

# Comp 790-184: Hardware Security and Side-Channels

## Lecture 4: Side-Channel Defenses

February 20, 2024  
Andrew Kwong



THE UNIVERSITY  
*of* NORTH CAROLINA  
*at* CHAPEL HILL

# Outline

---

- How to mitigate side-channel attacks
- Non-interference property
- Constant-time programming
- Constant-time under speculation

# Attack Examples

## Example #1: termination time vulnerability

```
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

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

## Example #3: Meltdown

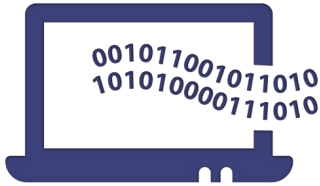
```
.....  
Ld1: uint8_t secret = *kernel_address;  
Ld2: unit8_t dummy = probe_array[secret*64];
```

# Who to blame? Who should fix the problem?



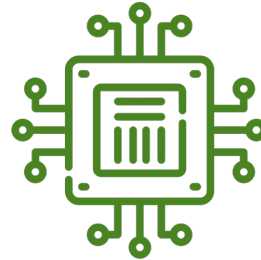
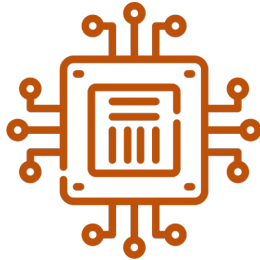
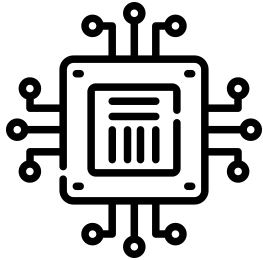
# Software Developer's Problem

---



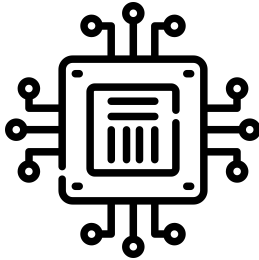
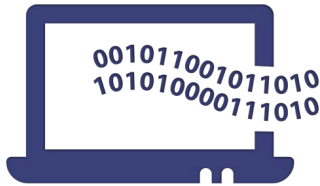
Software developers:

- 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

---

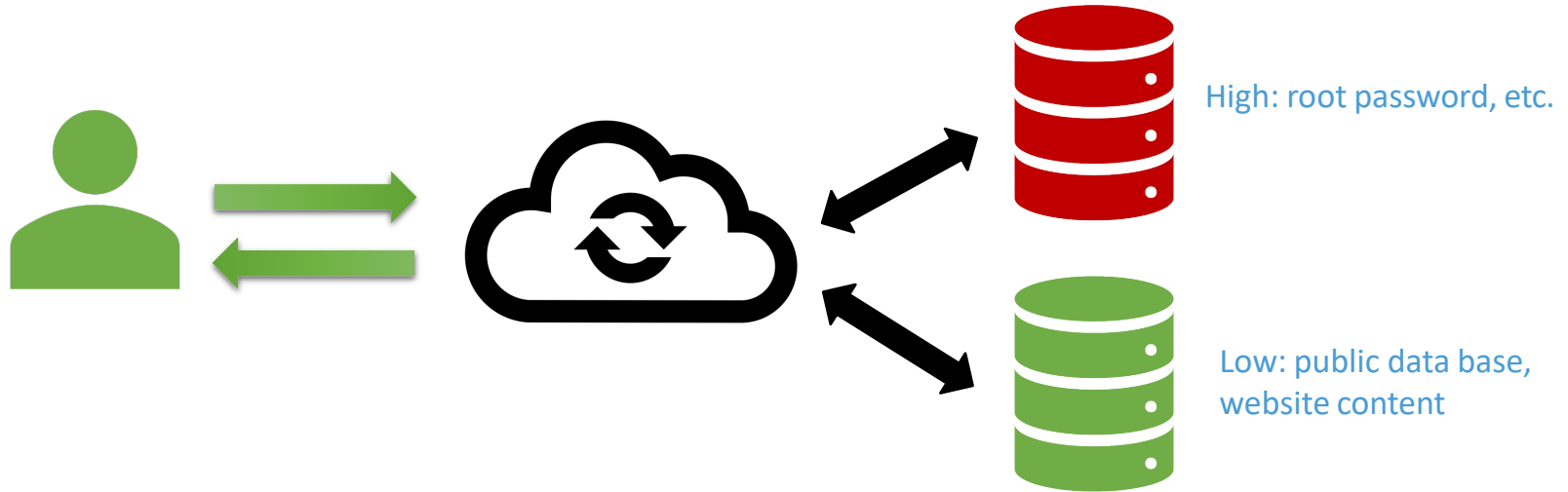
- How to fix it?

