Software-Hardware Contract for Side Channel Defenses

Mengjia Yan

Spring 2024

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

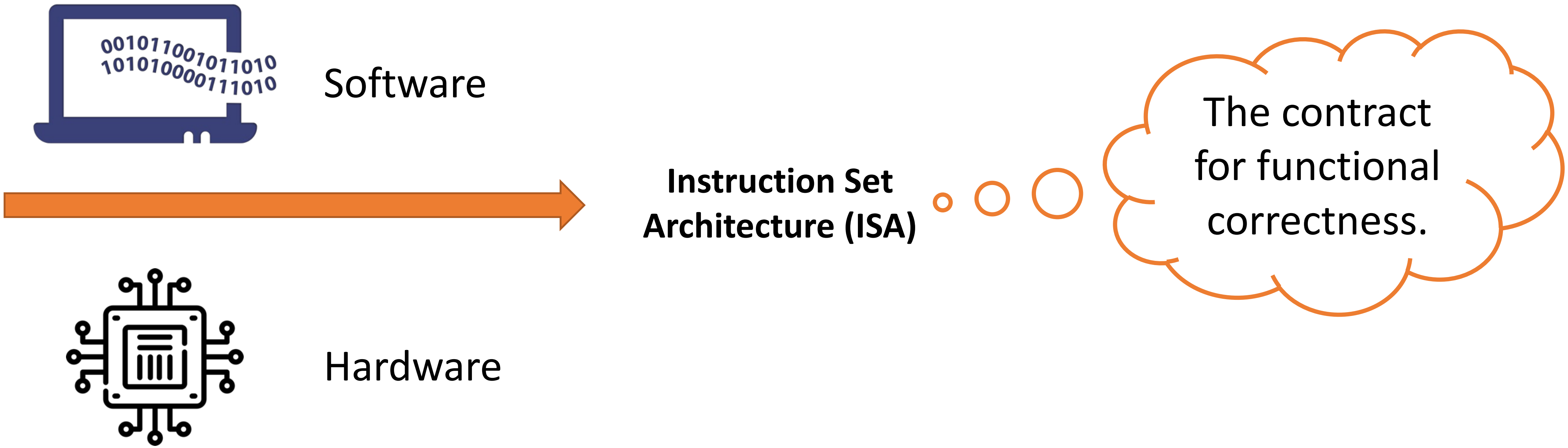
```
for i = n-1 to 0 do
  r = sqr(r)
  r  = r mod n
  if $e_i$ == 1 then
    r = mul(r, b)
    r  = r mod n
  end
end
```

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?

These Attacks Break SW-HW Contract



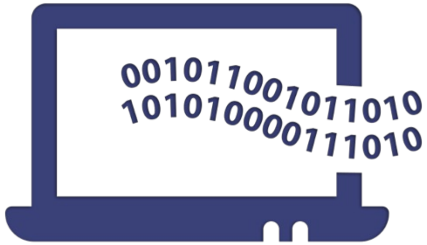
Software

Hardware

Instruction Set Architecture (ISA)

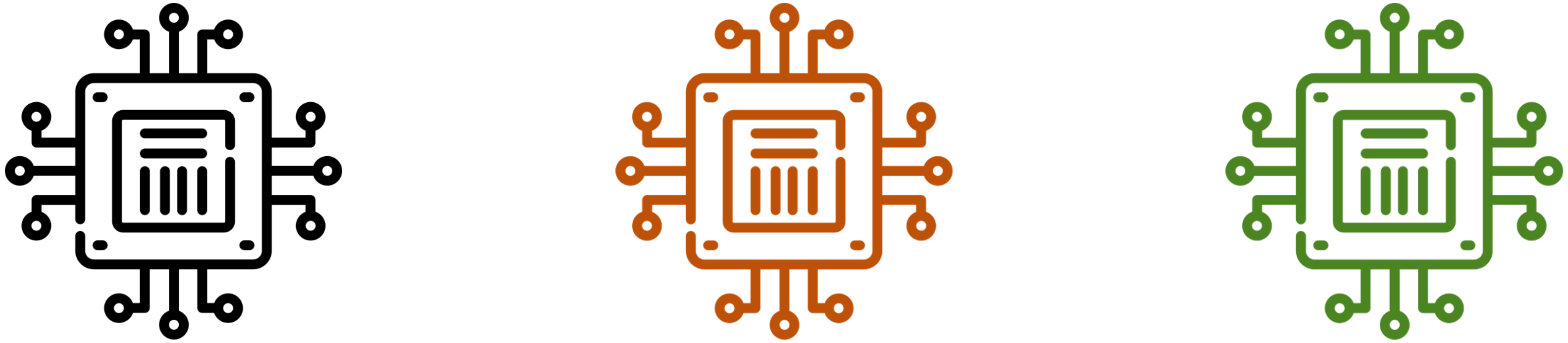
The contract for functional correctness.

