Software-Hardware Contract for Side Channel Defenses

Mengjia Yan

Spring 2023

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

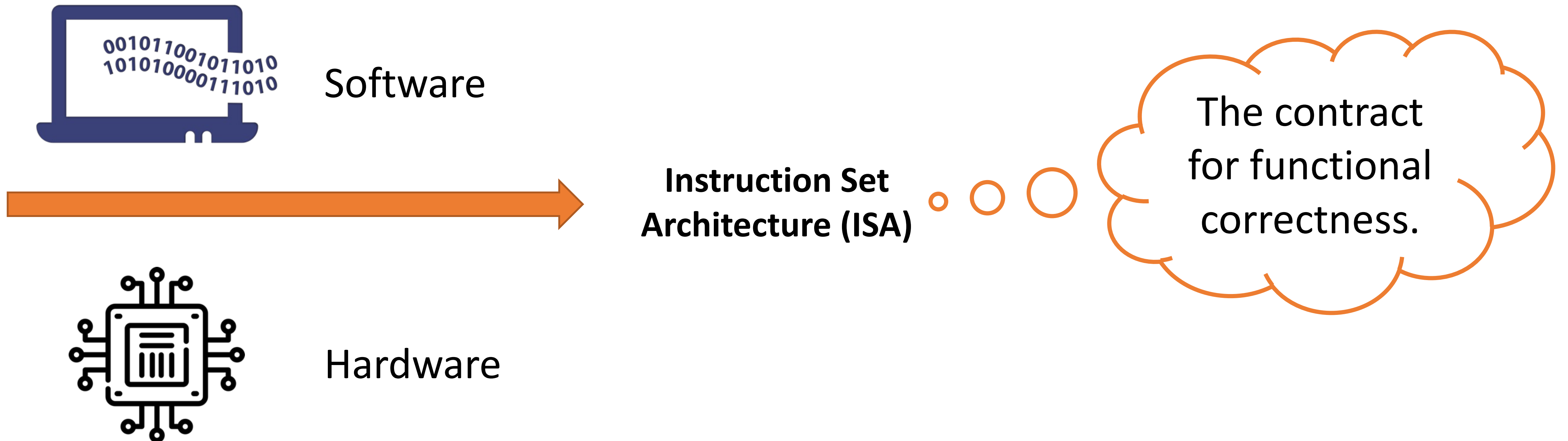
```
for i = n-1 to 0 do
  r = sqr(r)
  r  = r mod n
  if e_i == 1 then
    r = mul(r, b)
    r  = r mod n
  end
end
```

Example #3: Meltdown

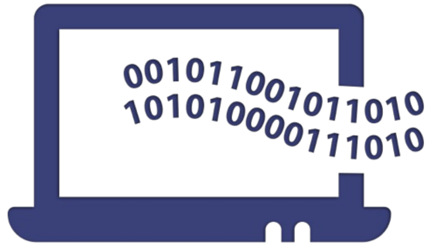
```
……
Ld1: uint8_t secret = *kernel_address;
Ld2: unit8_t dummy = probe_array[secret*64];
```

Who to blame? Who to fix the problem?

Break SW and HW Contract

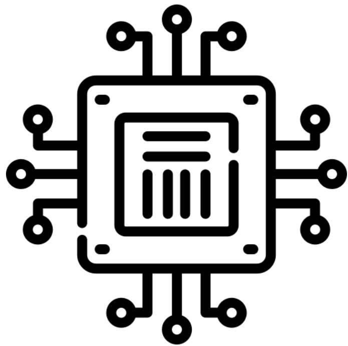


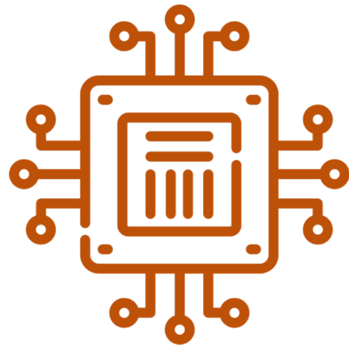
Software Developer's Problem

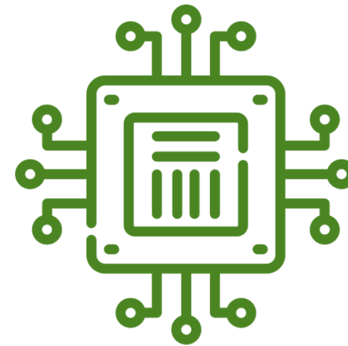


Software developers:

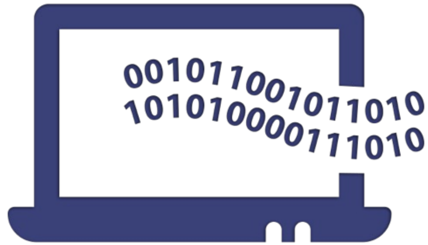
- Need to write software for devices with unknown design details.
- How can I know whether the program is secure running on different devices?