```
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  $P$

$$\begin{aligned} & \forall M1, M2, P \\ & M1_L = M2_L \quad \wedge \quad (M1, P) \rightarrow^* M1' \quad \wedge \quad (M2, P) \rightarrow^* M2' \\ & \Rightarrow \quad M1'_L = M2'_L \end{aligned}$$

## Non-Interference for Side Channels

---

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

$$\begin{array}{c} \forall M1, M2, P \\ M1_L = M2_L \wedge (M1, P) \xrightarrow{O1}^* M1' \wedge (M2, P) \xrightarrow{O2}^* M2' \\ \Rightarrow O1=O2 \end{array}$$

What should be included in the observation trace?

# Understand the Property

$$\begin{aligned} & \forall M1, M2, P \\ M1_L = M2_L \wedge (M1, P) \xrightarrow{O1}^* M1' \wedge (M2, P) \xrightarrow{O2}^* M2' \\ \Rightarrow & O1=O2 \end{aligned}$$

Consider input as part of M

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

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

## Constant-Time Programming

---

- 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.

# Data-oblivious/Constant-time programming

---

- How do we deal with conditional branches/jumps?
- How do we deal with memory accesses?
- How do we deal with arithmetic operations: division, shift/rotation, multiplication?

Your Code

Compiler

Hardware

*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;
```



```
def check_password(input):  
    size = len(password);  
    dontmatch = false;  
    for i in range(0,size):  
        dontmatch |= (input [i] != password[i])  
  
    return dontmatch;
```

## Real-world Crypto Code

---

from libsodium cryptographic library:

```
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.



# Eliminate Secret-dependent Branches

---

- 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

## More Conditional Branches

---

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



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

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)?