Software Developer's Problem

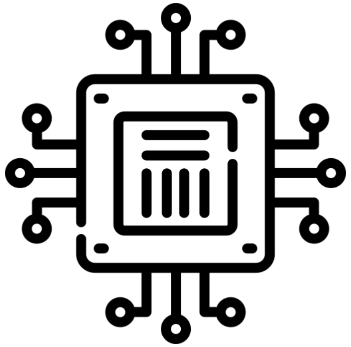


Software developers need to write software for devices with unknown design details.

>> How can I know whether the program is secure running on different devices?



Hardware Designer’s Problem



Hardware designers need to design processors for arbitrary programs.

>> How to describe what kind of programs can run securely on my device?

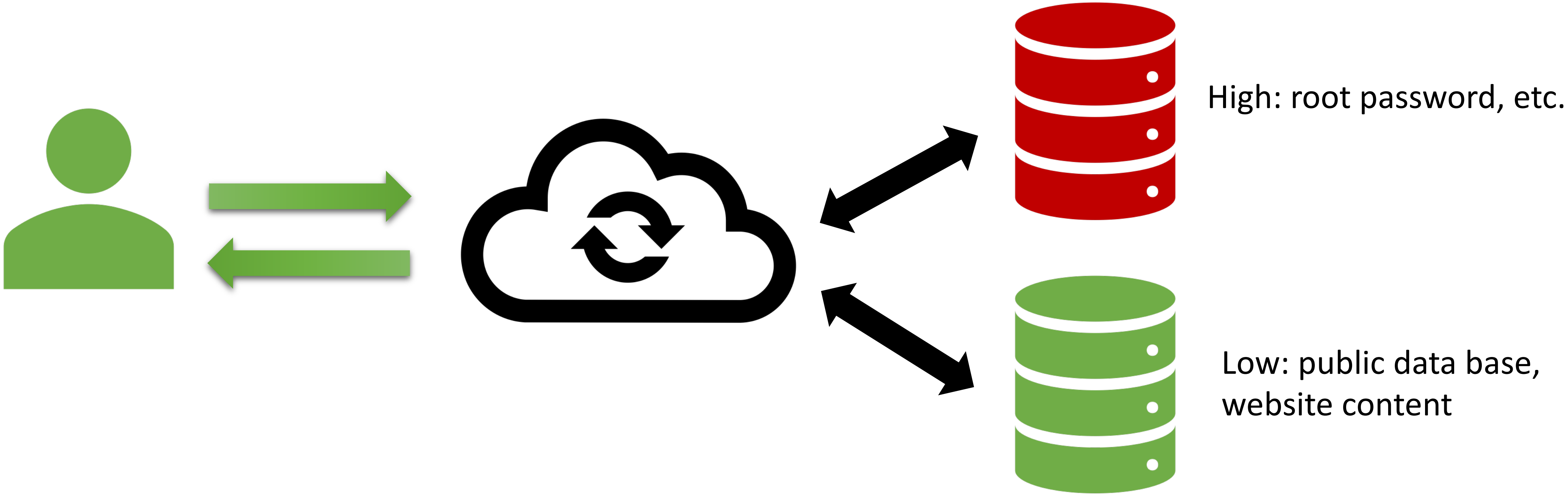
Example: Termination Time Vulnerability

- 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


High: root password, etc.
Low: public data base, website content

- 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

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

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

$$\Longrightarrow \quad \mathrm{M1}_{\mathrm{L}}' = \mathrm{M2_L}'$$

Non-Interference for Side Channels

- The definition of noninterference for a deterministic program P

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

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

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

What should be included in the observation trace?

