Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors

Bart Coppens, Ingrid Verbauwhede,
Koen De Bosschere, Bjorn De Sutter

Presented by Lili Chen, Xinyun Jiang, Shengyi Qian, Xingyu Wang

Overview

Background

Methodology

Experiments & Results

Discussions

Side-Channel Attacks

A side-channel attack is any attack based on **information gained from the computer system**, rather than **weaknesses in the algorithm itself** (e.g. software bugs).

https://en.wikipedia.org/wiki/Side-channel_attack

Timing-Based Side-Channel Attacks

$$c = b^e \mod m = b^{e_0} \cdot b^{e_1} \cdots \mod m$$ (e0, e1,... are binary bits)

```
//Modular Exponentiation Algorithm.
int result = 1; //initialization
int i = length of exponent in binary - 1;
int b = base;
do {
    result = (result*result) % n;
    if ((exponent>>i) & 1)
        result = (result*b) % n;
    i--;
}
while (i >= 0);
```

```
//Initialization
//Exponent = 3 (11 in binary)
    int i = 1;
    int result = 1;
//First Iteration:
//First bit of exponent is 1
    result = 1 x 1 % n;
    result = result x b % n;
    i = 0;
//Second Iteration:
//Second bit of exponent is 1
    result = b x b % n;
    result = result x b % n;//b^3 % n
    i = -1;
//Exit
```

Timing-Based Side-Channel Attacks

$$c = b^e \mod m = b^{e_0} \cdot b^{e_1} \cdots \mod m$$ (e0, e1,... are binary bits)

```
//Modular Exponentiation Algorithm.
int result = 1; //initialization
int i = length of exponent in binary - 1;
int b = base;
do {
      result = (result*result) % n;
      if ((exponent>>i) & 1)
            result = (result*b) % n;
      i--;
}
while (i >= 0);
```

Control Flow

```
//Modular Exponentiation Algorithm.
int result = 1; //initialization
int i = length of exponent in binary - 1;
int b = base;
do {
     result = (result*result) % n;
     if ((exponent>>i) & 1)
          result = (result*b) % n;
     i--;
}
while (i >= 0);
```

The private key!

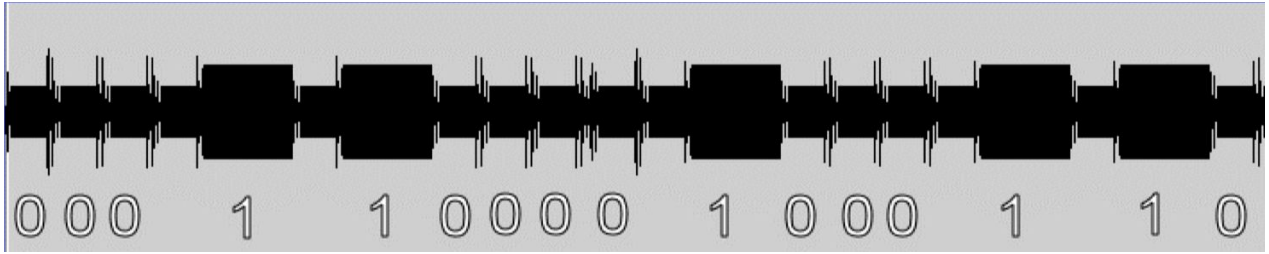


Image Credit: Manuel Charlemagne

Can we fix it using a Compiler back-end Approach?

Methodology

1. If-conversion -- replace branches with predicates
    a. Handle exceptions (division and memory)
    b. Handle function call
2. Variable-latency instructions (division) elimination

If-Conversion -- Division

```
if (c) {
    d = x/y;
} else {
    b = 10;
}
```

Predicate	Instruction
	tmp_y = y;
if (~c)	tmp_y = 1;
	tmp_d = x / tmp_y;
	tmp_b = 10;
if (c)	d = tmp_d;
if (~c)	b = tmp_b;

If-Conversion -- Memory Op. (load)

```
if (a != NULL) {
    b = *a;
}
```

Denote (a!=NULL) as c

Predicate	Instruction
	`tmp_a = a;`
`if (~c)`	`tmp_a = dummy_location;`
	`tmp_b = *tmp_a;`
`if (c)`	`b = tmp_b;`

If-Conversion -- Memory Op. (store)

```
if (c) {
    *a = 10;
}
```

Predicate	Instruction
	`tmp_a = a;`
`if (~c)`	`tmp_a = dummy_location;`
	`*tmp_a = 10;`

If-Conversion -- Function Call

```
void f(int x) {
    *a = x;
}

...
if (c) {
    f(10);
}
```

```
void f(int x, int c) {
    if (c) {
        *a = x;
    }
}

...
f(10, c);
```

Then apply normal if-conversion to the new function f

Caution: If any call to the function is key-independent, use original function to have less overhead.

Solution to Variable Latency Instructions

1. Add compensation code
    - Complex to determine number of cycles it takes for one certain division

2. Avoid variable latency instructions
    - Significant performance overhead workaround (Implemented in this paper)

Division elimination

```
//Modular Exponentiation Algorithm.
int result = 1; //initialization
int i = length of exponent in binary - 1;
int b = base;
do {
    result = (result*result) % n;
    if ((exponent>>i) & 1)
        result = (result*b) % n;
    i--;
}
while (i >= 0);
```

Division

Significant Overhead!

