Side channel attack against the Mbed TLS implementation of the RSA algorithm.

Victor Micó Biosca

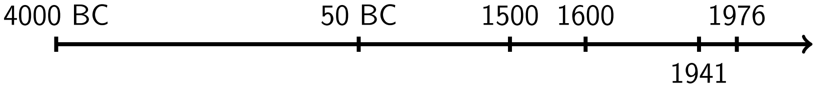
Escola Politècnica Superior
Universitat de Girona

May 29, 2024

Universitat de Girona

Table of Contents

Brief History of Cryptography



- 4000 BC Hieroglyphics in Egypt
- 50 BC Caesar Cipher
- 1553 Vigenère Cipher
- 1941 Alan Turing Deciphers the Enigma Machine
- 1976 The DES symmetric encryption algorithm is published
- 1976 Diffie and Hellman introduce Public and Private Key Exchange
- 1978 Publication of the RSA Public Key Encryption System (Rivest, Shamir, and Adleman)

RSA - Cryptographic Primitives

- **Encryption**: $c = m^e \mod n$ where m is the message, e is the public key, and c is the ciphertext.
- **Decryption**: $m = c^d \mod n$ where c is the ciphertext, d is the private key, and m is the message.
- **Signature**: In the signing process, the author of the message uses their private key to generate a signature $s = m^d \mod n$.
- **Signature Verification**: The signature s of a message m is verified by computing $m' = s^e \mod n$. If $m = m'$, then the signature is valid.

RSA - RSA Key Generation Process

1. Two large distinct prime numbers, p and q, of similar bit length are generated.
2. The modulus $n = p \cdot q$ is calculated.
3. The totient of n is calculated, i.e., $\varphi(n) = (p-1) \cdot (q-1)$.
4. A positive integer e is chosen such that it is coprime with $\varphi(n)$ and satisfies $1 < e < \varphi(n)$. The pair (n, e) will be the public key.
5. The private exponent d is calculated using a modular arithmetic operation called the multiplicative inverse. It must satisfy $d \cdot e \equiv 1 \mod \varphi(n)$. The exponent d will be the private key.

Cryptographic Devices

Cryptographic devices are capable of receiving a message through an interface, encrypting the content of the message, and transmitting the encrypted message. Generally, they are also capable of performing the reverse operation: receiving an encrypted message, decrypting it, and transmitting the plaintext message.

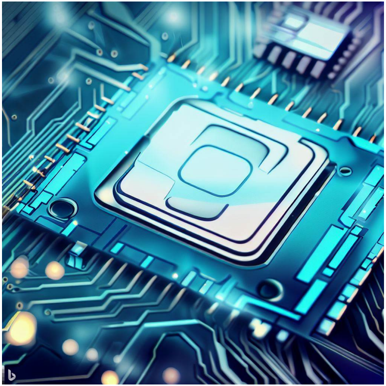


Figure: Representation of a cryptographic device created with Dall-E

Attacks on Cryptographic Devices

- **Active Attacks**: An active attack involves manipulating the inputs or the environment of the device to make it operate incorrectly or differently from normal conditions. Through fault injection, it is possible to make an incorrect PIN appear correct or extract cryptographic keys, among other things.
- **Passive Attacks**: In a passive attack, the attacker extracts information from the device through side channels while the device operates under normal conditions. These side channels can include power consumption, electromagnetic radiation, or even sound or temperature.

Modular Exponentiation Algorithms

Modular exponentiation is the most important operation in RSA. The most basic algorithm for computing m^e involves multiplying m by itself e times, i.e., $m \cdot m \cdot \ldots \cdot m$. For a 1024-bit key, this would mean:

$$2^{1024} > 2^{300}$$

Number of operations > Estimated number of atoms in the universe.

