Timing Side-Channel Attack

Using linear correlation to reveal secrets

A. Anselmo, S.A. Chiaberto, F. Chiatante, G. Roggero

Supervisor: *Prof. Renaud Pacalet*

21st June, 2019

Outline

Leaking informations

Side-channel attacks

Leaking informations

Side-channel attacks

1. any attack based on information gained from the implementation of a computer system

Leaking informations

Side-channel attacks

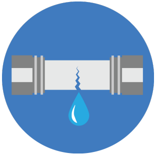
1. any attack based on information gained from the implementation of a computer system
2. timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information

Leaking informations

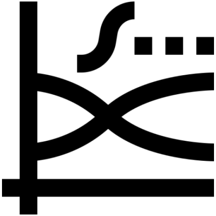
Side-channel attacks

1. any attack based on information gained from the implementation of a computer system
2. timing information, power consumption, electromagnetic leaks or even sound can provide an extra source of information
3. such information are therefore exploitable by an attacker

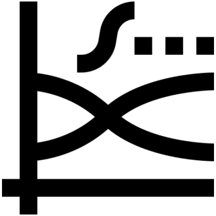
Therefore, our goal will consist in investigate such leaked information, trying to unveal secrets.



Introduction

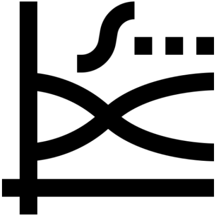


Introduction



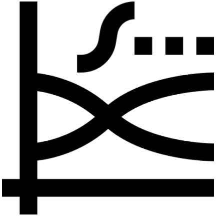
- in several algorithms used for security purposes some optimizations are introduced

Introduction



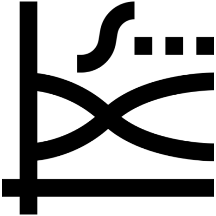
- in several algorithms used for security purposes some optimizations are introduced
- these optimizations lead to a linear dependency between time and the data encrypted

Introduction



- in several algorithms used for security purposes some optimizations are introduced
- these optimizations lead to a linear dependency between time and the data encrypted
- knowing information regarding the time-data pair, it is possible to find a correlation

Introduction



- in several algorithms used for security purposes some optimizations are introduced
- these optimizations lead to a linear dependency between time and the data encrypted
- knowing information regarding the time-data pair, it is possible to find a correlation
- this correlation can be used to unveal part of the secret

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

- timing for a sufficiently large number of cyphertexts is known

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

- timing for a sufficiently large number of cyphertexts is known
- cyphertexts are known

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

- timing for a sufficiently large number of cyphertexts is known
- cyphertexts are known
- secret is the same for all cyphertexts

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

- timing for a sufficiently large number of cyphertexts is known
- cyphertexts are known
- secret is the same for all cyphertexts
- the HW/SW implementation is known to the attacker

Hypothesis

Our starting point

In order to successfully extract the secret through the correlation, we have to make a list of assumptions:

- timing for a sufficiently large number of cyphertexts is known
- cyphertexts are known
- secret is the same for all cyphertexts
- the HW/SW implementation is known to the attacker
- a timing model can be built

From the very beginning

BIGINT required

In order to operate with large integers, we decided to develop our own library of functions to operate over integers of arbitrary length, in particular with the following elementary instructions:

From the very beginning

BIGINT required

In order to operate with large integers, we decided to develop our own library of functions to operate over integers of arbitrary length, in particular with the following elementary instructions:

- addition and subtraction

From the very beginning

BIGINT required

In order to operate with large integers, we decided to develop our own library of functions to operate over integers of arbitrary length, in particular with the following elementary instructions:

- addition and subtraction
- multiplication

From the very beginning

BIGINT required

In order to operate with large integers, we decided to develop our own library of functions to operate over integers of arbitrary length, in particular with the following elementary instructions:

- addition and subtraction
- multiplication
- bitwise operation, such as `AND`, `OR`, `XOR`, `NOT`

From the very beginning

BIGINT required

In order to operate with large integers, we decided to develop our own library of functions to operate over integers of arbitrary length, in particular with the following elementary instructions:

- addition and subtraction
- multiplication
- bitwise operation, such as `AND`, `OR`, `XOR`, `NOT`
- logical comparison

Library testing

Checking the correctness

As we wanted to check for formal correctness of the library, we proceed in the following way:

Library testing

Checking the correctness

As we wanted to check for formal correctness of the library, we proceed in the following way:

1. design a function written in C for our library

Library testing

Checking the correctness

As we wanted to check for formal correctness of the library, we proceed in the following way:

1. design a function written in *C* for our library
2. emulate its behavior in *Python*, using its infinite precision

TSCA
WE JUST CORRELATE

Library testing

Checking the correctness

As we wanted to check for formal correctness of the library, we proceed in the following way:

1. design a function written in *C* for our library
2. emulate its behavior in *Python*, using its infinite precision
3. launched an intense series of tests to check the formal equality

Library testing

Checking the correctness

As we wanted to check for formal correctness of the library, we proceed in the following way:

1. design a function written in *C* for our library
2. emulate its behavior in *Python*, using its infinite precision
3. launched an intense series of tests to check the formal equality

An interesting discovery

We have found out that the shift bt 32 bits (or multiples) does not produce an effect. This special case has to be handled in our library.

The least complex attack

Bare metal

We wanted to exploit the easiest possible attack. Since on a normal device an OS might cause interrupts, thus changing the total time of the enciphering, we decided to:

WE JUST CORRELATE

The least complex attack

Bare metal

We wanted to exploit the easiest possible attack. Since on a normal device an OS might cause interrupts, thus changing the total time of the enciphering, we decided to:

- compile our code for an ARM architecture

The least complex attack

Bare metal

We wanted to exploit the easiest possible attack. Since on a normal device an OS might cause interrupts, thus changing the total time of the enciphering, we decided to:

- compile our code for an ARM architecture
- add it to an *Eclipse* project

The least complex attack

Bare metal

We wanted to exploit the easiest possible attack. Since on a normal device an OS might cause interrupts, thus changing the total time of the enciphering, we decided to:

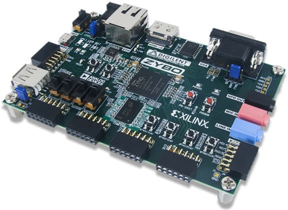
- compile our code for an ARM architecture
- add it to an *Eclipse* project
- used the MAKEFILE generated by Xilinx SDK

The least complex attack

Bare metal

We wanted to exploit the easiest possible attack. Since on a normal device an OS might cause interrupts, thus changing the total time of the enciphering, we decided to:

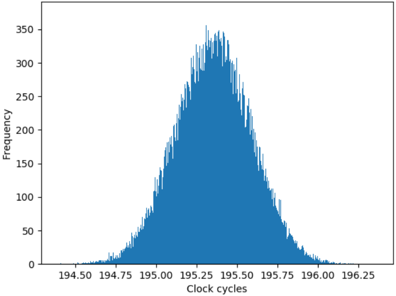
- compile our code for an ARM architecture
- add it to an *Eclipse* project
- used the MAKEFILE generated by Xilinx SDK
- copy the executable on the Zybo board



Finding correlations

PCC: our game changer

In order to find the linear contribution of each sample in the overall time, we have used the *Pearson Correlation Coefficient* as an estimator. It has proved to be really effective for our needs, working on the realizations of a random variable.



Algorithm

Development

Algorithm

Development

- Started in Python to allow better flexibility and library support (infinite precion, `scipy.stats`)

T S C A

WE JUST CORRELATE

Algorithm

Development

- Started in Python to allow better flexibility and library support (infinite precion, `scipy.stats`)
- At first, attacking conditional Montgomery Mult., 1 bit at-a-time, using fixed threshold

T S
C A
WE JUST CORRELATE

Algorithm

Development

- Started in Python to allow better flexibility and library support (infinite precion, `scipy.stats`)
- At first, attacking conditional Montgomery Mult., 1 bit at-a-time, using fixed threshold
- Move on to attack both MM to improve statistical relevance of 0 guesses

Algorithm

Development

- Started in Python to allow better flexibility and library support (infinite precion, `scipy.stats`)
- At first, attacking conditional Montgomery Mult., 1 bit at-a-time, using fixed threshold
- Move on to attack both MM to improve statistical relevance of 0 guesses
- Get rid of fixed threshold by: using multi bit analysis and taking max between the accumulated PCCs on a common path

Algorithm

Final implementation

Algorithm

Final implementation

- Attack at the same time the two Montgomery moltiplications present in an RSA iteration

