Universitat de Girona
Escola Politècnica Superior

Master's Thesis

Program: Master in Data Science

Title: Side Channel Attack Against the Mbed TLS Implementation of the RSA Algorithm.

Document: Thesis

Student: Victor Micó Biosca

Supervisor: David Juher Barrot
Department: Department of Computer Science, Applied Mathematics, and Statistics
Area: Applied Mathematics

Session (month/year): June 2023

Universitat de Girona
Escola Politècnica Superior

MASTER'S THESIS

Side Channel Attack Against the Mbed TLS Implementation of the RSA Algorithm.

Author:
Victor MICÓ BIOSCA

June 2023

Master in Data Science

Supervisor:
David JUHER BARROT

Abstract

The main operation of RSA is modular exponentiation. Depending on the implementation, side-channel attacks such as DPA can be performed. To protect this operation, cryptographic library developers implement countermeasures like exponent obfuscation, which prevents the use of multiple traces. Consequently, attacks must be carried out using a single trace.

In this work, we target the RSA implementation in the Mbed TLS library, which uses a sliding window exponentiation algorithm. Through SPA and pattern matching techniques, we have successfully extracted both private exponents of the RSA key, d_p and d_q, using a single power trace.

The code and trace used in this work can be found at the following GitHub address: victormico/sca-mbedtls-rsa

Acknowledgments

Firstly, I want to express my sincere gratitude to Colin O'Flynn and the staff at *NewAE Technology Inc.* for selecting and supporting the research proposal that has led to this thesis.

Additionally, I would like to extend special thanks to my supervisor, David Juher, for his invaluable guidance and collaboration.

I would also like to convey my deep gratitude to my partner, Marina, my mother, Pura, and my sister, Maria, for their constant support and patience throughout this process.

Likewise, I want to give special thanks to my friend Alberto Marcos for the insightful conversations about cryptography and data science, which have been a true source of inspiration for this work.

Finally, I dedicate this work to the memory of my father, whom I am sure would have enjoyed reading it.

Without the unwavering support of all of them, this work would not have been possible.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

The need to transmit sensitive information to another person without third parties being able to know its content is the basis of cryptography. The history of cryptography dates back to 4000 BC in Egypt, where hieroglyphics were used to encode messages for transmission. During the time of the Roman Emperor Julius Caesar, the Caesar cipher was used. The Caesar cipher involves replacing each letter with the corresponding letter shifted by a certain number of positions.

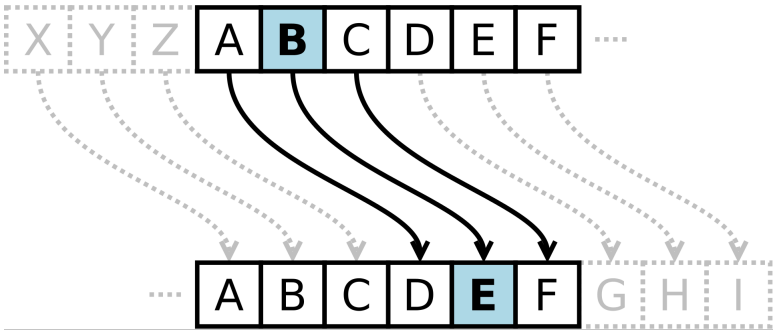

Figure 1.1: Character substitution in the Caesar cipher

It took 1500 years for an improvement to be introduced in this cipher algorithm. [Aumasson 2017] The Vigenère cipher is a modification of the Caesar algorithm where, instead of replacing each letter using a fixed number, a variable number of positions is used for each letter.

This algorithm was used until World War I. Afterward, with the advent of computing, especially during World War II, multiple encryption mechanisms were developed. In particular, the Allied forces intercepted communications of the enemy by discovering how the Enigma machine worked. This proved to be a crucial point for the subsequent development of the conflict.

Cryptography, therefore, was used by organizations, governments, and nations, primarily for military purposes. Nowadays, with the widespread use of computers and the internet, cryptography plays a crucial role in ensuring the confidentiality of the information we handle in our daily lives.

Currently, there are public and standardized algorithms like DES (Data Encryption Standard) or its successor, AES (Advanced Encryption Standard).

Both DES and AES base their security on combining the initial message with a key and then performing bit permutations and substitutions.

In the case of DES, the key has a size of 56 bits. This allows for a brute-force attack with current hardware. Brute-force attacks involve trying all possible keys until the correct message is obtained. With a machine composed of an array of 120 FPGAs called COPACOBANA, with an approximate cost of around $ 9,000, it is possible within an average of 6.4 days [van Tilborg 2014].

In the case of AES, its 128-bit length prevents a brute-force attack with current computational capacity.

All mentioned algorithms are symmetric-key algorithms, meaning the same key is used for encryption and decryption.

In 1976, Diffie and Hellman introduced the concept of public and private keys, laying the foundation for asymmetric cryptography. The exchange of public and private keys relies on the discrete logarithm problem. However, Diffie and Hellman did not propose any specific encryption system. Two years later, in 1978, Rivest, Shamir, and Adleman discovered a public-key encryption system, which was named RSA, using the initials of each of them. RSA is based on another mathematical problem, the difficulty of factoring prime numbers.

With the widespread use of encryption algorithms, it soon became necessary to use devices to securely store encryption keys because storing keys directly on a computer or mobile device can pose a security problem. External devices such as smart cards or digital wallets are necessary for this purpose. Some everyday items that use embedded cryptographic devices include credit cards, mobile phone SIM cards, ID cards, or passports.

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.

The security of these devices is not only based on the robustness of their algorithms but also on the theoretical impossibility of extracting the cryptographic keys they contain. The action of attempting to obtain the keys of a cryptographic device without authorization is called an attack.

Attacks on cryptographic devices can be divided into two main categories:

- Active attacks: An active attack involves manipulating the inputs or the environment of the device with the aim of making it operate erroneously or differently from normal conditions. With fault injection, it is possible to pass a wrong PIN as 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 be electrical consumption, electromagnetic radiation, or even sound or temperature.

Obtaining the keys of the aforementioned devices can mean gaining access to a person's bank account, intercepting their communications, or impersonating their identity. That's why these types of attacks are of great interest to the academic community, and new ones and countermeasures are continually being published to prevent them.

The objective of this work is to perform a side-channel attack on an open-source implementation of the RSA algorithm.

Although the state of the art mentions many publications on how to carry out side-channel attacks, most of them focus on symmetric key algorithms. Among the publications on asymmetric keys, few openly publish the data and code to reproduce the attack.

The motivations for this work are as follows:

1. Strengthen an open-source code library.
2. Disseminate the methods of capturing, processing, and attacking an asymmetric key algorithm.
3. Openly publish the analysis and traces.

The result of this work is the complete extraction of the RSA key through a single power consumption trace.

CHAPTER 2

Preliminaries

2.1 Domain

2.1.1 Public Key Cryptography

Public key cryptography uses two different keys for encrypting and decrypting data: a public key and a private key. The public key can be openly shared with others, while the private key is kept secret by the user.

The encryption process involves using the public key to transform a message into encrypted text that cannot be read without the corresponding private key. This encrypted text can be securely sent through a public channel. To decrypt the encrypted text and obtain the original message, the private key is used.

In addition to encryption, public key cryptography has other important applications such as digital signatures and authentication. With digital signatures, a unique digital signature of a message can be generated using the private key, allowing verification of its authenticity and integrity. This is useful for verifying the authorship of a message and ensuring it has not been altered during transmission.

Public key cryptography is based on complex mathematical problems that are difficult to solve, such as factoring large numbers (in the case of the RSA algorithm) or the discrete logarithm problem in elliptic curve cryptography (in the case of the ECC algorithm). These problems provide the security of the system, as they require significant computational effort to be solved.

2.1.1.1 RSA

As introduced in Chapter 1, RSA was designed by Rivest, Shamir, and Adleman and relies on the difficulty of factoring large numbers.

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 message's author 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 Key Generation Process The RSA public and private keys are generated as follows:

1. Generate two large, distinct prime numbers, p and q, with similar bit lengths.
2. Calculate the modulus $n = p \cdot q$.
3. Calculate the totient of n, i.e., $\varphi(n) = (p-1) \cdot (q-1)$.
4. Choose a positive integer coprime to $\varphi(n)$ and satisfies $1 < e < \varphi(n)$. The pair (n, e) becomes the public key.
5. Calculate the private exponent d by performing a modular arithmetic operation called multiplicative inverse. It must satisfy $d \cdot e \equiv 1 \mod \varphi(n)$. The exponent d becomes the private key.

2.1.1.2 Modular Exponentiation Algorithms

Among all the operations required to compute any of the RSA primitives, modular exponentiation is the most fundamental. The basic algorithm for computing m^e is to multiply m by itself e times, i.e., $m \cdot m \cdot \ldots \cdot m$. For large numbers like those in RSA, this method is highly inefficient, requiring $e-1$ multiplications. A typical RSA key is 1024 bits long, and performing exponentiation in this manner is practically impossible, given the astronomical number of operations involved (on the order of 2^{1024}).