Modular Exponentiation Algorithms

Algorithm Left-to-right binary exponentiation

Require: m as message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_2)$ for $e_i \in (0, 1)$
Ensure: m^e

1: $A \leftarrow 1$
2: **for** $i \leftarrow t$ to 0 **do**
3: $\quad A \leftarrow A \cdot A$ {Square}
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad A \leftarrow A \cdot m$ {Multiply}
6: **return** A

Left-to-right Binary Exponentiation
Example: m=10, e=1693

Bit number:
Exponent: 11010011101
Current A: m^0 = 1E+0

Operation sequence:

Side-Channel Attacks

SPA: Simple Power Analysis

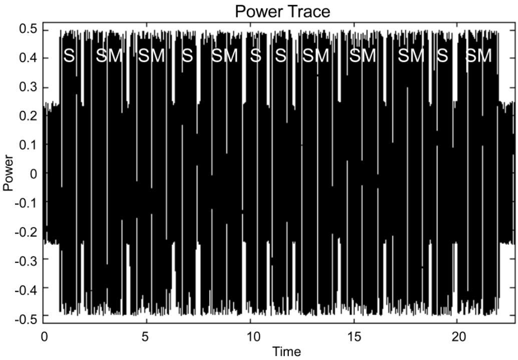


Figure: Power trace of RSA

Modular Exponentiation Algorithms

Algorithm Left-to-right multiply always binary exponentiation

Require: m as message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_2)$ for $e_i \in (0, 1)$
Ensure: m^e

1: $A \leftarrow 1$
2: **for** $i \leftarrow t$ to 0 **do**
3: $\quad A \leftarrow A \cdot A$ {Square}
4: $\quad$ **if** $e_i = 1$ **then**
5: $\quad\quad A \leftarrow A \cdot m$ {Multiply}
6: $\quad$ **else**
7: $\quad\quad T \leftarrow A \cdot m$ {Multiply and discard}
8: **return** A

Left-to-right Multiply Always Binary Exponentiation
Example: m=10, e=1693

Bit number:
Exponent: 11010011101
Current A: m^0 = 1E+0

Operation sequence:

Modular Exponentiation Algorithms

Algorithm Left-to-right k-ary exponentiation

Require: m as message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_b)$ for e_i where $b = 2^k$ for some $k > 1$
Ensure: m^e

$m_0 \leftarrow 1$
for $i \leftarrow 1$ to $(2^k - 1)$ **do**
  $m_i \leftarrow m_{i-1} \cdot m$ {Thus $m_i = m^i$}
$A \leftarrow 1$
for $i \leftarrow t$ to 0 **do**
  $A \leftarrow A^{2^k}$ {k Squares}
  $A \leftarrow A \cdot m_{e_i}$ {Multiply}
return A

Left-to-right k-ary Exponentiation
Example: m=10, e=16932, k=3

Window number:
Exponent: 100001000100100
Current A: m^0 = 1E+0

Operation sequence:

Modular Exponentiation Algorithms

Algorithm Sliding-window exponentiation

Require: m as message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_2)$ with $e_t = 1$ and integer $k \geq 1$
Ensure: m^e

$m_1 \leftarrow m$
$m_2 \leftarrow m^2$
for $i \leftarrow 1$ to $(2^{k-1} - 1)$ **do**
    $m_{2i+1} \leftarrow m_{2i-1} \cdot m_2$
$A \leftarrow 1$
$i \leftarrow t$
while $i \geq 0$ **do**
    if $e_i = 0$ **then**
        $A \leftarrow A \cdot A$ {Square}
        $i \leftarrow i - 1$
    else {Find the longest bitstring $e_i e_{i-1} \dots e_l$ such that $i - l + 1 \geq k$ }
        $A \leftarrow A^{i-l+1}$ {k Squares}
        $A \leftarrow A \cdot m_{(e_i e_{i-1} \dots e_l)_2}$ {Multiply}
        $i \leftarrow l - 1$
return A

Sliding-window Left-to-right Exponentiation
Example: m=10, e=16932, w=3

Current window index:
Exponent: 100001000100100
Current A: m^0 = 1E+0

Operation sequence:

Vertical Attacks vs. Horizontal Attacks

Vertical Attacks

- SPA
- CPA
- Template attacks
- DL-Based attacks

Horizontal Attacks

- Big Mac attack
- Horizontal Correlation Analysis
- Cross-correlation
- Clustering Analysis

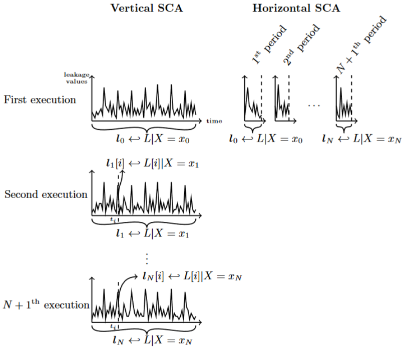


Figure: Vertical and Horizontal Attacks.