Algorithm

Final implementation

- Attack at the same time the two Montgomery moltiplications present in an RSA iteration
- Timing estimate: dummy Montgomery moltiplication which evaluates the number of taken branches

Algorithm

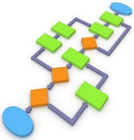
Final implementation

- Attack at the same time the two Montgomery moltiplications present in an RSA iteration
- Timing estimate: dummy Montgomery moltiplication which evaluates the number of taken branches
- Multi bit guessing

Algorithm

Final implementation

- Attack at the same time the two Montgomery moltiplications present in an RSA iteration
- Timing estimate: dummy Montgomery moltiplication which evaluates the number of taken branches
- Multi bit guessing
- Error-detection capabilities



Algorithm

Optimizations

Algorithm

Optimizations

- Completely rewritten in C with +10x speedup over Python

Algorithm

Optimizations

- Completely rewritten in C with +10x speedup over Python
- Fully customizable number of bits considered and guessed in one attack iteration

WE JUST CORRELATE

Algorithm

Optimizations

- Completely rewritten in C with +10x speedup over Python
- Fully customizable number of bits considered and guessed in one attack iteration
- Tweakable filtering of input data with `#define` parameters for noisy samples

Algorithm

To have an idea...

The C implementations, running on a machine with 2.4Ghz Intel i5:

Algorithm

To have an idea...

The C implementations, running on a machine with 2.4Ghz Intel i5:

- cracks 128-bit RSA in 3m40sec

Algorithm

To have an idea...

The C implementations, running on a machine with 2.4Ghz Intel i5:

- cracks 128-bit RSA in 3m40sec
- using 10k plaintexts sampled on Zybo board

Algorithm

To have an idea...

The C implementations, running on a machine with 2.4Ghz Intel i5:

- cracks 128-bit RSA in 3m40sec
- using 10k plaintexts sampled on Zybo board
- considering 2 bits and guessing 1 per iteration of the attack

Just a simplification

Works on computer also

Even if we mainly worked on the *Zybo* board, we can claim that:

Just a simplification

Works on computer also

Even if we mainly worked on the *Zybo* board, we can claim that:

- our attack works also when mounted for other devices, including different architectures (Intel x86, ..)

Just a simplification

Works on computer also

Even if we mainly worked on the *Zybo* board, we can claim that:

- our attack works also when mounted for other devices, including different architectures (Intel x86, ..)
- with an OS, more tuples (cipher, timing) are needed

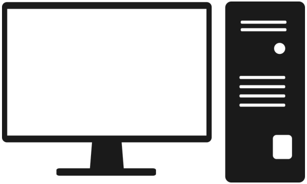
Just a simplification

Works on computer also

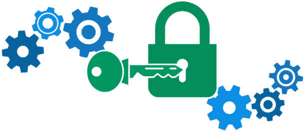
Even if we mainly worked on the *Zybo* board, we can claim that:

- our attack works also when mounted for other devices, including different architectures (Intel x86, ..)
- with an OS, more tuples (cipher, timing) are needed
- the attack is still feasible

We have completely tested what is mentioned above.



Bigger keys



RSA on 512/1024/2048/4096

The algorithm is capable of handling larger keys on 512, 1024, 2048 and 4096 bits. However, the processing time is longer, and a more complex backtrack might be necessary in some cases.

Possible solution

Blinding

The proposed countermeasure is the one given in Kocher (1996). It consists in blinding the message before the encryption using a couple of values v_f, v_i chosen in such a way that:

$$v_i^e \cdot v_f \, mod \; N = 1$$

This contermeasure, in all our tests, has proven to be really effective. Ciphers are completely masked, no correlation can be identified.

Possible solution

Blinding

Future expectations

What more

Future expectations

What more

- porting the attack in C++ to keep class structure and speedup w.r. to Python

Future expectations

What more

- porting the attack in C++ to keep class structure and speedup w.r. to Python
- find an optimal filter and explain the strange behavior of the implemented filter

Future expectations

What more

- porting the attack in C++ to keep class structure and speedup w.r. to Python
- find an optimal filter and explain the strange behavior of the implemented filter
- try to parallelize the estimation for all the messages, as every message is data-independent from each other

Our team

ANSELMO, Alberto

CHIABERTO, Simone

CHIATANTE, Fausto

ROGGERO, Giulio

References I

Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer.

TSCA
WE JUST CORRELATE