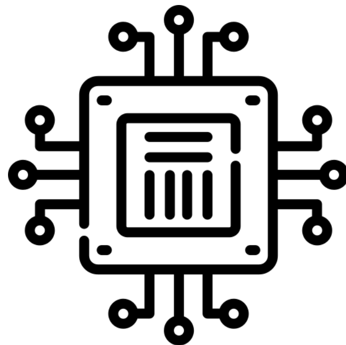






Hardware Designer’s Problem





Hardware designer:

- Need to design processors for arbitrary programs.
- How to describe what kind of programs can run securely on my device?

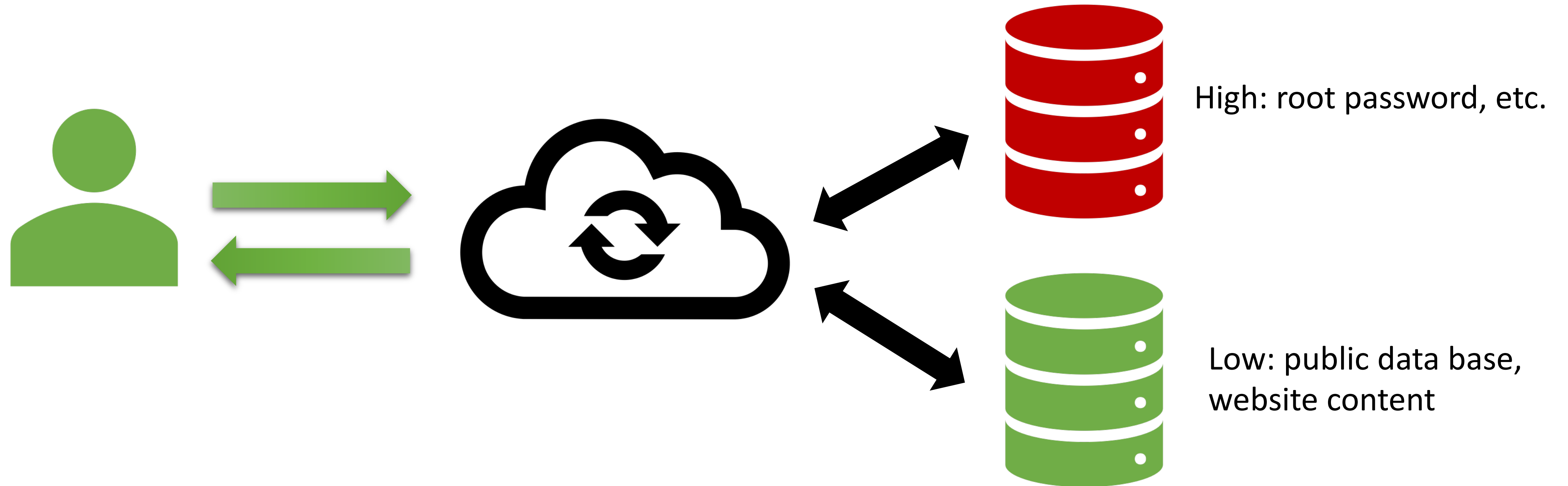
Example: Termination Time Vulnerability

- 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

$$\forall \mathrm{M1}, \mathrm{M2}, \mathrm{P}$$

$$\mathrm{M1_L} = \mathrm{M2_L} \quad \wedge \quad (\mathrm{M1}, \mathrm{P}) \rightarrow^{*} \mathrm{M1'} \quad \wedge \quad (\mathrm{M2}, \mathrm{P}) \rightarrow^{*} \mathrm{M2'}$$

$$\Longrightarrow \quad \mathrm{M1'_L} = \mathrm{M2_L}'$$

Non-Interference for Side Channels

- The definition of noninterference for a deterministic program P

$$\forall\, \mathrm{M1}, \mathrm{M2}, \mathrm{P}$$

$$\mathrm{M1_L} = \mathrm{M2_L} \quad \wedge \quad (\mathrm{M1}, \mathrm{P}) \xrightarrow{\mathbf{O1}}{}^{*} \mathrm{M1}' \quad \wedge \quad (\mathrm{M2}, \mathrm{P}) \xrightarrow{\mathbf{O2}}{}^{*} \mathrm{M2}'$$

$$\Longrightarrow \quad \mathrm{O1} = \mathrm{O2}$$

What should be included in the observation trace?