Side-Channel Attacks

CPA: Correlation Power Analysis

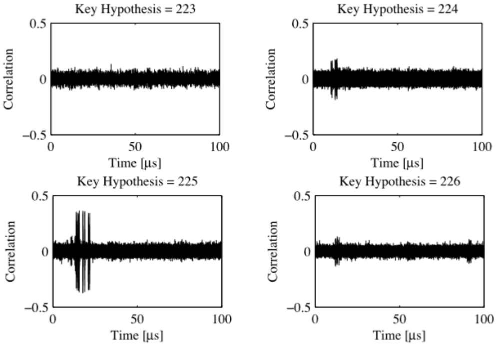


Figure: Result of CPA for different intermediate value hypotheses

Countermeasures against Side-Channel Attacks applied to RSA

Exponent Obfuscation

CPA attacks target the exponent, which remains fixed across multiple traces. To prevent this, it is possible to obfuscate the exponent in each new execution by adding an additive mask.

The secret exponent is randomized using the following equation:

$$d' \leftarrow d + r \cdot \phi(n)$$

Where r is a random number and $\phi(n)$ is Euler's totient function applied to the modulus n.

Using the obfuscated exponent yields the same ciphertext, i.e., $m^d \equiv m^{d'}$.

Countermeasures against Side-Channel Attacks applied to RSA

Message Obfuscation

CPA attacks also exploit the ability to control the message or the fact that the message is known. To prevent this, we can obfuscate the message before encryption. To do this, a random number r is generated, and with this number, r_1 and r_2 are calculated to make the input message unpredictable and to correct the final result, respectively:

$$r_1 = r^e \mod n$$
$$r_2 = r^{-1} \mod n$$

Then during the RSA operation:

$$x' = x \cdot m_1$$

Set up

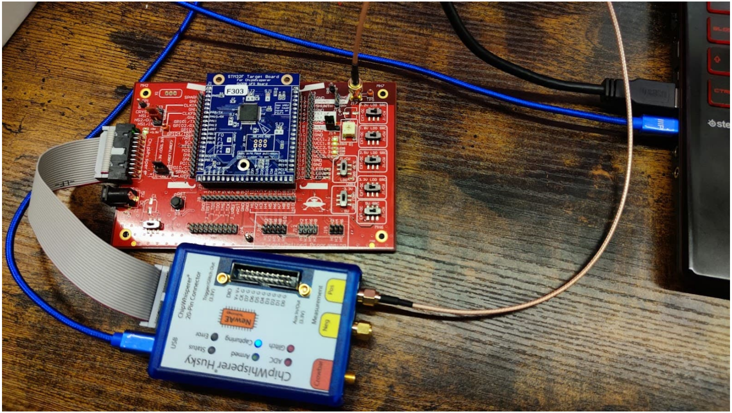


Figure: Set up

Proposed Method for Analysis

1. Analyze various public RSA libraries and select one that uses a window exponentiation algorithm for the attack.
2. Capture a trace.
3. Perform a visual analysis of the power trace, identifying RSA regions and modular operations.
4. Develop a method to distinguish multiplication from squares.
5. Develop a method to distinguish between different precalculated values.