Instruction completion time
Addresses issued to the memory systems (for both data and instruction)

Understand the Property

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

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

$$\Longrightarrow \quad O1 = O2$$

```
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 public inputs, secret values, and machines, a program always takes the same amount of time to execute.
- For any public 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 public 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 public inputs.

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;
```

What do we assume about the hardware here?

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

Another Example

From the “donna” Curve25519 implementation

```
for (i = 0; i < 5; ++i)
{
    if (swap) {
        tmp = a[i];
        a[i] = b[i];
        b[i] = tmp;
    }
}
```

```
for (i = 0; i < 5; ++i) {
    const limb x = swap & (a[i] ^ b[i]);
    a[i] ^= x;
    b[i] ^= x;
}
```

swap is a mask, either 0 or 0xFFFFFFFF

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 (as if a NOP) and execution continues with the instruction following the `cmov` instruction

Conditional Branches

```
if (secret) x = e

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

What do we assume about the hardware here? (Hint: there are two.)

```
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
```

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?

Memory Accesses

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

⬇

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

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

An Optimization

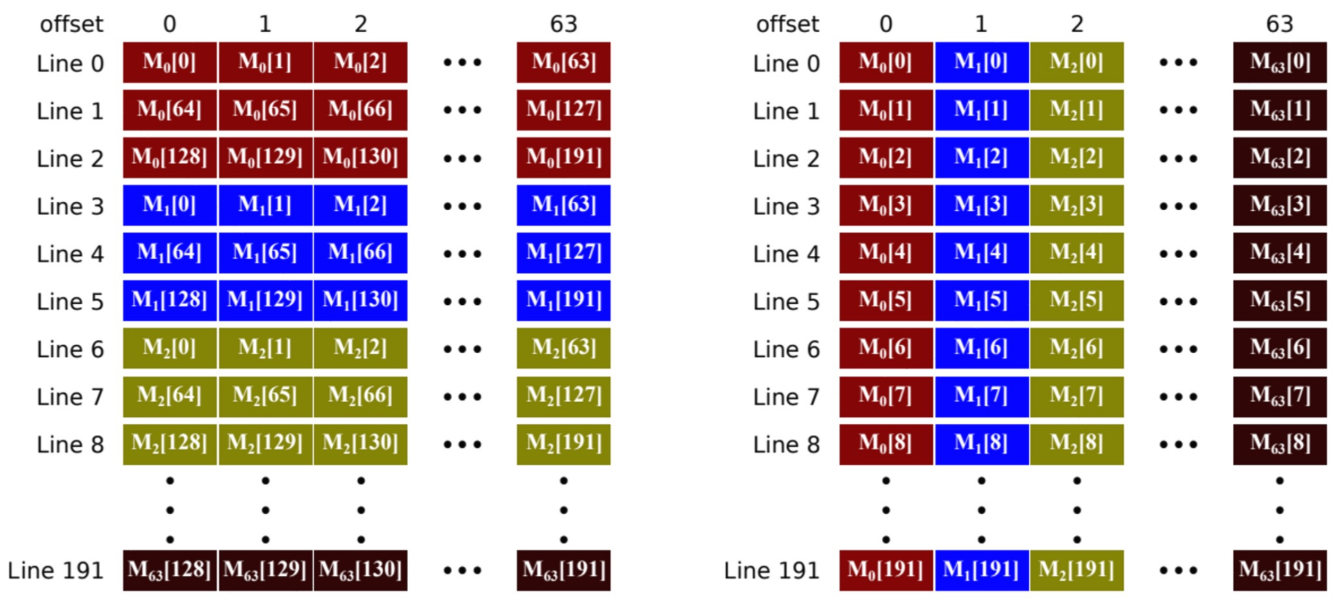
- We can reduce the redundant accesses by only accessing one byte in each cache line.