Understand the Property

$$\forall\, M1, M2, P$$

$$M1_L = M2_L \;\wedge\; (M1, P) \overset{O1}{\rightarrow^*} M1' \;\wedge\; (M2, P) \overset{O2}{\rightarrow^*} M2'$$

$$\Longrightarrow \quad O1 = O2$$

```
def check_password(input):

  size = len(password);

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


  return (“success”);
```

Consider input as part of M

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

Constant-Time Programming

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.

$$\forall\, M1, M2, P$$

$$M1_L = M2_L \;\wedge\; (M1, P) \xrightarrow{O1}{}^{*} M1' \;\wedge\; (M2, P) \xrightarrow{O2}{}^{*} M2'$$

$$\Longrightarrow \quad O1 = O2$$

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?

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;
```

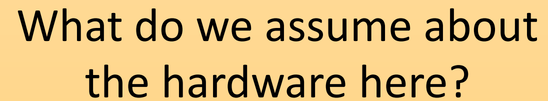
```
def check_password(input):

  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.

Examples from Cauligi et al. FaCT: A DSL for Timing-Sensitive Computation. PLDI'19

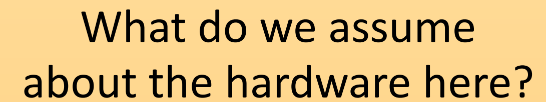
Eliminate Secret-dependent Branches

- 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

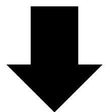
Conditional Branches

```
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
```



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
```

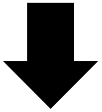
What do we assume about the hardware here?

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]
```



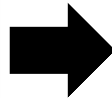
```
for (i=0; i<size; i++)
{
    tmp = buffer[i];
    xor secret, i
    cmovz a, tmp
}
```