SPA

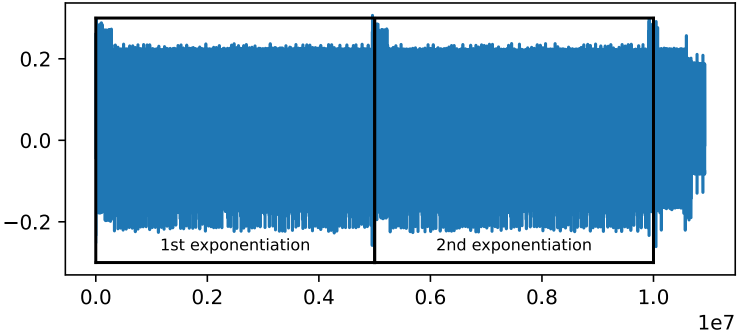


Figure: Complete RSA trace

SPA

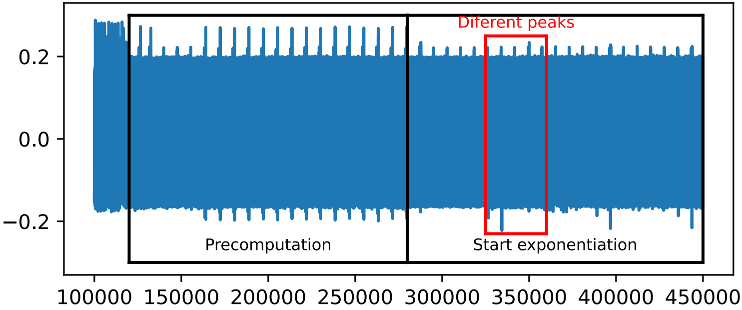


Figure: Precalculations and start of exponentiation

SPA

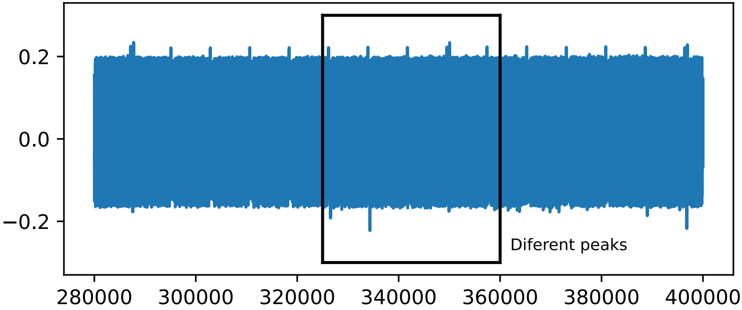


Figure: Different spikes between modular operations

SPA

Algorithm Lowpass filter

Require: t as Trace to filter
Require: $weight$ as Weight of the lowpass filter
Ensure: $result$ Trace filtered

$weight_1 \leftarrow weight + 1$
$N \leftarrow length(trace)$
for $i \leftarrow 1 to N$ **do**
$result[i] \leftarrow (result[i] + weight * result[i-1])/weight_1$
$i \leftarrow N - 2$
while $i \geq 0$ **do**
$result[i] \leftarrow (result[i] + weight * result[i+1])/weight_1$
return $result$

SPA

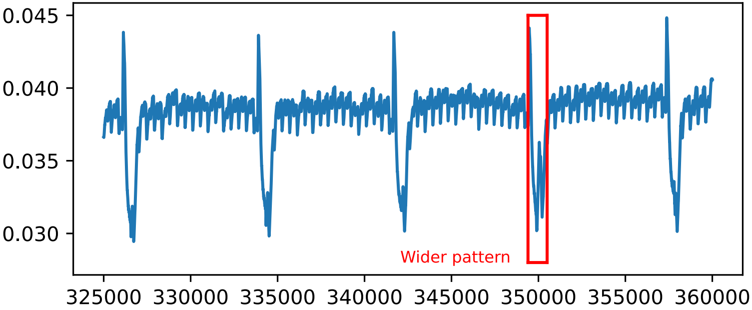


Figure: Lowpass filterd trace

Pattern Matching

Algorithm Pattern match

Require: t as trace
Require: ref as Reference pattern
Ensure: $scores$

$N \leftarrow \text{length}(trace)$
$n \leftarrow \text{length}(ref)$
for $i \leftarrow 1$ to N **do**
  $score[i] \leftarrow \text{corr}(ref, \text{trace}(i, i+n))$
return $score$