```
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
}
```

What do we assume about the hardware here?

OpenSSL Patches Against Timing Channel



offset	0	1	2		63
Line 0	$M_0[0]$	$M_0[1]$	$M_0[2]$	•••	$M_0[63]$
Line 1	$M_0[64]$	$M_0[65]$	$M_0[66]$	•••	$M_0[127]$
Line 2	$M_0[128]$	$M_0[129]$	$M_0[130]$	•••	$M_0[191]$
Line 3	$M_1[0]$	$M_1[1]$	$M_1[2]$	•••	$M_1[63]$
Line 4	$M_1[64]$	$M_1[65]$	$M_1[66]$	•••	$M_1[127]$
Line 5	$M_1[128]$	$M_1[129]$	$M_1[130]$	•••	$M_1[191]$
Line 6	$M_2[0]$	$M_2[1]$	$M_2[2]$	•••	$M_2[63]$
Line 7	$M_2[64]$	$M_2[65]$	$M_2[66]$	•••	$M_2[127]$
Line 8	$M_2[128]$	$M_2[129]$	$M_2[130]$	•••	$M_2[191]$
	⋮	⋮	⋮		⋮
Line 191	$M_{63}[128]$	$M_{63}[129]$	$M_{63}[130]$	•••	$M_{63}[191]$

offset	0	1	2		63
Line 0	$M_0[0]$	$M_1[0]$	$M_2[0]$	•••	$M_{63}[0]$
Line 1	$M_0[1]$	$M_1[1]$	$M_2[1]$	•••	$M_{63}[1]$
Line 2	$M_0[2]$	$M_1[2]$	$M_2[2]$	•••	$M_{63}[2]$