To reduce the number of operations, various algorithms perform modular exponentiation by combining multiplications with squares. In the case of Algorithm 1 (left-to-right binary exponentiation), also known as square and multiply, the exponent is processed by examining each bit from left to right. If the bit is '0', a square is performed, and if the bit is '1', both a square and a multiplication are performed. Thus, on average, if there are as many 1s as 0s in a randomly chosen RSA key, the required number of operations for a 1024-bit key would be $1.5 \cdot 1024 = 1,536$, a computation easily achievable by any device [Menezes 2007].

The choice of the exponentiation algorithm is not only relevant in terms of efficiency but also crucial for security. If we can distinguish between multiplications and squares, it is possible to directly extract the exponent [Kocher 1999].

In Figure 2.1, we can see an example of a power trace where the sequence of modular operations can be identified. Squares are marked with an S (*square*), and multiplications are marked with an M.

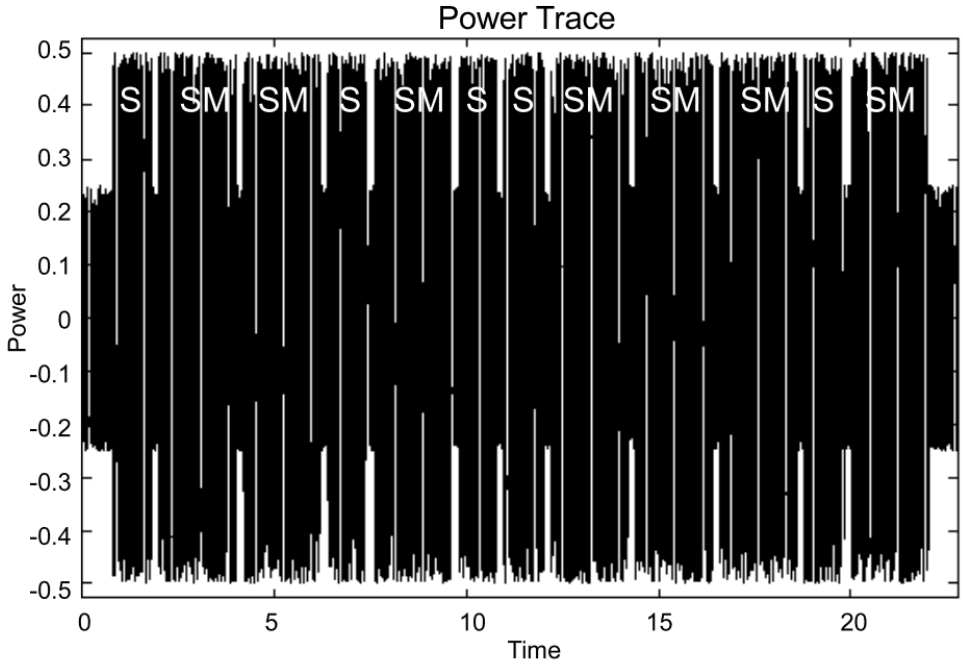

Figure 2.1: RSA Power Trace [Paar 2010]

In Chapter 3, we will examine in detail the side-channel attacks and how, through power consumption, we can observe these differences.

Algorithm 1 Left-to-right binary exponentiation

Require: m as the message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_2)$ for $e_i \in (0, 1)$
Ensure: m^e
  $A \leftarrow 1$
  for $i \leftarrow t$ to 0 **do**
    $A \leftarrow A \cdot A$ {Square}
    if $e_i = 1$ **then**
      $A \leftarrow A \cdot m$ {Multiply}
  return A

An easy and efficient countermeasure involves performing an additional multiplication in case the bit is '0' but discarding the result [Coron 1999]. In algorithm 2, we can see this modification.

Algorithm 2 Left-to-right multiply always binary exponentiation

Require: m as the message
Require: $(e = (e_t e_{t-1} \dots e_1 e_0)_2)$ for $e_i \in (0, 1)$
Ensure: m^e
  $A \leftarrow 1$
  for $i \leftarrow t$ to 0 **do**
    $A \leftarrow A \cdot A$ {Square}
    if $e_i = 1$ **then**
      $A \leftarrow A \cdot m$ {Multiply}
    else
      $T \leftarrow A \cdot m$ {Multiply and discard}
  return A

Despite this countermeasure, in case there is a small difference, for example in the exponent management, it is still possible to extract the key [Amiel 2009].

Another type of exponentiation algorithms is windowed exponentiation, which, instead of processing the exponent bit by bit, processes it in windows of k bits. These algorithms require precomputing some values using the message to be exponentiated. In algorithm 3, powers of m from 0 to $2^k - 1$ are precomputed.

For example, for $k = 3$, powers $m^0, m^1, \dots, m^6, m^7$ are precomputed. Once calculated, the exponent is processed in windows of three bits (000, 001, 010, 011, 100, 101, 110, 111) or equivalently (0, 1, 2, 3, 4, 5, 6, 7). For each window, three squares and one multiplication by the corresponding precomputed value are performed.

In this algorithm, it is not possible to extract the exponent by distinguishing only between multiplications and squares since it is independent of the sequence of operations. Therefore, the use of these algorithms can be considered a countermeasure. Horizontal side-channel attacks are necessary to attack them [Järvinen 2017], which we will explain in detail in section 3.

Algorithm 3 Left-to-right k-ary exponentiation

Require: m as the 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

Finally, we will introduce the sliding window algorithm (*sliding window*) in algorithm 4. It differs from the previous one in that it requires fewer precomputations and fewer multiplications [Menezes 2007].

This algorithm goes through the exponent bit by bit. While the bits are 0, it computes squares. In contrast, when it encounters a 1, it selects a window of bits of size k and computes k squares and one multiplication by the corresponding precomputed value. That's why it's called a sliding window. This algorithm is

Algorithm 4 Sliding-window exponentiation

Require: m as the message
Require: $(e = (e_t e_{t-1} ... e_1 e_0)_2)$ 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 $i - l + 1 \geq k$ }
      $A \leftarrow A^{i-l+1} \cdot A$ {k Squares}
      $A \leftarrow A \cdot m_{(e_i e_{i-1} ... e_l)2}$ {Multiply}
      $i \leftarrow l - 1$
  return A

implemented by the Mbed TLS library and is the subject of the attack in this work.

To implement RSA more efficiently, the RSA-CRT mode is utilized. For simplicity, we will only look at how the Chinese Remainder Theorem (CRT) is applied to accelerate decryption $m = c^d \mod n$ or signing a message $s = m^d \mod n$. To refer interchangeably to both operations, we will use the following nomenclature: $y = x^d \mod n$.

Using CRT, instead of performing an operation with a long modulus n, it can be replaced with two shorter exponentiations with the prime numbers p and q. As it involves a type of arithmetic transformation, there are three steps: transforming to the CRT domain, performing the operation within the CRT domain, and finally, applying the inverse transformation. [Paar 2010]

Transforming the input to the CRT domain To transform to the CRT domain, we only need to reduce the base element x modulo the two factors p and q of the modulus n. This gives us the modular representation of x.

$$x_p \equiv x \mod p$$
$$x_q \equiv x \mod q$$

Exponentiation in the CRT domain With the reduced versions of x, the following two exponentiations are performed:

$$y_p \equiv x_p^{d_p} \mod p$$
$$y_q \equiv x_q^{d_q} \mod q$$

where:

$$d_p \equiv d \mod (p-1)$$
$$d_q \equiv d \mod (q-1)$$

Inverse Transformation To combine and perform the inverse CRT transformation, the following operation is carried out:

$$h = qInv \cdot (y_p - y_q) \mod p$$

$$y = y_q + h \cdot q$$

where:

$$qInv = q^{-1} \mod p$$

In practice, d_p, d_q, and $qInv$ are precalculated, and the private key is stored as the following five parameters: (p, q, d_p, d_q, $qInv$).

2.2 Set-up

2.2.1 ChipWhisperer

ChipWhisperer is an open-source project created by Colin O'Flynn as part of his doctoral thesis [O'Flynn 2017]. It is a standardized platform for capturing and analyzing the power consumption of cryptographic devices. The project's goal is to make the tools necessary for conducting side-channel attacks and fault injection accessible to hardware researchers and developers. With this objective,

the company NewAE was founded, which provides capture hardware, target devices, and the software environment.

2.2.1.1 ChipWhisperer Husky

The ChipWhisperer Husky is a tool for conducting side-channel attacks and fault injection. It can capture samples of a power signal in streaming mode, making it ideal for capturing asymmetric algorithms with long execution times, such as RSA.