Pattern Matching: Square and Multiplication Identification

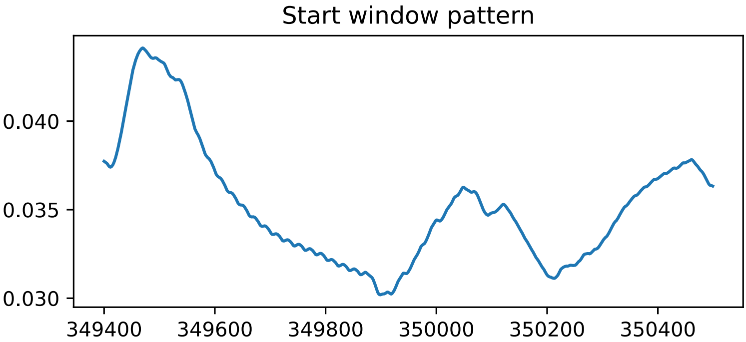


Figure: Window start pattern

Pattern Matching: Square and Multiplication Identification

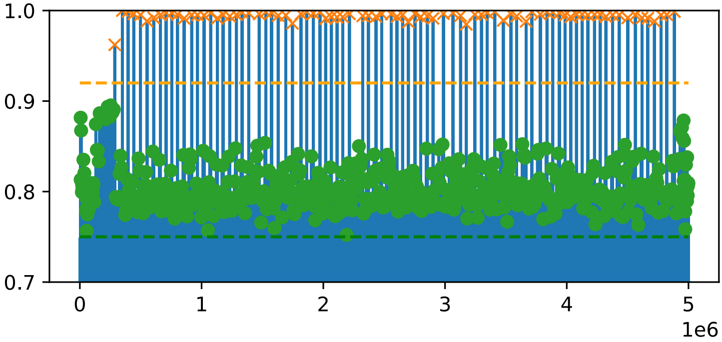


Figure: Result of pattern matching

Pattern Matching: Square and Multiplication Identification

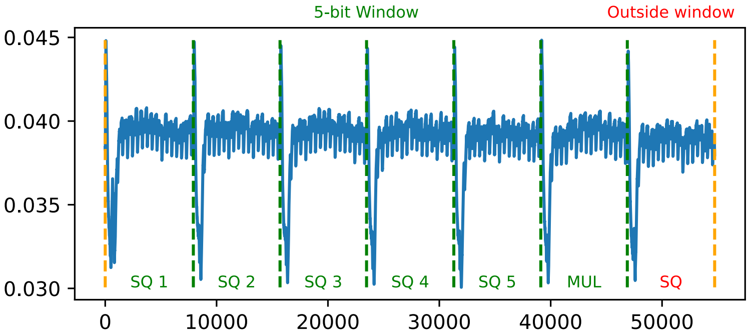


Figure: Identification of modular operations

Pattern Matching: Square and Multiplication Identification

Bits obtained from the first exponentiation

1xxxx001xxxx1xxxx1xxxx01xxxx01xxxx01xxxx01xxxx1xxxx1xxxx1xxxx01x
xxx001xxxx1xxxx01xxxx01xxxx01xxxx1xxxx01xxxx01xxxx1xxxx01xxxx000
001xxxx01xxxx1xxxx01xxxx001xxxx01xxxx0001xxxx1xxxx01xxxx01xxxx1x
xxx1xxxx1xxxx1xxxx01xxxx000000001xxxx1xxxx001xxxx1xxxx00001xxxx1
xxxx1xxxx1xxxx01xxxx1xxxx01xxxx1xxxx000001xxxx00001xxxx001xxxx1x
xxx0001xxxx01xxxx1xxxx001xxxx0001xxxx001xxxx1xxxx00001xxxx1xxxx0
001xxxx01xxxx1xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx01xxxx1xxxx1xxx
x01xxxx01xxxx0001xxxx001xxxx01xxxx1xxxx01xxxx01xxxx1xxxx00xxxxxx

Pattern Matching: Square and Multiplication Identification

Bits obtained from the second exponentiation

1xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx0001xxxx1xxxx1xxxx1xxxx00001
xxxx1xxxx1xxxx01xxxx1xxxx1xxxx00001xxxx01xxxx01xxxx1xxxx1xxxx1xx
xx1xxxx1xxxx01xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx1xxxx01xxxx1xxx
x1xxxx1xxxx01xxxx1xxxx001xxxx1xxxx1xxxx000001xxxx01xxxx1xxxx1xxx
x1xxxx00001xxxx0001xxxx0001xxxx1xxxx01xxxx01xxxx1xxxx1xxxx1xxxx0
1xxxx0001xxxx1xxxx1xxxx1xxxx1xxxx01xxxx001xxxx1xxxx0001xxxx1xxxx
1xxxx0001xxxx1xxxx1xxxx1xxxx1xxxx1xxxx00001xxxx01xxxx1xxxx1xxxx1
xxxx01xxxx01xxxx01xxxx01xxxx01xxxx0001xxxx001xxxx00001xxxxxxxxx