- 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 ßaccessing 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
}
```

What do we assume about the hardware here?

OpenSSL Patches Against Timing Channel

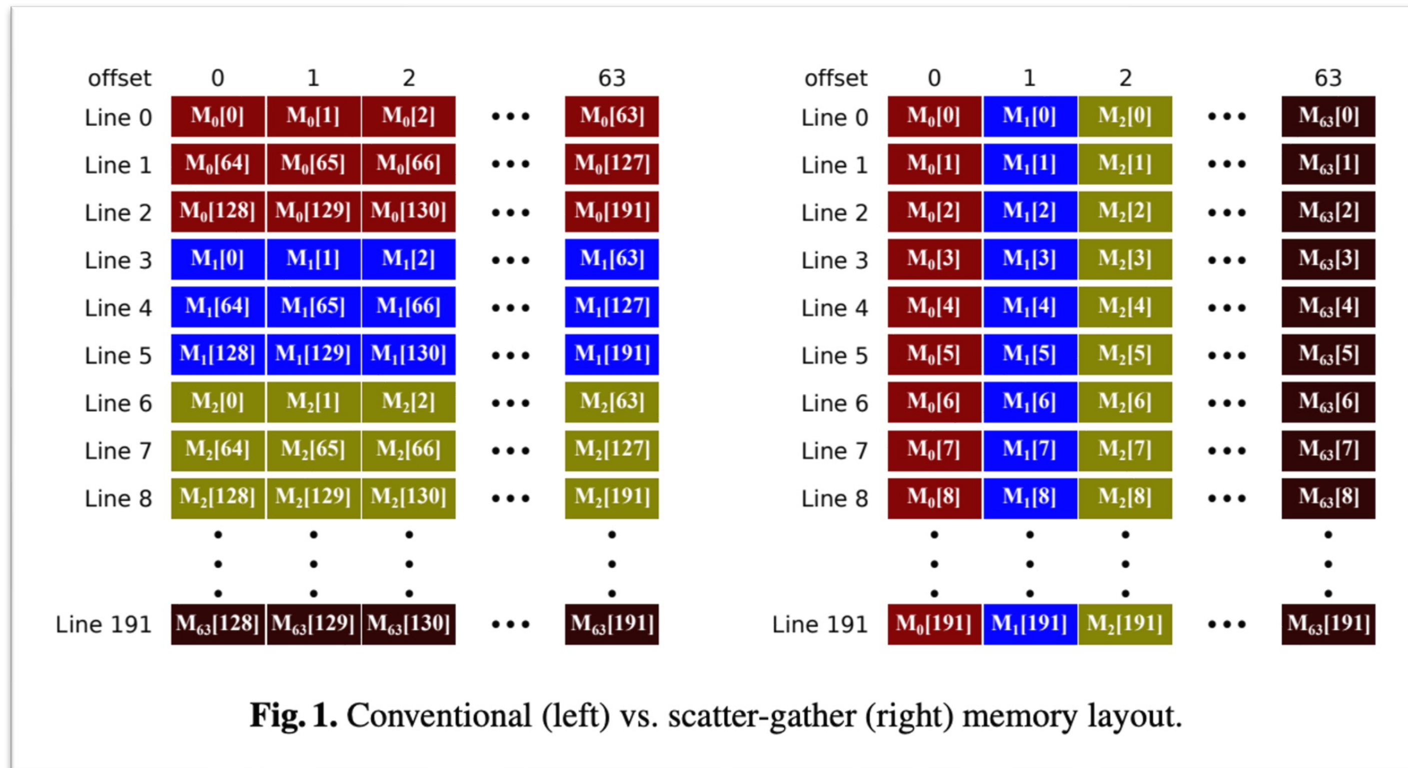


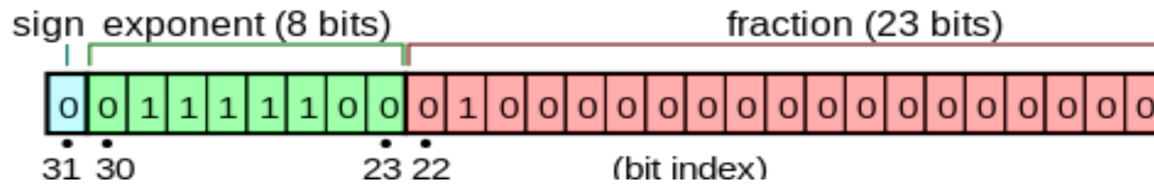
Fig. 1. Conventional (left) vs. scatter-gather (right) memory layout.

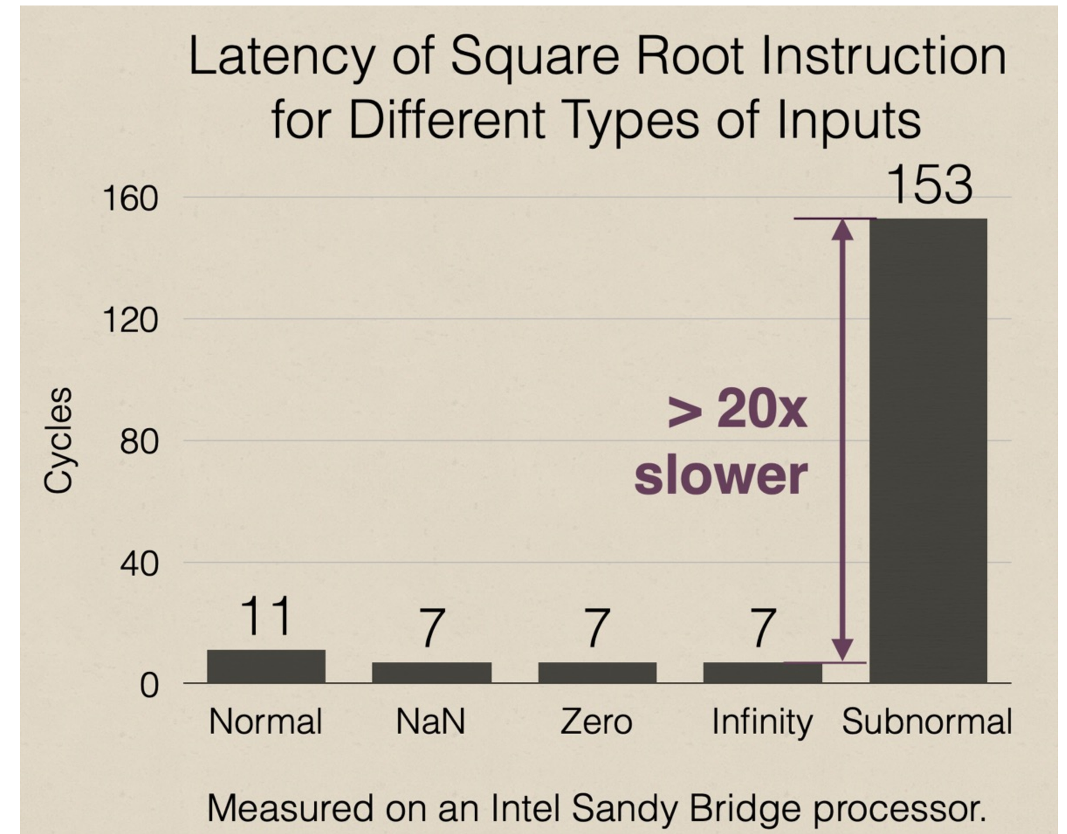
CacheBleed, an attack leaks SSL keys via **L1 cache bank conflict**.

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

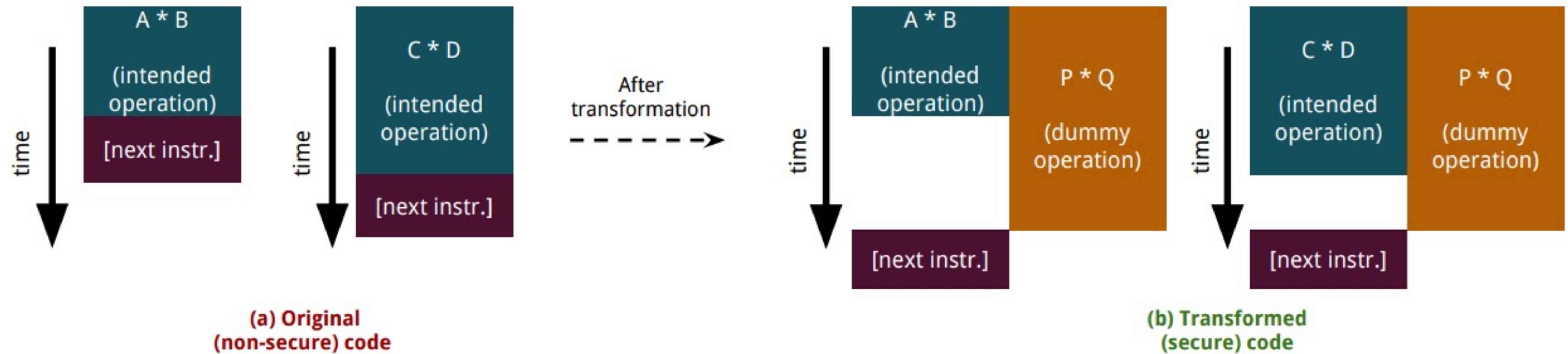