Line 3	$M_0[3]$	$M_1[3]$	$M_2[3]$	•••	$M_{63}[3]$
Line 4	$M_0[4]$	$M_1[4]$	$M_2[4]$	•••	$M_{63}[4]$
Line 5	$M_0[5]$	$M_1[5]$	$M_2[5]$	•••	$M_{63}[5]$
Line 6	$M_0[6]$	$M_1[6]$	$M_2[6]$	•••	$M_{63}[6]$
Line 7	$M_0[7]$	$M_1[7]$	$M_2[7]$	•••	$M_{63}[7]$
Line 8	$M_0[8]$	$M_1[8]$	$M_2[8]$	•••	$M_{63}[8]$
	⋮	⋮	⋮		⋮
Line 191	$M_0[191]$	$M_1[191]$	$M_2[191]$	•••	$M_{63}[191]$

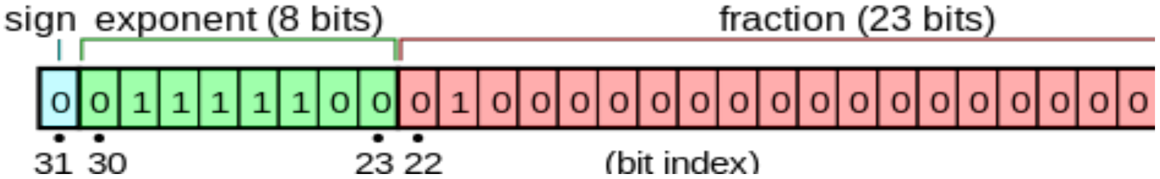
Fig. 1. Conventional (left) vs. scatter-gather (right) memory layout.

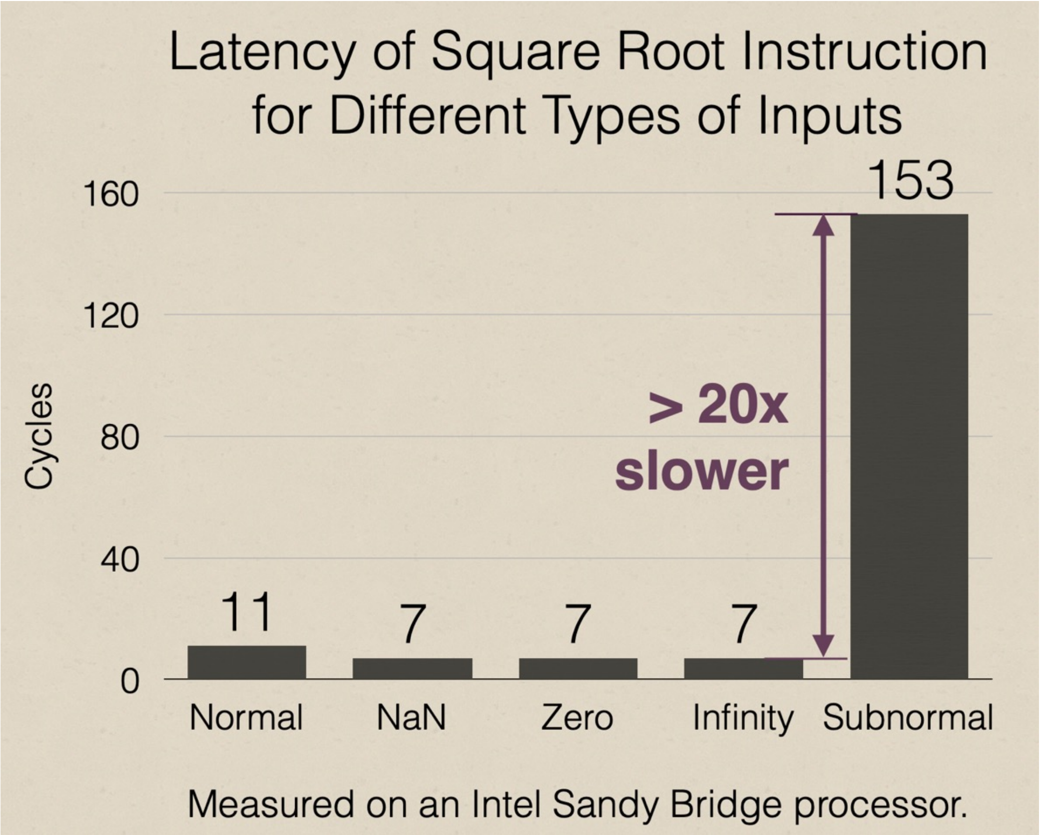
CacheBleed, an attack leaks SSL keys via **L1 cache bank conflict**.

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

Arithmetic Operations

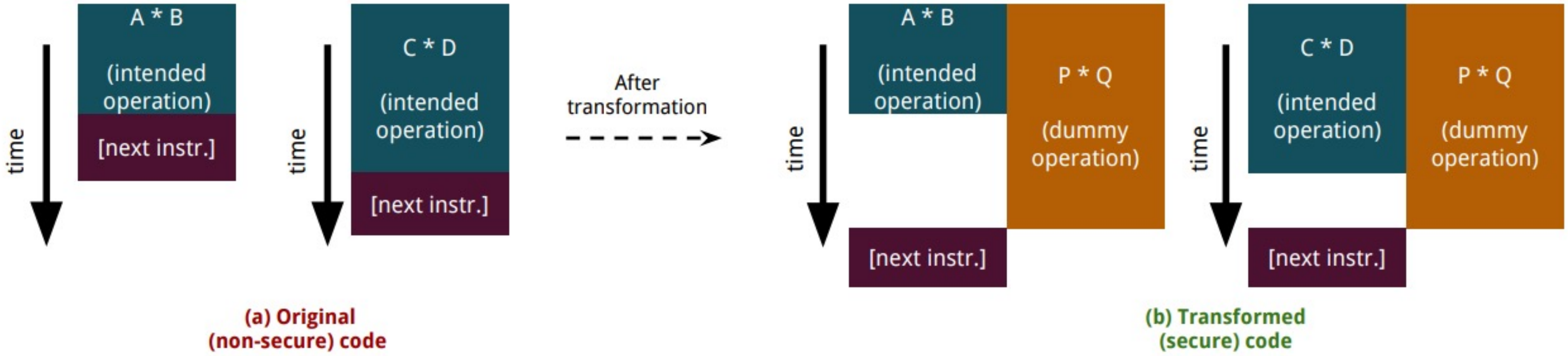
Subnormal floating point numbers




Latency of Square Root Instruction for Different Types of Inputs

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)