# Memory Accesses

---

```
a = buffer[secret]
```



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

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

## An Optimization

---

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

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



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

# OpenSSL Patches Against Timing Channel

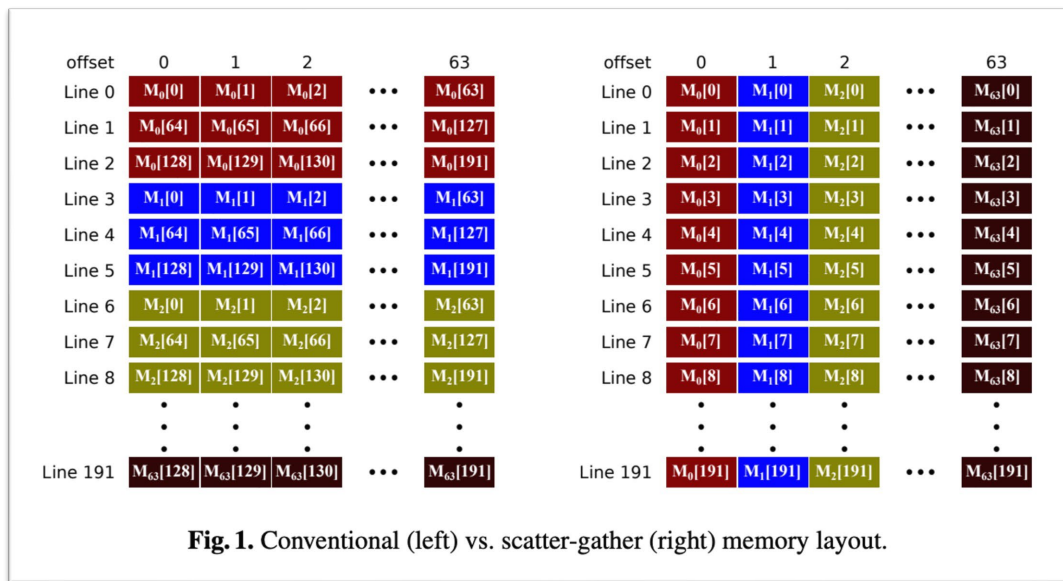
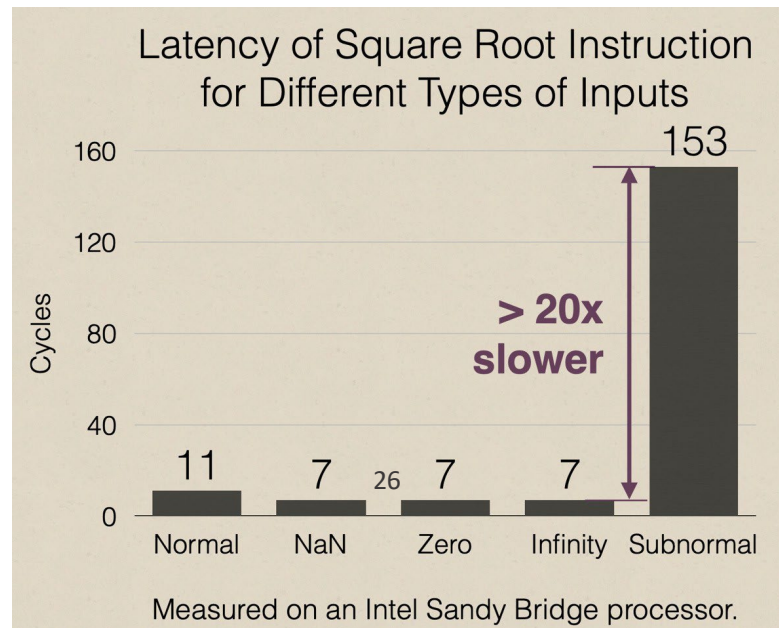
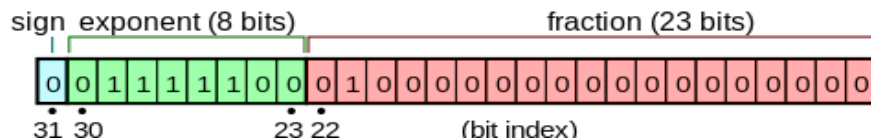


Fig. 1. Conventional (left) vs. scatter-gather (right) memory layout.

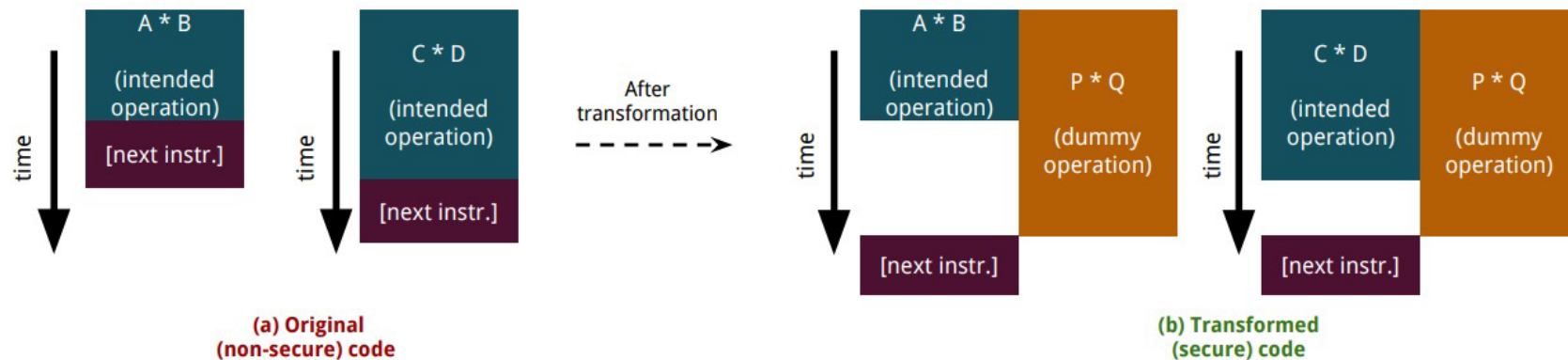
CacheBleed, an attack leaks SSL keys via L1 cache bank conflict.

# Arithmetic Operations

## Subnormal floating point numbers



# The Problem and A Solution



# Constant-time ISA

---

- Some 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>

Intel DOIT: <https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html>



## Constant-time under Speculation

---

- What problems arise?