Arithmetic Operations

Subnormal floating point numbers





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)

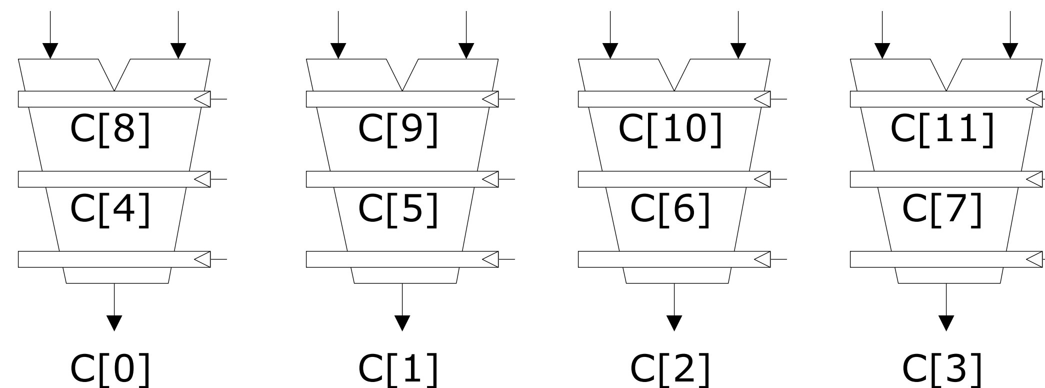
# C code	# Scalar code	# Vector code

`for (i=0; i<64; i++)`
`  C[i] = A[i] + B[i];`
	`  LI R4, 64`
	`loop:`
	`  L.D F0, 0(R1)`
	`  L.D F2, 0(R2)`
	`  ADD.D F4, F2, F0`
	`  S.D F4, 0(R3)`
	`  DADDIU R1, 8`
	`  DADDIU R2, 8`
	`  DADDIU R3, 8`
	`  DSUBIU R4, 1`
	`  BNEZ R4, loop`
		`LI VLR, 64`
		`LV V1, R1`
		`LV V2, R2`
		`ADDV.D V3, V1, V2`
		`SV V3, R3`

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