```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

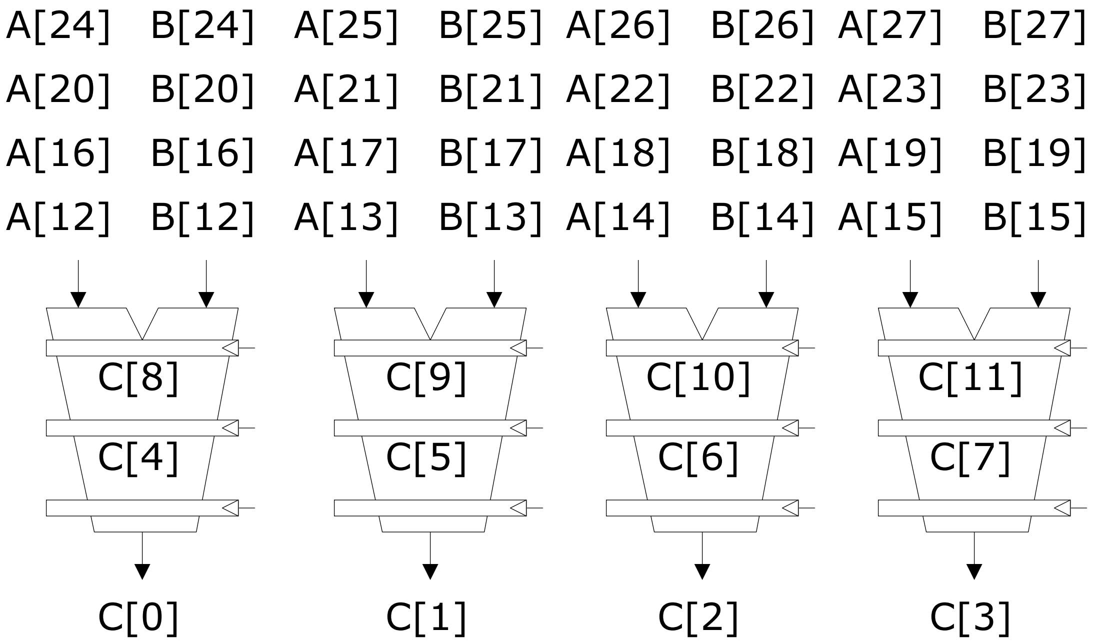
```
# Scalar code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  ADD.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector code
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 //length
LV V1, R1  // vec 1
LV V2, R2  // vec 2
ADDV.D V3, V1, V2
SV V3, R3
```

Example: 4 pipelined functional units

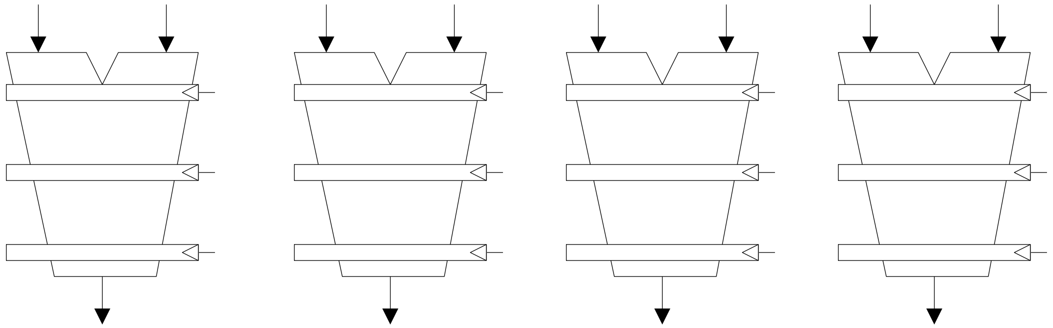

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

Make Floating-Point Constant Time

What do we assume
about the hardware here?

Parameters for
the actual
computation

Selected
subnormal
numbers



Hardware Assumption:

1. The selected subnormal number takes the maximum length
2. SIMD returns only if the slowest lane finishes

How shall we proceed?

- 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

So far, we have not discussed how to deal with speculation…

😉