Pattern Matching: Identification of Bits within a Window

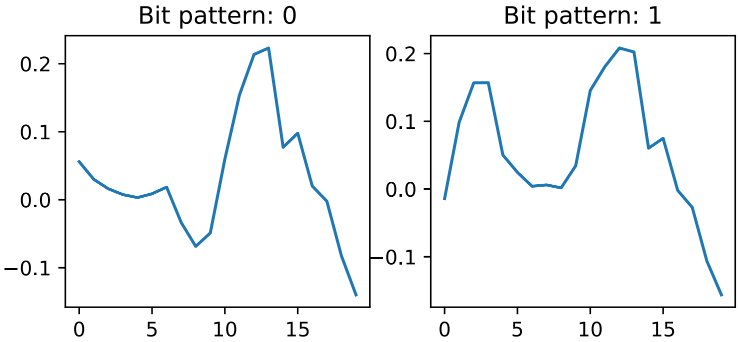


Figure: Patterns corresponding to the loading of a zero and a one

Pattern Matching: Identification of Bits within a Window

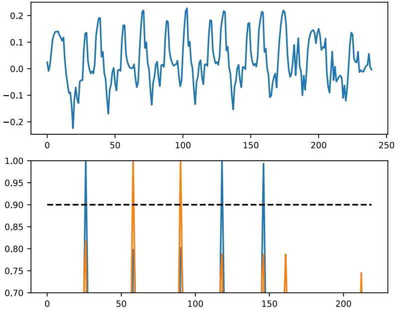


Figure: Top: Trace segment corresponding to the loading of bits within a window Bottom: Result of pattern matching for zero and one bits.

Pattern Matching: Square and Multiplication Identification

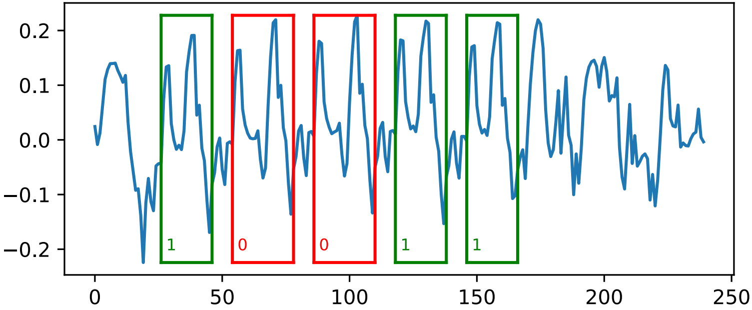


Figure: Identification of individual bit loading within the window

Pattern Matching: Identification of Squares and Multiplications

Bits obtained from the first exponentiation

1100000110101100111101010110011101010110010000100111010011111011
0000011110100000101110111010110101011101001001101110001010011000
0011110010010100110100100010001010001000101011001101011101100011
1111110110001110010101100000000011101101010010100111000000101101
111110000100000011001100010100101111100000011011000010111001110010
0010001101101010111001001001000011001001001111110000001111111000
0011110011000101010110011100110100100111101111000010110100011100
1010010010001000100000001000101110011000010111011010101110010010x

Pattern Matching: Identification of Squares and Multiplications

Bits obtained from the second exponentiation

1001001010110011100101111110110111110001111110111101110100001
0101110111001101001011101111100000101010101010111100111000111 1
1011000101010101110101001010110011101001001111010100010100001100
0100011001001110010111001001111011111010000010110010010101011110
1111100000111110001001100010011110010101100100101000111000110110
1011100010100100011101111111110010110010010111101010001011110110
1000000011110100001101111010111011011000001011001001011100100111
101101000101101001100101010001110000011111001011000001101010111

Summary of Results

Distinguishing	1st exponentiation	2nd exponentiation
Squares from multiplications	33.98%	30.91%
Bits of each window	99.80%	100%

Table: Summary of Results

Conclusions

Side-channel attacks are feasible, they can be carried out with a tight budget and relatively simple signal processing methods.

As future work, we propose:

1. Update the code of the Mbed TLS library to the latest version to check if it is possible to exploit this vulnerability.
2. Try other target devices alternatives to the STM32F3.
3. Use other techniques to extract the exponent values, such as clustering algorithms.

Many thanks!

Slides, traces and complete attack available at github.com/victormico/sca-mbedtls-rsa