```
double escort_mul_dp(double x, double y)
{
  const double k_normal_dp = 1.4;
  const double k_subnormal_dp = 2.225e-322;
  double result;
  __asm__ volatile(
    "movdqa %1, %%xmm14;"
    "movdqa %2, %%xmm15;" // save x and y
    "pslldq $8, %1;"
    "pslldq $8, %2;"  // put x, y in lane 2
    "por %3, %1;"
    "por %4, %2;"     // put dummy in lane 1
    "movdqa %2, %0;"  // adjust destination reg
    "mulpd %1, %0;"   // Perform an SIMD multiply
    "psrldq $8, %0;"  // remove lane 1 result
    "movdqa %%xmm14, %1;"
    "movdqa %%xmm15, %2;"  // restore x and y
    : "=x" (result), "+x" (x), "+x" (y)
    : "x" (k_subnormal_dp), "x" (k_normal_dp)
    : "xmm15", "xmm14");

  return result;
}
```

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
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

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.

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

What do we assume
about the hardware here?

Spectre v1

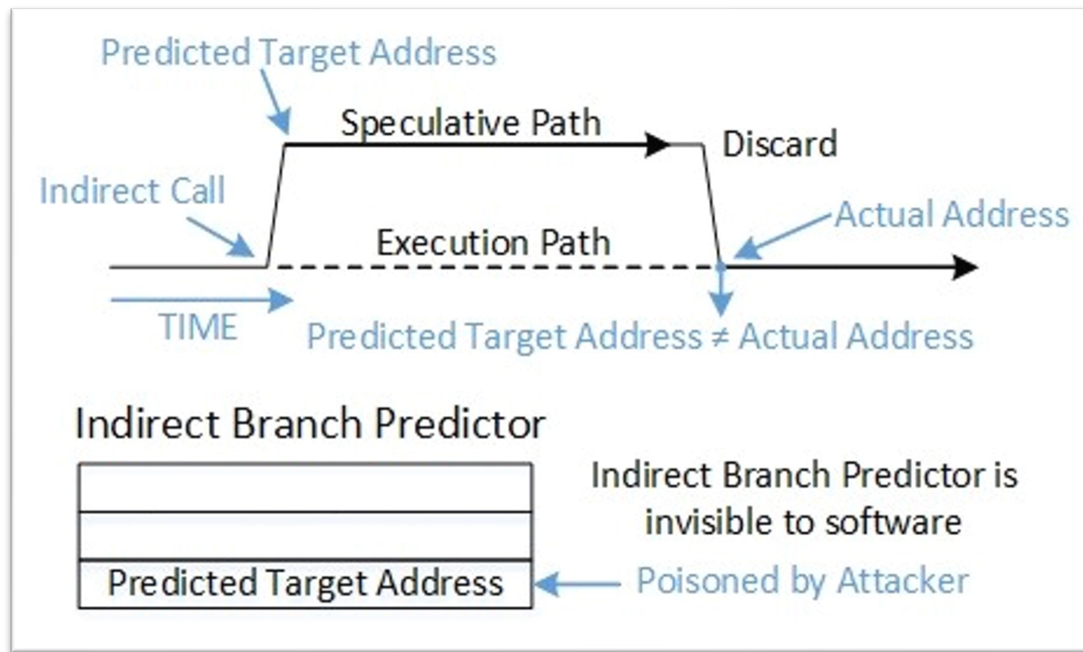
```
Br:  if (x < size_array1) {
Ld1:      secret = array1[x]
Ld2:     y = array2[secret*64]
     }
```

Spectre v2