Addition, Subtraction, Shift, Multiplication

```
#Implementation of Modulus Operation
function modulus(x,y){
    var m = Math.floor(x / y);
    var r = m * y;
    return x - r;
}
```

Experiments

A variety of microbenchmarks are tested:

1. f1, f2, f3, f4 are simple if-condition/nested if-condition codes
    - Tests for Efficiency (overhead)
2. Memread1, Memread2 for memory accesses test
    - Tests for Efficiency (overhead)
3. Modexp32, Modexp64 for modular exponentiation test
    - Tests for Effectiveness (leakage)

Results: Effectiveness

```
//Modular Exponentiation Algorithm.
int result = 1; //initialization
int i = length of exponent in binary - 1;
int b = base;
do {
    result = (result*result) % n;
    if ((exponent>>i) & 1)
        result = (result*b) % n;
    i--;
}
while (i >= 0);
```

	original				if-converted				if-converted + div elimination			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	0.785	1.377	1.027	1.223	1.504	1.535	1.524	1.515	26.473	26.473	26.474	26.474
modexp64	0.911	1.816	1.354	1.405	1.847	1.897	1.871	1.877	21.109	21.110	21.109	21.109

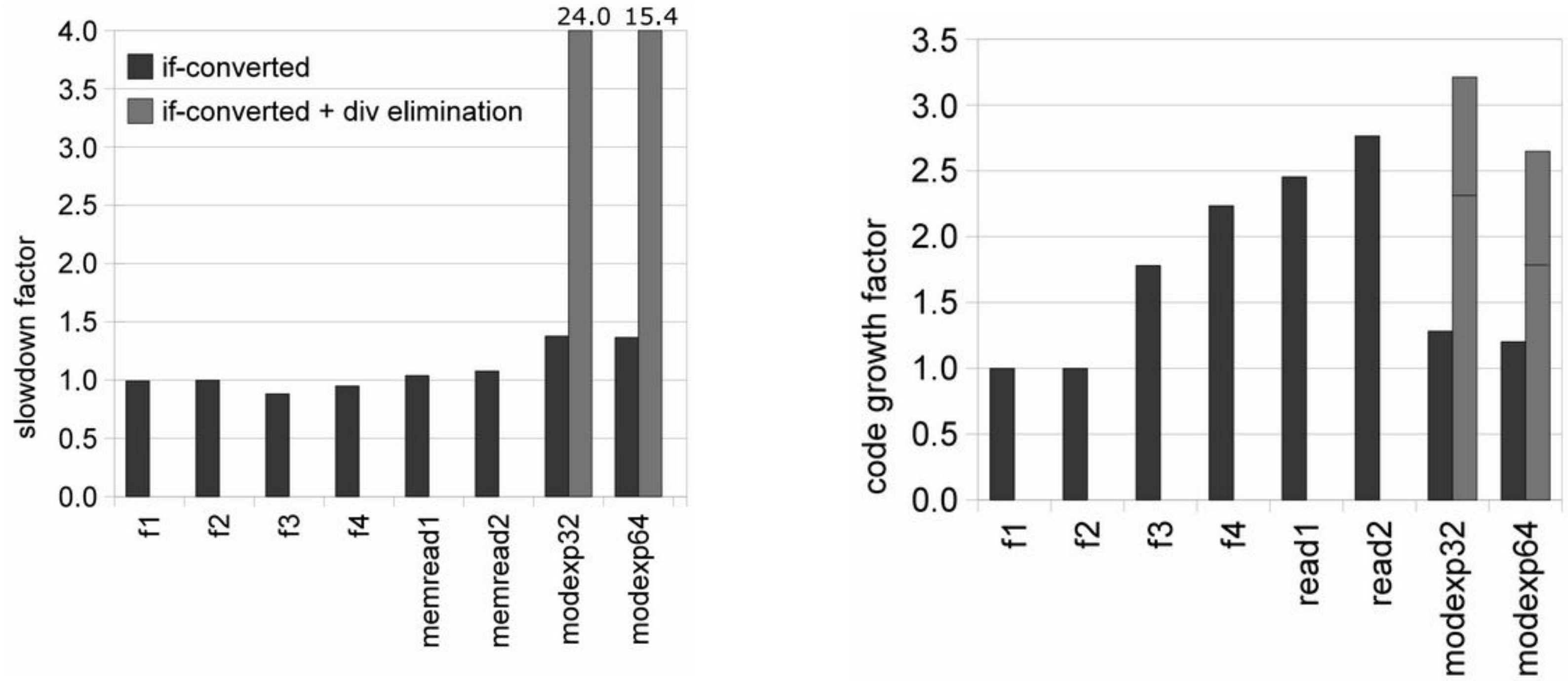
(a) Average execution time in seconds

	original				if-converted				if-converted + div elimination			
	all zero	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*	*all zero*	*all one*	*regular*	*random*
modexp32	0.015	0.000	0.001	0.003	0.001	0.001	0.001	0.001	0.007	0.007	0.007	0.007
modexp64	0.000	0.001	0.000	0.000	0.000	0.001	0.000	0.001	0.005	0.005	0.005	0.004

(b) Standard deviation of execution time

all zero	all one	regular	random
1100...000000	1111...111111	11...1100...00	1011...010011

Results: Efficiency



(c) Slowdown factor and code growth factor for microbenchmarks

Paper Critics

Strengths:

1. Scope is not very restricted (no “naive” assumptions)
    a. Function calls, variable latency instructions, and etc.
    b. Branch prediction
    c. General optimizations

Limitations:

1. Heavily rely on programmer annotation
2. Missing solutions for recursive calls
3. Simple experiments only

Questions?