# The Usage of Fences

---

## Meltdown

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

## Spectre v1

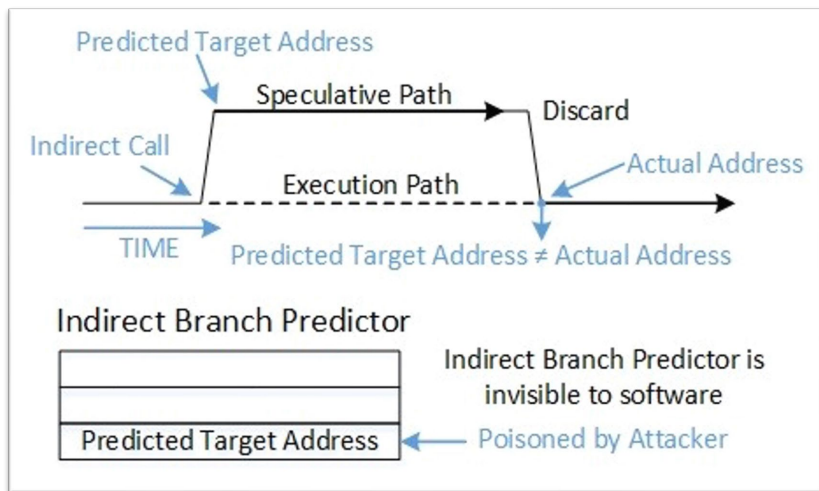
```
Br:  if (x < size_array1) {  
Ld1:      secret = array1[x]  
Ld2:      y = array2[secret*64]  
      }
```

## Spectre v2

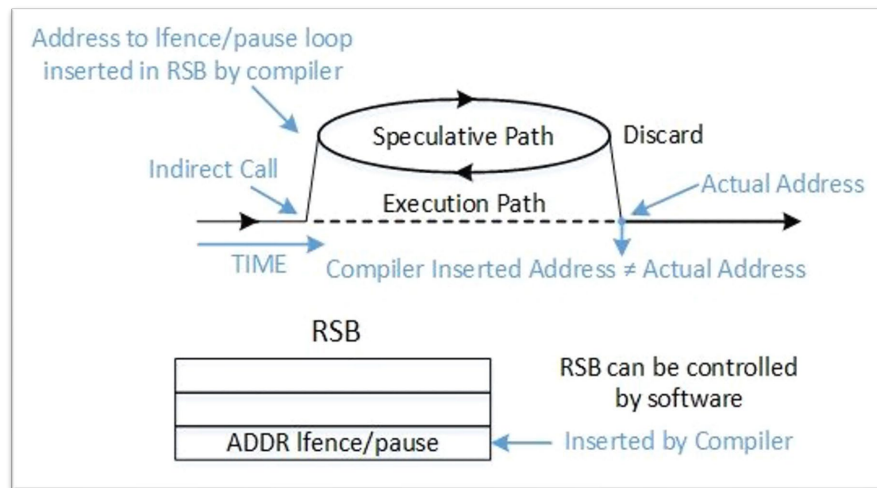
```
Br:  jmp target // indirect jump  
      // target = Ld1  
...  
Ld1: secret = array1[x]  
Ld2: y = array2[secret*4096]
```