Features:

- **Sample Rate & ADC**: 200 MS/s, 12-bit
- **Sample Buffer Size**: > 80K Sample
- **Streaming Support (limited by computer buffer)**: >20 MS/s, 8-bit data can stream back for unlimited capture sizes.
- **Voltage Glitching**: 2-size Crowbar glitch
- **Clock Glitching**: High-resolution glitch generation based on phase-shift architecture (sub nS resolution)
- **I/O Pins**: ChipWhisperer 20-pin header, additional 8 data + 1 clock line. All I/O pins 3.3 V
- **FPGA**: Artix A35

2.2.1.2 CW308 UFO Target Board

The CW308 board is designed to mount various target boards on top of it. It has power regulators, oscillator filters, and all the necessary interfaces to extract the signal from the microcontroller and send it to the capture device.

Features:

- 1.2V, 1.8V, 2.5V, 3.3V, and V-ADJ (1.25V - 3.5V range) power supplies.
- Oscillator driver with a crystal socket to allow the use of most 2 or 3-pin crystals to drive the target device or ChipWhisperer.
- On-board LC low-pass filter to provide "clean" power supply for resistive shunt measurement.
- Diode protection on I/O lines to allow voltage glitch insertion on the target with less risk to connected devices.

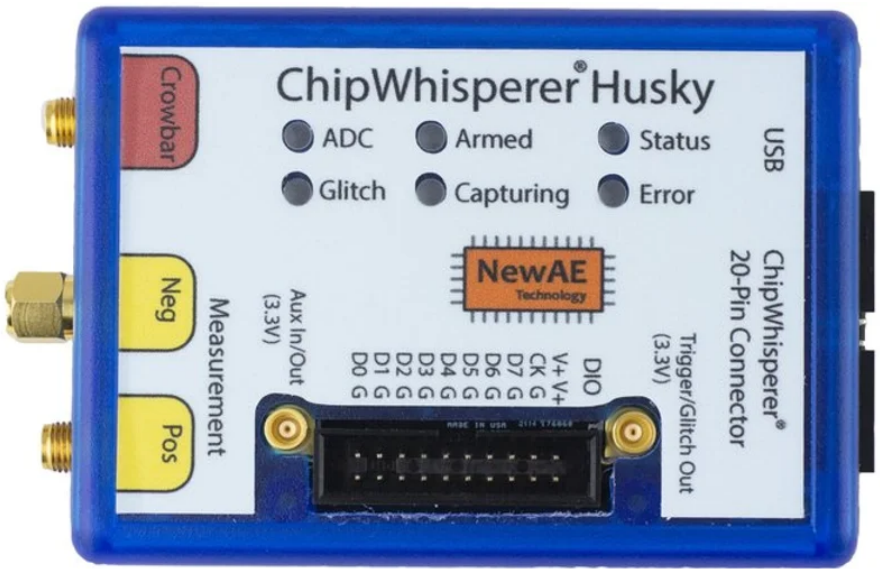

Figure 2.2: ChipWhisperer Husky [NewAE 2023d]

- Soft-start on input power to avoid disconnecting ChipWhisperer-Lite USB when switching power on/off.
- Includes 8-bit Atmel XMEGA and 32-bit STM32F3 (Cortex M3) target devices.

2.2.1.3 STM32F3 Target Board

The STM32F3 board hosts the ST microcontroller and allows it to be connected to the CW308 board. This board houses and runs the code of the Mbed TLS library.

2.2.2 Mbed TLS Cryptographic Library

Mbed TLS is an open-source cryptographic library that provides security functionalities for network applications and embedded devices. Originally known as PolarSSL, it was acquired by ARM and later rebranded as Mbed TLS.

The library offers an efficient and reliable implementation of cryptographic algorithms such as symmetric encryption, asymmetric encryption, hash algorithms, secure layer protocols (TLS/SSL), and other security-related functionalities. It is designed to be lightweight, modular, and easy to integrate into different platforms and environments.

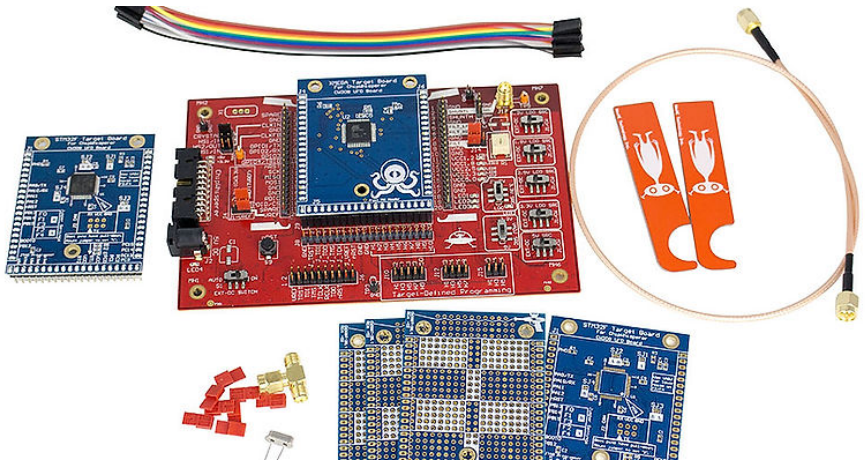

Figure 2.3: CW308 UFO Target board and accessories [NewAE 2023b]

Mbed TLS is widely used in various applications, including IoT (Internet of Things) devices, web servers, network devices, embedded systems, and secure communication applications in general.

Mbed TLS supports different security protocols, such as TLS 1.2 and 1.3, and provides tools and functionalities for certificate and cryptographic key management. It also implements various security measures to protect against attacks such as side-channel vulnerabilities and memory management-related vulnerabilities.

The RSA implementation of this library uses the sliding-window exponentiation algorithm (4). The code for this implementation can be found in the annex 6.3.

To perform the attack, we will use version **2.5.1** as it is integrated into the ChipWhisperer software.

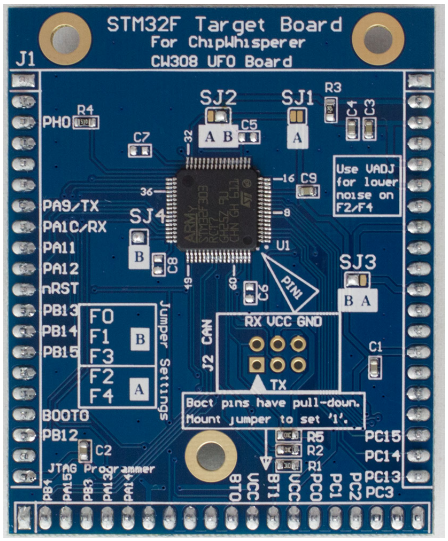

Figure 2.4: STM32F3 Target Board [NewAE 2023c]

CHAPTER 3

State of the Art

3.1 Side-channel attacks

3.1.1 Simple Power Analysis

When an electronic device runs code or implements a cryptographic operation, it performs different operations. Each operation has a different computational cost, some are more complex and require more power, and others require less. Each operation can have a differentiated power consumption profile. The direct analysis of one or several power traces is called Simple Power Analysis (SPA).

Through SPA, an attacker can deduce which operations a cryptographic device is performing. For example, if an attacker knows that a device is performing a cryptographic operation but does not know which one, and observes ten similar patterns, they can deduce that the operation is AES, as the algorithm consists of ten rounds.

SPA attacks can be very powerful if the attacker has access to the source code being executed by the device because it is then possible to map each operation in the code to identified patterns in the power trace.

The first published SPA attack was carried out by Kocher *et al.* [Kocher 1999], who demonstrated how RSA could be attacked using power consumption traces. The exponentiation algorithm used was binary exponentiation (Algorithm 1). As seen in Chapter 2, they were able to distinguish between squares and multiplications, thus extracting the entire RSA exponent.

3.1.2 Correlation Power Analysis

In a Correlation Power Analysis (CPA), the data processed by the cryptographic device and the power consumption traces are correlated. It differs from SPA in that a large number of traces are needed, sometimes on the order of millions. To perform a CPA attack, a specific moment in the power trace when the operation of interest is executed is observed. In the case of DES and AES, this moment typically corresponds to the use of Substitution Boxes (S-Box) in the first round of the algorithm [Mangard 2007]. For RSA, common points of attack are modular reduction [Boer 2002], exponentiation [Messerges 1999], and recombination [Witteman 2009].

