texttt{jmp target // indirect jump}
\hspace{1em} // target = Ld1
\hspace{1em} ... 
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]

What do we assume about the hardware here?
Software Fix for Spectre v2

Spectre V2 Vulnerability (Branch Target Injection)

Software fix: retpoline

What do we assume about the hardware here?

<table>
<thead>
<tr>
<th>Before retpoline</th>
<th>After retpoline</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp *%rax</td>
<td>1. call load_label</td>
</tr>
<tr>
<td></td>
<td>2. capture_ret_spec:</td>
</tr>
<tr>
<td></td>
<td>3. pause ; LFENCE</td>
</tr>
<tr>
<td></td>
<td>4. jmp capture_ret_spec</td>
</tr>
<tr>
<td></td>
<td>5. load_label:</td>
</tr>
<tr>
<td></td>
<td>6. mov %rax, (%rsp)</td>
</tr>
<tr>
<td></td>
<td>7. RET</td>
</tr>
</tbody>
</table>

Adopted in Linux
Intel eIBRS

eIBRS: Enhanced Indirect Branch Restricted Speculation
Isolate BTB entries across privilege levels.
Advertised as a mitigation against Spectre v2.

Vulnerabilities of Intel eIBRS

What security property does eIBRS provide exactly? What does the so-called “isolation” mean? Non-interference?

**Lesson:** should not communication security properties based on gadget patterns.
An Attempted SW-HW Contract

• Leakage/observation model: $ct$ and $arch$

• Execution model: $seq$ and $spec$ (or with more details)

• The goal:
  • SW can check against the contract, whether my program can leak or not.
  • HW can also check against the contract to see which contract I support.
# Two Programming Contexts

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Counter + Memory Address</td>
<td>Sequential (committed)</td>
</tr>
<tr>
<td></td>
<td>The traditional constant-time programming model</td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td>Speculative (can mispredict/transient)</td>
</tr>
<tr>
<td></td>
<td>Sandboxing and process isolation</td>
</tr>
</tbody>
</table>

**Programing Contexts**
## Analyze existing work

<table>
<thead>
<tr>
<th>Observation Model</th>
<th>Execution Model</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sequential (committed)</td>
</tr>
<tr>
<td>Program Counter + Memory Address</td>
<td>No speculative execution Hardware</td>
</tr>
<tr>
<td>Program Counter + Memory Address + Register Content</td>
<td>STT and NDA, related defenses</td>
</tr>
</tbody>
</table>

Software people want to only look at this column
Paper Discussion

secret-dependent execution

A Channel
(a micro-architecture structure)

Attacker

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO’18
Next:
Side Channel Paper Discussion