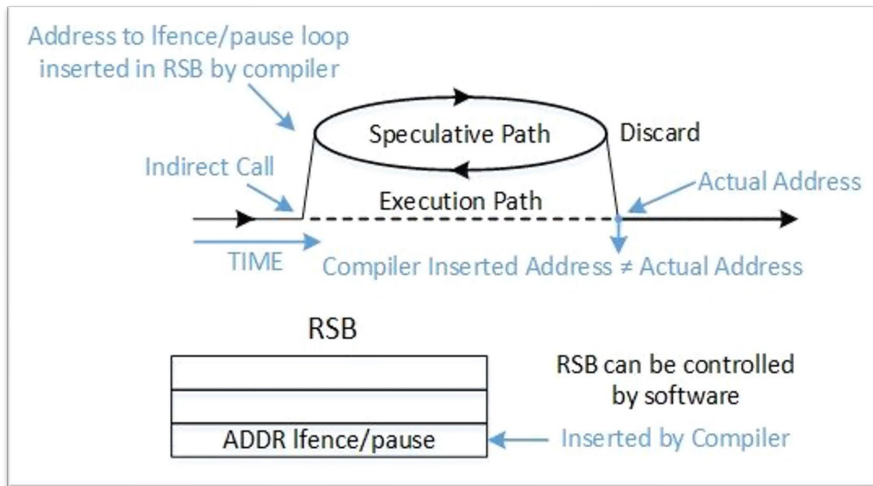
# Software Fix for Spectre v2

## Spectre V2 Vulnerability (Branch Target Injection)



## Software fix: retpoline



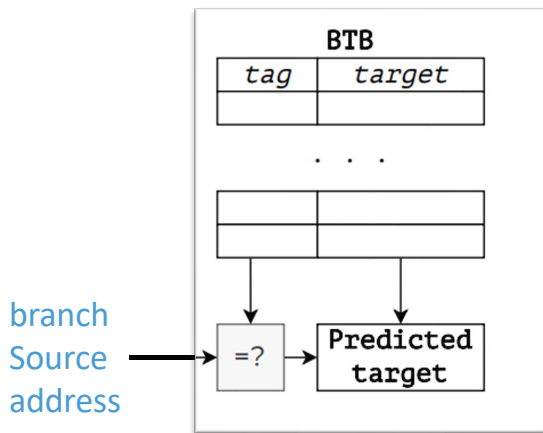


<b>Before retpoline</b>	<code>jmp *%rax</code>
<b>After retpoline</b>	<ol style="list-style-type: none"> <li>1. <code>call load_label</code></li> <li>2. <code>capture_ret_spec:</code></li> <li>3. <code>pause ; LFENCE</code></li> <li>4. <code>jmp capture_ret_spec</code></li> <li>5. <code>load_label:</code></li> <li>6. <code>mov %rax, (%rsp)</code></li> <li>7. <code>RET</code></li> </ol>

Adopted in Linux

# Intel eIBRS

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



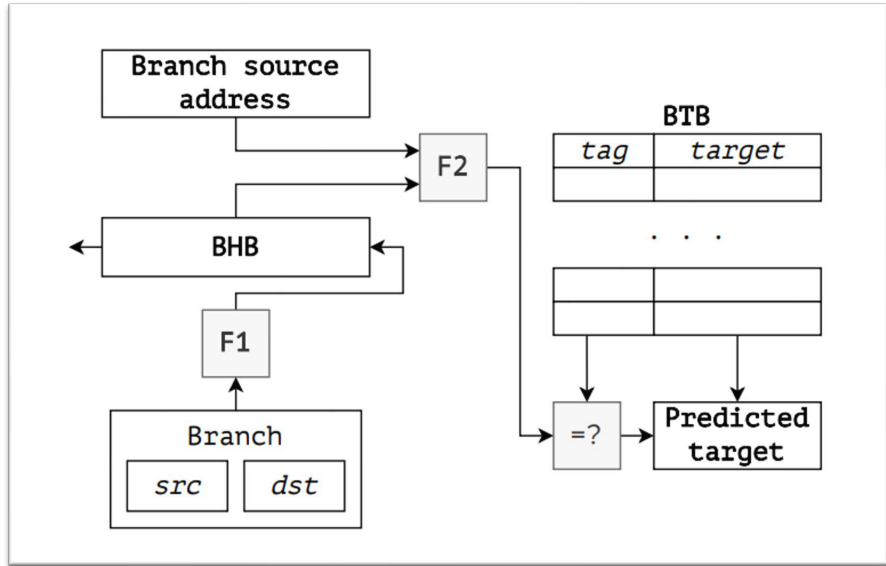
**Listing 3** Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
1 do_syscall_64:
2     ;...
3     mov     rax, [sys_call_table + rax*8]
4     call   __x86_indirect_thunk_rax

1 ;with eIBRS support
2 __x86_indirect_thunk_rax:
3     jmp     rax

1 ;without eIBRS support (retpoline)
2 __x86_indirect_thunk_rax:
3     call   B
4 A:  pause
5     lfence
6     jmp     A
7 B:  mov     [rsp], rax
8     ret
```

# Vulnerabilities of Intel eBRS



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

**Lesson:** should not base communication security properties based on gadget patterns. Instead, want clearly defined contracts



THE UNIVERSITY  
*of* NORTH CAROLINA  
*at* CHAPEL HILL