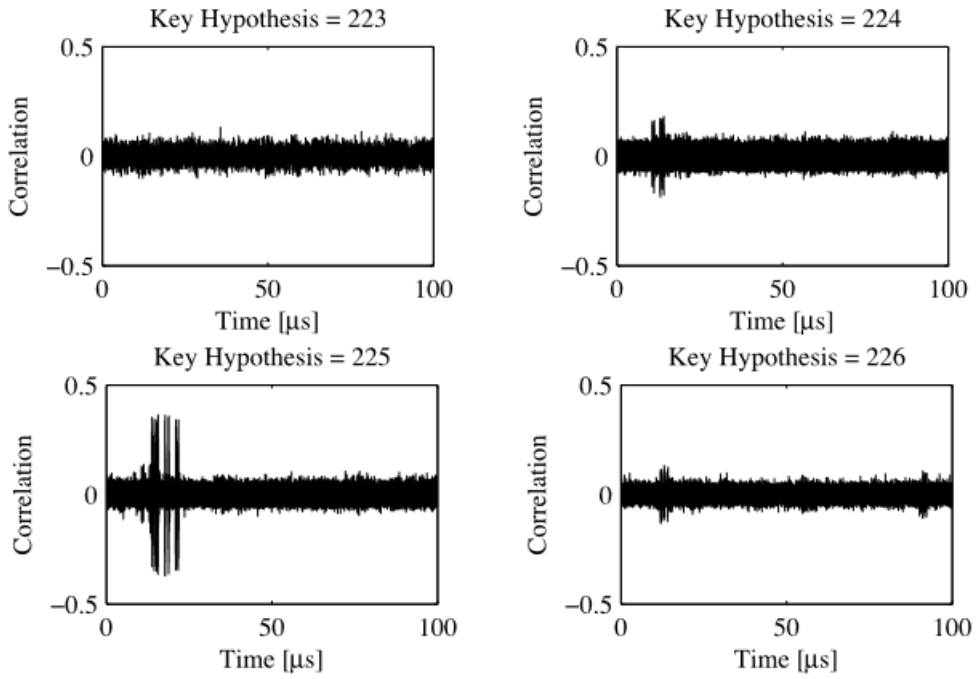

Figure 3.1: CPA Result for Different Intermediate Value Hypotheses [Mangard 2007]

The general strategy for conducting a CPA consists of five steps [Mangard 2007]:

1. Choose an intermediate result of the cryptographic algorithm.

2. Measure power consumption.

3. Calculate hypothetical intermediate values.

4. Apply a power model, for example, calculate the Hamming Weight, to the hypothetical intermediate values.

5. Compare the hypothetical power consumption values with the actual power consumption values from the traces.

In Figure 3.1, we can see the result of a CPA for four hypothetical keys. In the key corresponding to the correct value, correlation peaks are observed.

3.1.3 Countermeasures against Side-channel Attacks on RSA

3.1.3.1 Exponent Obfuscation

CPA attacks target the exponent, which remains fixed across multiple traces. To counter 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 [Clavier 2013]:

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

where r is a random number, and $\phi(n)$ is the Euler totient applied to the modulus n. Using the obfuscated exponent yields the same ciphertext, i.e., $m^d \equiv m^{d'}$.

3.1.3.2 Message Obfuscation

CPA attacks also take advantage of the controllable or known message. To prevent this, we can obfuscate the message before encryption. To do this, a random number r is generated, and with it, r_1 and r_2 are calculated to make the input message unpredictable and correct the final result, respectively [Clavier 2013]:

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

Then during the RSA operation,

$$x' = x \cdot m_1$$
$$y' = x'^d \mod n$$
$$y = y' \cdot m_2$$

3.1.4 Horizontal Attacks vs. Vertical Attacks

Analyses like CPA exploit information vertically, i.e., they attack the same time instant across multiple traces.

Horizontal analyses, on the contrary, analyze different time instants for a single trace.

Figure 3.2 illustrates these two concepts:

Horizontal attacks only require a single trace to extract the exponent. For this reason, the exponent obfuscation countermeasure is ineffective because, as we have seen before, obfuscated exponents are equivalent to the originals for encrypting a message, according to $m^d \equiv m^{d'}$.

Next, we will provide a brief description of the main horizontal attacks.

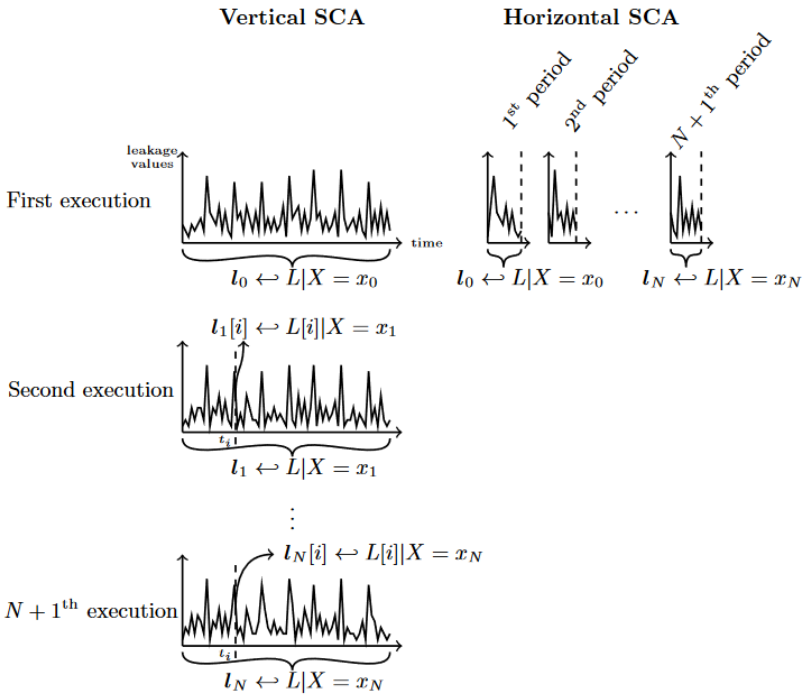

Figure 3.2: Vertical and horizontal attacks. [Bauer 2013]

3.1.4.1 Big Mac Attack

The so-called Big Mac attack is considered the first horizontal attack. The authors explain why they chose this curious name for the attack:

A well-known brand product is so generously large as to be impossible to have a bite taken out of the whole at one go - like the method of attack, it must be nibbled at and consumed by tackling individual layers one by one in any order. [Walter 2001]

The main goal of this attack is to find collisions between exponentiation operations performed in the RSA algorithm. The attack compares these operations to identify which ones are more similar. This allows distinguishing between multiplications with different precalculated values, as well as between multiplications and squares.

To perform this comparison, the attack leverages a characteristic of algorithms for multiplying large numbers. It assumes that a similar algorithm to "textbook multiplication" is used, where the number being multiplied is included in the power consumption trace. This leaked information can be extracted by using what is called leakage, averaging parts of the trace and comparing the resulting vectors with Euclidean distances. Operations with smaller Euclidean distances between them are considered to belong to the same precalculated

value.

It is worth noting that this attack assumes that the multiplication algorithm is known or can be deduced by SPA.

3.1.4.2 Horizontal Correlation Analysis

Unlike the previous attack, where only the trace was used, in the attack proposed by Clavier *et al.* [Clavier 2010], the knowledge of the message is used to make hypotheses about the exponent and try to obtain correlations on the intermediate values of the multiplication operation.

The attack is less realistic and, in practice, cannot be used if the message is obfuscated with a sufficiently large random number.

3.1.4.3 Cross-correlation

Another proposal for a horizontal attack involves reducing the entire modular operation to a single value using a compression operation. Once the trace is reduced to a vector in which each sample is a modular operation, a cross-correlation operation is performed. All samples are compared using Pearson's correlation, and samples with high correlation share the same operand.

In Figure 3.3, the result of this cross-correlation for the algorithm 1 can be seen. Each square corresponds to the comparison between one operation and another. The only operations that can share an operand are multiplications since they always multiply the accumulator register with the initial message. Therefore, it is possible to read the exponent directly.

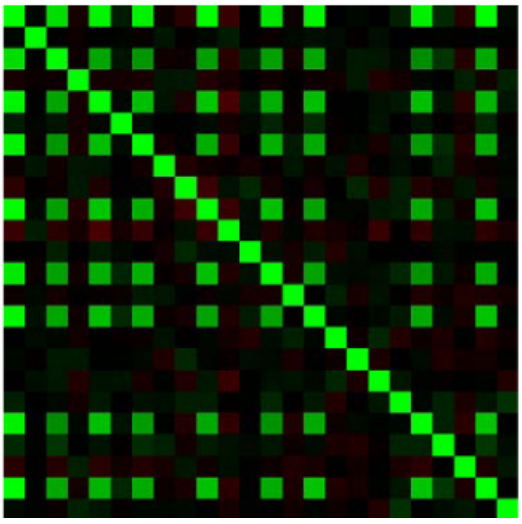

Figure 3.3: Cross-correlation between modular operations.

The article also tests the algorithm 2, detecting the *discard* operations, proving that this type of attack can defeat the Square and Multiply Always countermeasure. [Witteman 2011]

In a more recent conference [Vadnala 2017], the same technique was tested on window exponentiation algorithms such as the algorithm 4.

3.1.4.4 Clustering Analysis

The use of clustering algorithms like k-means has been the main trend in the state of the art during the 2010s. The first authors to propose using this unsupervised technique are Heyszl *et al.* [Heyszl 2013]. Using this technique, they managed to reduce the entropy for a brute-force attack.

A year later, Perin *et al.* [Perin 2014] proposed a framework to perform this type of attack, which is divided into four distinct phases:

1. Preprocess the trace into the necessary sub-traces
2. Find points of interest
3. Group classification using an unsupervised algorithm
4. Exponent recovery

This comprehensive framework provides clear instructions on how to effectively perform this type of attack.