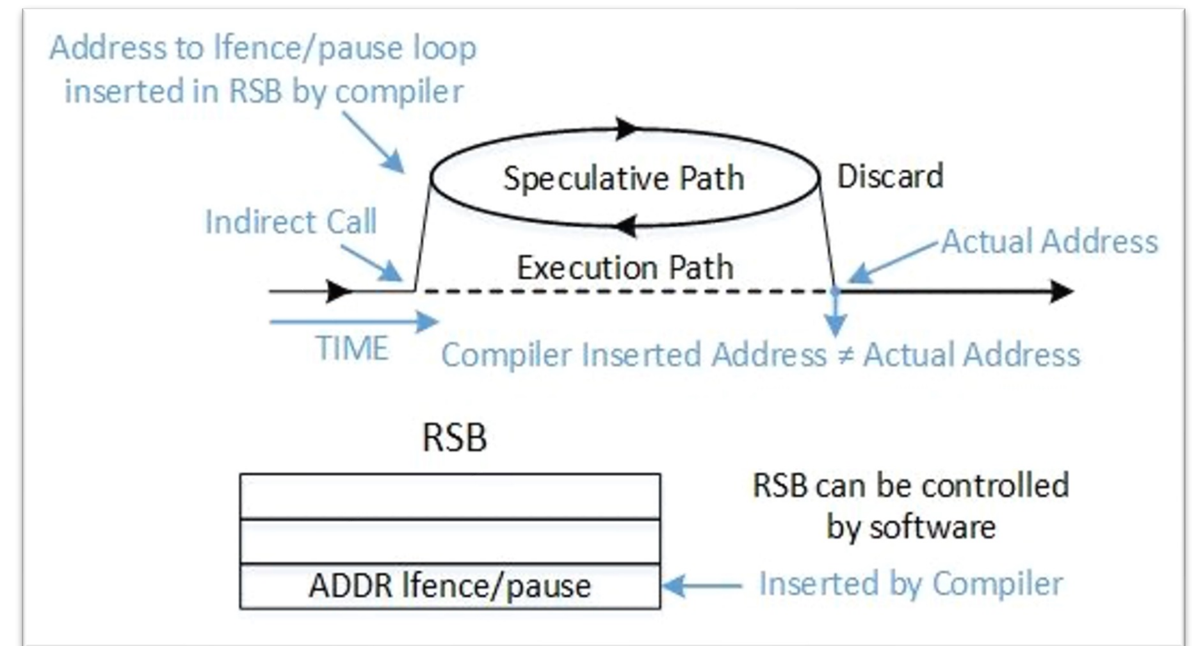
```
Br: jmp target // indirect jump
               // target = Ld1
…
Ld1: secret = array1[x]
Ld2: y = array2[secret*4096]
```

Software Fix for Spectre v2

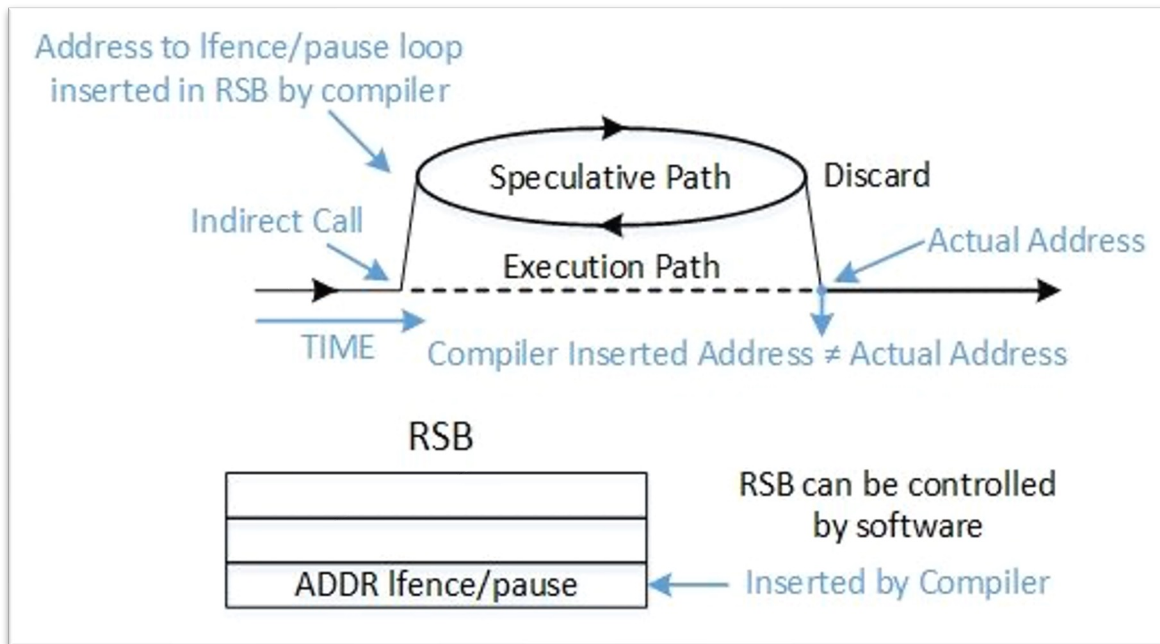
Spectre V2 Vulnerability (Branch Target Injection)



Software fix: retpoline



https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/retpoline-branch-target-injection-mitigation.html



What do we assume about the hardware here?

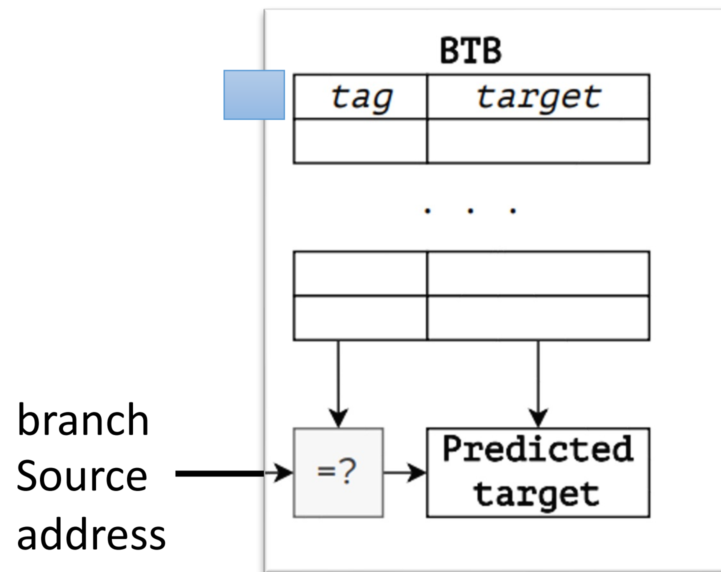
Before retpoline	`jmp *%rax`

After retpoline	`1. call load_label`
	`2.capture_ret_spec:`
	`3. pause ; LFENCE`
	`4. jmp capture_ret_spec`
	`5.load_label:`
	`6. mov %rax, (%rsp)`
	`7. RET`

Adopted in Linux

Intel eIBRS

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

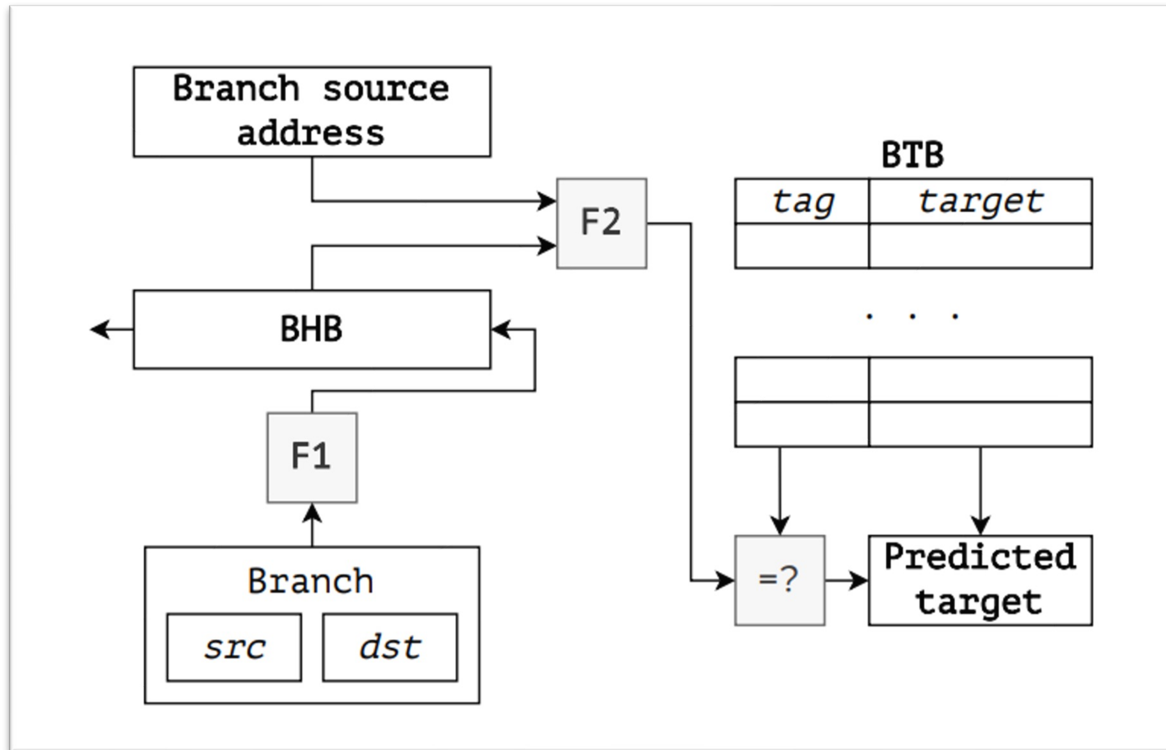


Listing 3 Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
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
```

Barberis et al. Branch History Injection: On the Effectiveness of Hardware Mitigations Against Cross-Privilege Spectre-v2 Attacks. USENIX'22 https://www.vusec.net/projects/bhi-spectre-bhb/

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.

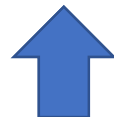
An Attempted SW-HW Contract

- 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.

Guarnieri et al. Hardware-Software Contracts for Secure Speculation. SP'19

Two Programming Contexts

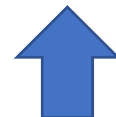
		Execution Model	
		Sequential (committed)	Speculative (can mispredict/transient)
Observation Model	Program Counter + Memory Address	The traditional constant-time programming model	
	Program Counter + Memory Address + Register Content	Sandboxing and process isolation	



Programing Contexts

Analyze existing work

		Execution Model	
		Sequential (committed)	Speculative (can mispredict/transient)
Observation Model	Program Counter + Memory Address	No speculative execution Hardware	Speculative Baseline
	Program Counter + Memory Address + Register Content	STT and NDA, related defenses	



Software people want to only look at this column

Paper Discussion

Kiriansky et al. DAWG: a defense against cache timing attacks in speculative execution processors. MICRO'18

Next:
Side Channel Paper Discussion