In the following years, the same author proposed additional improvements to the framework, such as using Fuzzy k-Means or the Expectation-Maximization (EM) algorithm instead of k-means, among other enhancements, with the intention of parameterizing the attack and making it less manual. [Perin 2015]

Nascimento *et al.* [Nascimento 2017] used Perin's framework to attack Elliptic Curves instead of RSA. In addition, they added an error correction layer.

It is important to note that all these articles analyze binary exponentiation algorithms, i.e., they work bit by bit and therefore "only" need to distinguish between zeros and ones. Taking this into account, Järvinen and Balasch [Järvinen 2017] proposed a method to extend clustering attacks to window exponentiation methods like algorithms 3 and 4.

Finally, in 2020, Perin revisited his framework by adding an iterative deep learning algorithm, which he claims is able to reduce the error with each iteration until extracting all exponent bits. [Perin 2020]

3.2 Agile Methodology

In this project, we have followed an adaptation of the agile methodology based on a backlog and the assignment of story points to tasks. The agile methodology provides a flexible approach that allows adapting to changes and delivering functionalities incrementally.

3.2.1 Iterations and Incremental Deliveries

The execution of the analysis has been divided into short iterations, known as sprints, each lasting two weeks of work. As we couldn't be fully dedicated to the project, a task was associated with each sprint, and the approximate number of hours spent on each task was counted.

3.2.2 Prioritization and Backlog Management

The backlog is an integral part of the agile methodology, referring to an ordered list of tasks, functionalities, or requirements representing the work set that still needs to be done and is pending to be addressed.

A backlog of tasks has been maintained, and each task has been prioritized based on its importance.

3.2.3 Story Points

Story points are a unit of measurement used in the agile methodology to estimate the complexity or size of a task or functionality. They represent a relative way to compare tasks based on their effort or difficulty.

The Fibonacci sequence (1, 2, 3, 5, 8, 13, 21, ...) is commonly used as it is a growing sequence providing progressively larger values. This progression reflects the increase in relative complexity of tasks as assigned story points increase. The use of this sequence helps avoid assigning overly specific or detailed values, as it can be challenging and imprecise to differentiate between tasks that are very similar in terms of complexity.

3.3 Backlog

In the following tables, we can see the backlog tasks with the corresponding story points, description, subtasks (if applicable), and associated deliverable.

Task Title	Conduct an analysis of public asymmetric cryptography libraries
Story Points	3
Description	Perform an analysis of publicly available libraries for implementing the RSA algorithm. This task aims to identify and evaluate the most popular and widely used libraries and understand their features, functionalities, and security considerations.
Subtasks	- Analyze the μecc library - Analyze the Mbed TLS library
Deliverable	Select a library to work with.

Table 3.1: Task 1

Task Title	Configure CW-Husky to capture RSA traces
Story Points	5
Description	This task involves configuring the environment and necessary devices to capture the trace of an RSA operation using the CW-Husky tool.
Subtasks	- Properly connect the devices - Program target board - Configure capture parameters
Deliverable	RSA trace.

Table 3.2: Task 2

Task Title	Conduct SPA
Story Points	5
Description	Perform a visual analysis of the power trace, identify the regions of RSA, and modular operations.
Subtasks	- Identify sections of RSA: Precomputations, exponentiations, and recombinations. - Identify modular operations
Deliverable	Jupyter notebook with identified regions.

Table 3.3: Task 3

3.4 Project Costs

In the following table 3.6, the costs associated with the project are detailed.

Task Title	Distinguish squares from multiplications
Story Points	8
Description	Use a signal analysis technique to distinguish squares from multiplications. In the Slidingwindow algorithm, if we can distinguish squares from multiplications, we can obtain the bits of the exponent that fall outside the windows.
Deliverable	Partial exponent

Table 3.4: Task 4

Task Title	Classify windows according to value
Story Points	13
Description	Use a signal analysis technique to distinguish which values are processed in each window (10000, 10001, 10010, ... 11110, 11111)
Deliverable	Complete exponent

Table 3.5: Task 5

Concept	**Quantity**	**Unit Cost**	**Cost**
Data Scientist hours	200	€50/hour	€10,000
Computer depreciation cost	200	€0.11/hour	€22
CW308 Target base board and targets	1	€306	€306
CW Husky	1	€550	€550
Other materials and resources	1	€100	€100
Total			**€10,978**

Table 3.6: Project Costs

CHAPTER 4

Methodological Contribution

4.1 Definition of the Proposal

The initial idea for the project was presented at the *NewAE ChipWhisperer Contest 2021* [Micó Biosca 2021]. The objective was to generate datasets to analyze the feasibility of conducting clustering attacks on window algorithms like the k-ary 3, using three different sizes of k (2, 3, and 4).

The proposal was awarded with:

- ChipWhisperer-Husky [NewAE 2023d]
- CW305 Artix FPGA 7A35 Target Board [NewAE 2023a]
- A signed copy of the book *The Hardware Hacking Handbook* [Woudenberg 2021]

The first task involved searching for public implementations of RSA or ECC for an FPGA. Despite finding some, the difficulty of programming an FPGA to adapt the code led me to dismiss this proposal.

After discussing it with Jean-Pierre Thibault (Senior Security Engineer) and Colin O'Flynn (CEO) at NewAE Technology Inc., I decided to shift the focus of the work towards attacking a public library and presenting the results in this thesis.

4.2 Method Proposed for the Analysis

1. Analyze different 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, identify the RSA regions, and modular operations.
4. Develop a method to distinguish squares from multiplications.
5. Develop a method to distinguish the different precalculated values.

6. In case of obtaining satisfactory results, inform the library developers of the vulnerability.

4.3 Reproducibility of the Analysis

The code and trace used in this work can be found at the following GitHub address: victormico/sca-mbedtls-rsa

Chapter 5
Results

In this chapter, we present the attack carried out on the RSA-CRT implementation of the *Mbed TLS* library. This library employs the *Sliding Windows* algorithm 4 with a window size of *5* bits for exponentiation.

The library is installed on an STM32F3 board 2.4, mounted on the CW308 motherboard 2.3, facilitating power consumption extraction.

The Chipwhisperer Husky 2.2 has been used as the capture device, configured for synchronous and streaming capture. **1,092,000** samples were captured at a sampling rate of **29.48 MS/s**.

In Section 5.1, we explain the SPA performed and how we identified the start pattern of each bit window. This pattern is used to perform a pattern matching analysis on the entire trace, as explained in Section 5.2. Through this analysis, we can distinguish which modular operations belong to the bit window and which do not. With this initial step, we are able to obtain approximately **30%** of the bits of the exponents dp and dq.

Finally, we identified the memory load for the 5 bits of the window. There is a differentiated pattern for loading a 0 and for loading a 1. Using the same pattern matching technique, we can identify the 5 bits of each window and thus extract approximately **100%** of the bits of the exponents dp and dq.

5.1 SPA

The first phase of SPA is to identify the two exponentiations of RSA-CRT. In Figure 5.1, the complete trace is shown with the two identified exponentiations.

If we zoom in on the start of the first exponentiation, as shown in Figure 5.2, we can see about 15 prominent peaks. These correspond to the 15 multiplications needed to calculate the precomputed values.

In the initial region of the exponentiation, which is enlarged in Figure 5.3, we can observe small repeating peaks. We also notice some distinct peaks. As the raw trace does not reveal more differences at this level, we apply a lowpass filter 5 to the signal.

After trying different values, we settle on the lowpass filter with a weight of 100. In Figure 5.4, we show an enlarged view of the area indicated as distinct

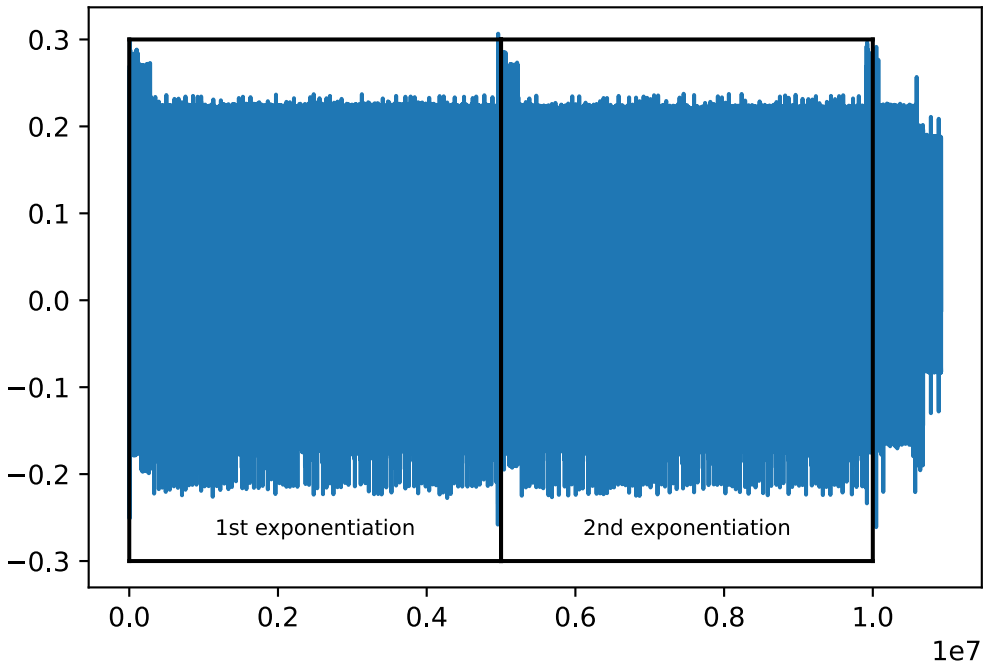

Figure 5.1: Complete RSA Trace

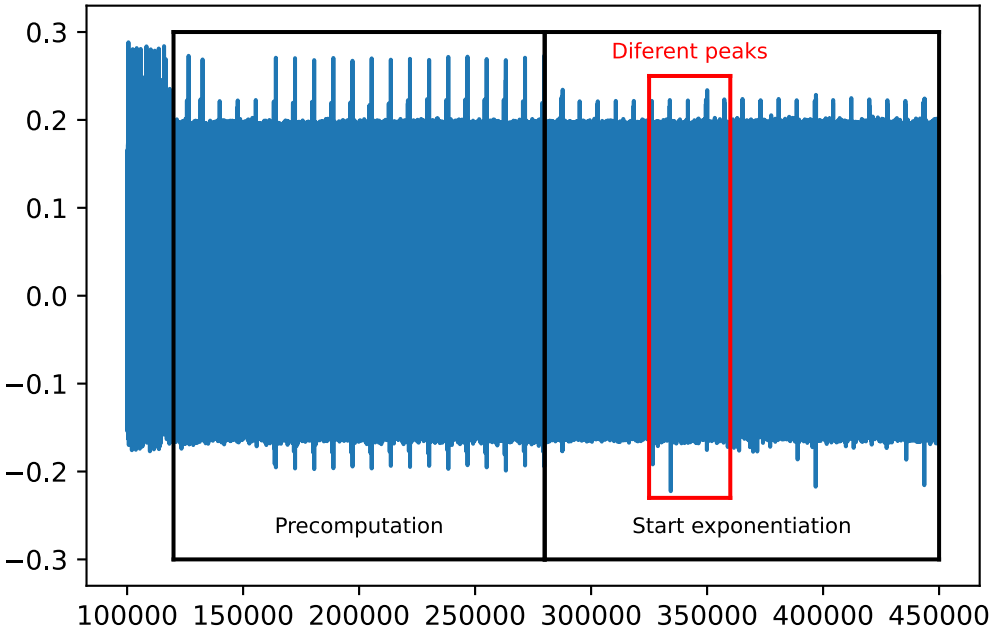

Figure 5.2: Precomputations and Start of Exponentiation

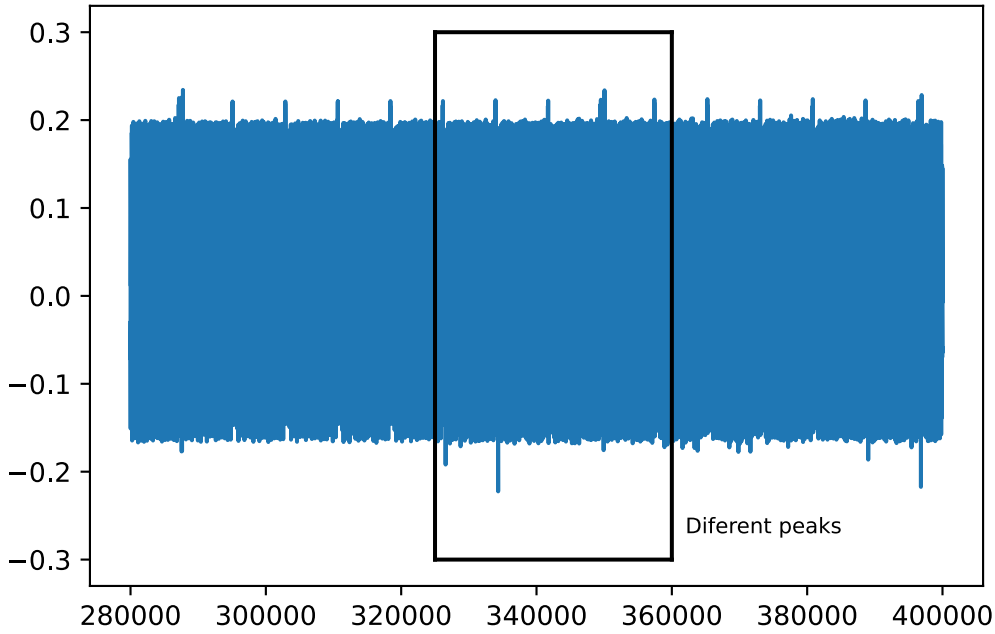

Figure 5.3: Distinct Peaks between Modular Operations

peaks in the previous figure, with the lowpass filter applied. In this figure, we highlight a pattern that stands out for its wider shape.

By observing the repetition frequency of the wider pattern, we hypothesize that it represents the start of the bit window. To confirm this, we conduct a pattern matching analysis.

5.2 Pattern Matching

Pattern matching involves comparing a pattern with the rest of the trace. The algorithm 6 describes the process. The pattern and the corresponding trace section are compared using the Pearson correlation coefficient [SciPy 2023b]. This process is repeated by moving the trace section in one-sample increments. The result of this analysis is a trace where each sample indicates the probability that that region resembles the pattern.

5.2.1 Identification of Squares and Multiplications

Figure 5.5 details the pattern we indicated in Figure 5.4. This pattern has been compared with the entire trace to obtain the instances in which the same operation occurs during exponentiation.

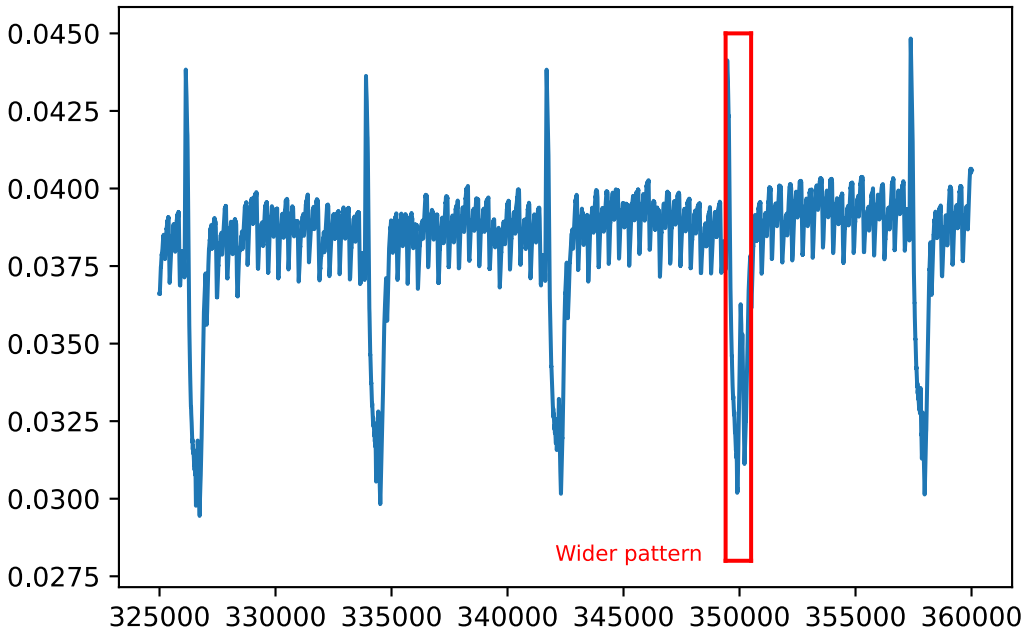

Figure 5.4: Trace filtered with *lowpass*

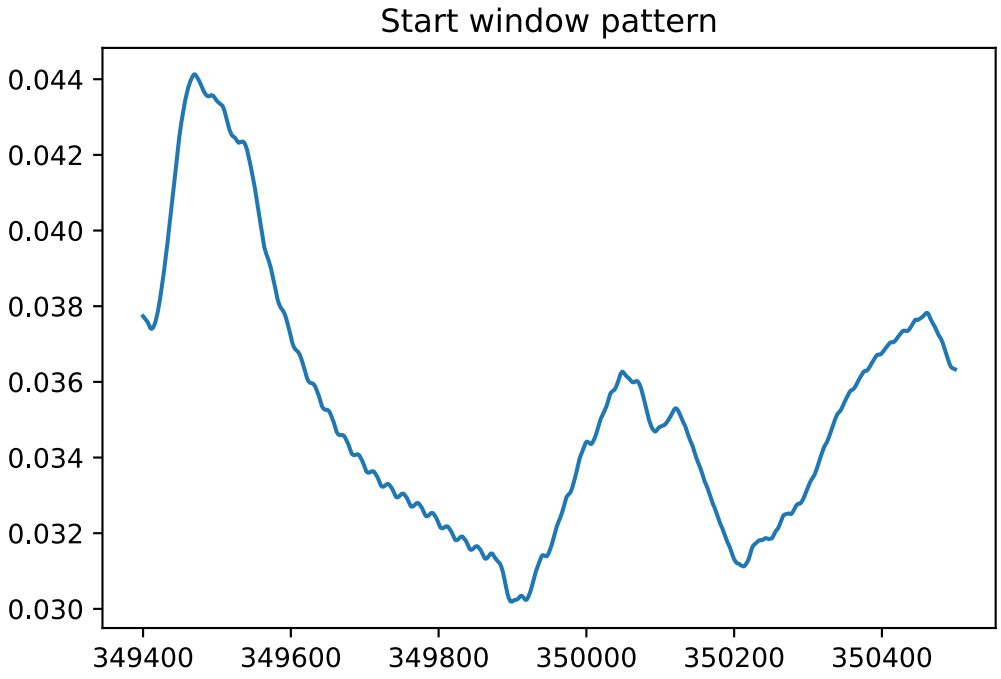

Figure 5.5: Pattern Start of Window

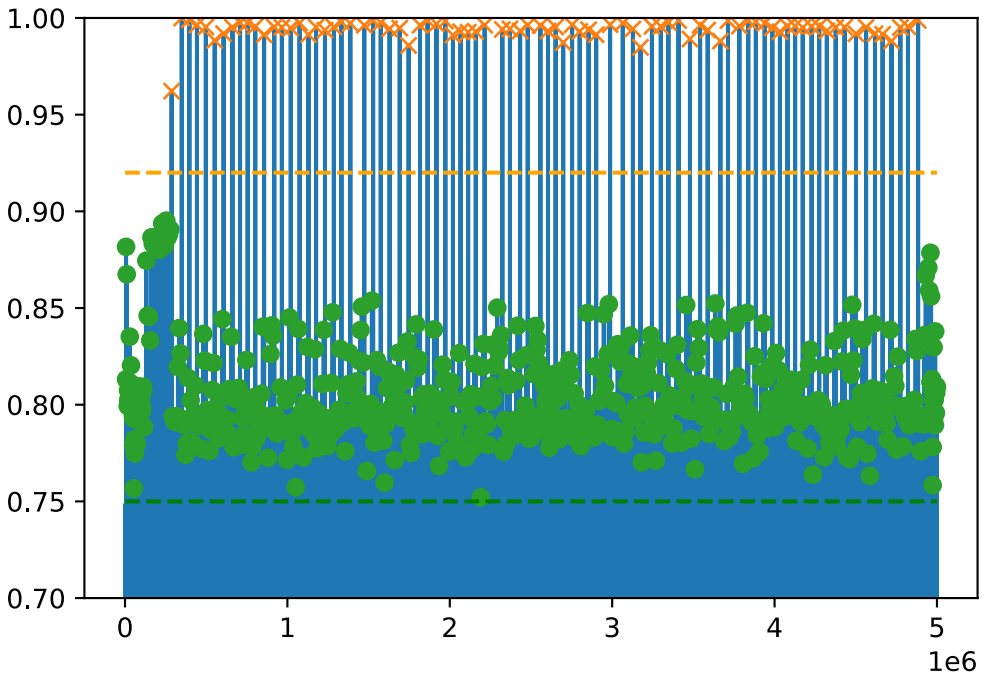

Figure 5.6: Result of Pattern Matching.

In Figure 5.6, we see the result of pattern matching applied to the first exponentiation. We can clearly observe how the pattern repeats. To extract the indices, we use the *find peaks* function from the *SciPy* library [SciPy 2023a], which allows specifying a threshold and returns the index where peaks above this threshold occur. In this case, we extracted two types of peaks. On one hand, the peaks above the threshold of 0.9 (marked in orange in the figure), indicating the start of each window. The second type of peaks lies between 0.75 and 0.9 (marked in green in the figure), indicating the start of a modular operation (square or multiplication).

Counting the number of operations between the start of each window, we can determine which operations are squares or multiplications. Note that the minimum number of green peaks (modular operations) between two orange peaks (window start) is 6, representing the operations needed in a 5-bit sliding window, SQ-SQ-SQ-SQ-SQ-MUL (where *SQ* is a square and *MUL* is a multiplication). This confirms the hypothesis that the pattern corresponded to the start of the window. If there are additional green points, it means we have multiplications outside the sliding windows, corresponding to when a 0 is in the exponent.

In Figure 5.7, we can see this process more clearly. The filtered trace is represented in the figure, and the dashed lines indicate the points identified with the pattern matching method. In orange for the start of the window and in green

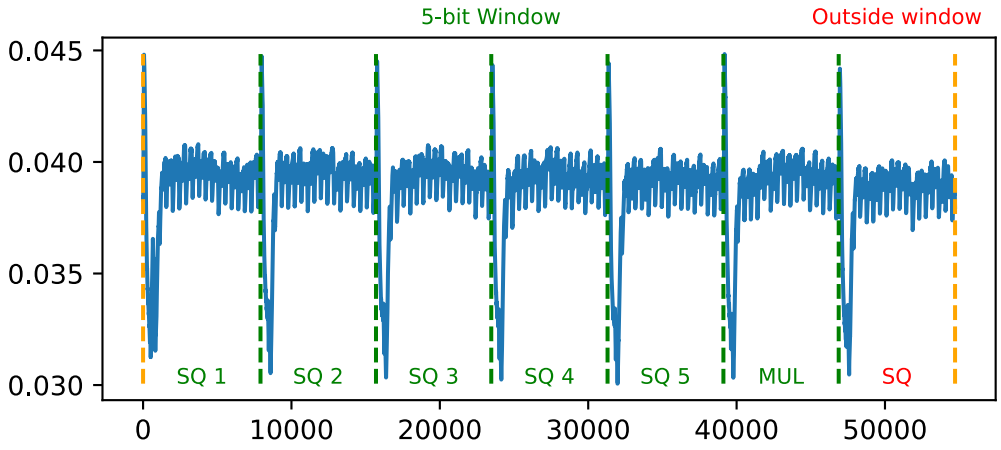

Figure 5.7: Identification of Modular Operations.

for the modular operations.

With this method, we have been able to obtain some of the bits of each exponentiation: those corresponding to the first bit of the window, which is always 1, and those outside the windows, which are 0.

5.2.1.1 Bits Obtained from the First Exponentiation

1xxxx001xxxx1xxxx1xxxx01xxxx01xxxx01xxxx01xxxx1xxxx1xxxx1xxxx01x
xxx001xxxx1xxxx01xxxx01xxxx01xxxx1xxxx01xxxx01xxxx1xxxx01xxxx000
001xxxx01xxxx1xxxx01xxxx001xxxx01xxxx0001xxxx1xxxx01xxxx01xxxx1x
xxx1xxxx1xxxx1xxxx01xxxx000000001xxxx1xxxx001xxxx1xxxx00001xxxx1
xxxx1xxxx1xxxx01xxxx1xxxx01xxxx1xxxx000001xxxx00001xxxx001xxxx1x
xxx0001xxxx01xxxx1xxxx001xxxx0001xxxx001xxxx1xxxx00001xxxx1xxxx0
001xxxx01xxxx1xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx01xxxx1xxxx1xxx
x01xxxx01xxxx0001xxxx001xxxx01xxxx1xxxx01xxxx01xxxx1xxxx00xxxxxx

5.2.1.2 Bits Obtained from the Second Exponentiation

1xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx0001xxxx1xxxx1xxxx1xxxx00001
xxxx1xxxx1xxxx01xxxx1xxxx1xxxx00001xxxx01xxxx01xxxx1xxxx1xxxx1xx
xx1xxxx1xxxx01xxxx01xxxx1xxxx1xxxx1xxxx1xxxx1xxxx1xxxx01xxxx1xxx
x1xxxx1xxxx01xxxx1xxxx001xxxx1xxxx1xxxx000001xxxx01xxxx1xxxx1xxx
x1xxxx00001xxxx0001xxxx0001xxxx1xxxx01xxxx01xxxx1xxxx1xxxx1xxxx0
1xxxx0001xxxx1xxxx1xxxx1xxxx1xxxx01xxxx001xxxx1xxxx0001xxxx1xxxx
1xxxx0001xxxx1xxxx1xxxx1xxxx1xxxx1xxxx00001xxxx01xxxx1xxxx1xxxx1
xxxx01xxxx01xxxx01xxxx01xxxx01xxxx0001xxxx001xxxx00001xxxxxxxxx

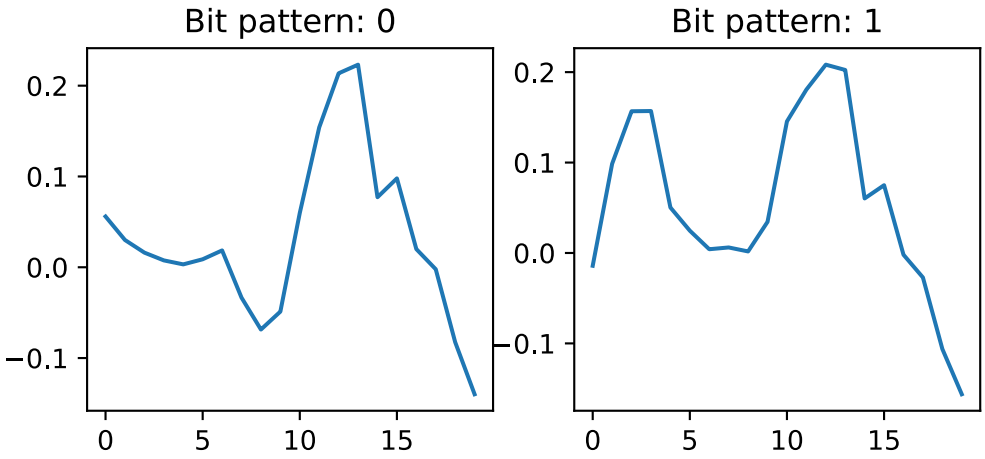

Figure 5.8: Patterns corresponding to the loading of a zero and a one.

5.2.2 Identification of Bits within a Window

Once each modular operation was identified, it was possible to determine the trace region where the individual bits of the window were loaded. The load corresponds to the following line of code:

```
wbits |= ( ei << ( wsize - nbits ) );
```

Where:

- *wbits* is a variable used to store a window of bits.
- *ei* is the current bit being processed.
- *wsize* is the size of the window, determining how many bits are processed together in each iteration, in this case, 5.
- *nbits* is the number of bits processed in the current window.

The line performs a bitwise OR (|) operation between *wbits* and (*ei* « (*wsize* - *nbits*)). The expression (*ei* « (*wsize* - *nbits*)) shifts the bit *ei* to the left to the appropriate position within the window, depending on the number of bits processed. Then, the result is combined with the current content of *wbits* using the OR operation, and the result is stored again in *wbits*.

This processing produces a different power consumption when storing a zero or a one. In Figure 5.8, we can see the corresponding patterns.

Using the pattern matching method, we can distinguish each bit of the window. In Figure 5.9, we can see the result of comparing the previous patterns with the bit loading process.

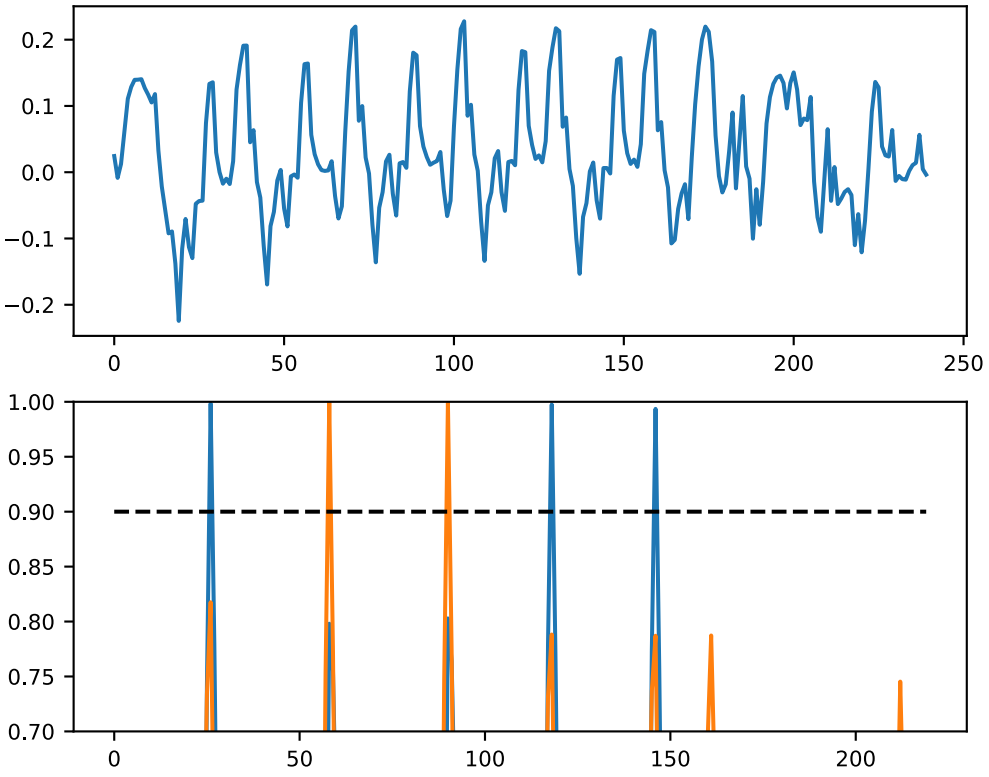

Figure 5.9: Top: Trace segment corresponding to the loading of window bits Bottom: Result of pattern matching for zero and one bits.

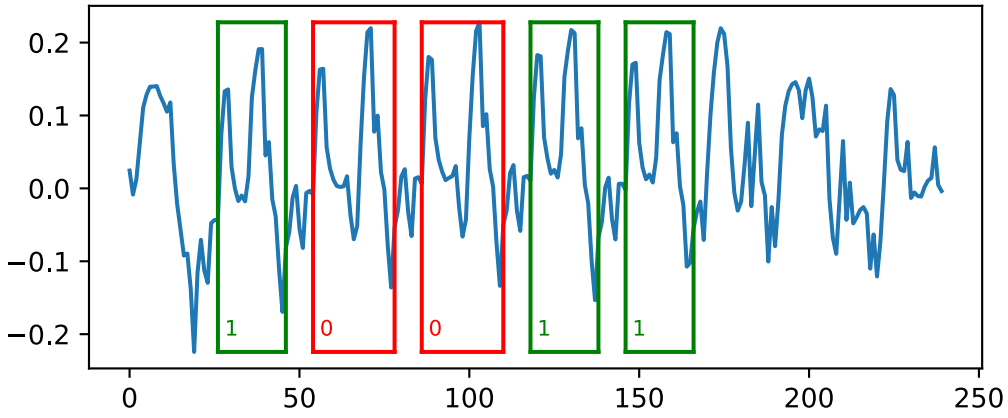

Figure 5.10: Identification of individual loading of each bit in the window.

The result is the complete identification of the bits in each window. As shown in Figure 5.10

Automating the process for each window and combining the bits we had extracted earlier yields the following result:

5.2.2.1 Bits Obtained from the First Exponentiation

1100000110101100111010101100111010101100100001001110100111110110000011110100000101110111010110101011101001001101110001010011000011110010010100110100100010001010001000101011001101011101100011111111011000111001010110000000001110110101001010011100000010110111111000010000011001100010100101111100000110110000101110011100100100011011010101110010010010000011001001001111110000011111111000001111100110001010101100111001101001001111011110000010110100001110010010001000100000010001011100110000101110110101011100100101x

5.2.2.2 Bits Obtained from the Second Exponentiation

1001001010110011100101111110110111110001111111011110111010100001010111011100110100101110111111000010101010101010111100111000111110110000101010101110101001010110011101001001111010100001010000110001000110011001011100100111011111101000001011001001010101111010111000101001000111011111111100101100100101111010100010111101101000001111010000110111110101110110110000010110010010111001001110111111010000101100100101111010100010111101101000001111010000110111110101110110110000010110010010111001001111011010001011010011001010100011100000111110010110000011010101111

5.3 Summary of Results

The following table shows a summary of the results obtained for both exponentiations during the two phases of the attack.

	1st Exponentiation	2nd Exponentiation
Distinguish squares from multiplications	33.98%	30.91%
Distinguish bits in each window	99.80%	100%

Table 5.1: Summary of Results

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

In this work, an attack on version 2.5.1 of the Mbed TLS library, which implements an RSA-CRT exponentiation algorithm using the Sliding Windows method, has been carried out.

The attack is based on capturing a trace of the RSA operation, and a SPA has been performed to identify both exponentiations. Through pattern matching analysis, the sequence of modular operations has been inferred, distinguishing between squares and multiplications.

From this phase, approximately 30% of the bits of each exponent have been obtained. By identifying the beginning of each window, the region where the bits are loaded has been located. Using the pattern matching technique, all the bits of both exponents have been determined.

This work demonstrates that side-channel attacks are feasible, can be carried out on a tight budget, and with relatively simple signal processing methods.

6.2 Future Work

As future work, the following is proposed:

1. Update the code of the Mbed TLS library to the latest version to verify if it is possible to exploit this vulnerability.

2. Test alternative target devices other than the STM32F3.

3. Explore other techniques to extract exponent values, such as clustering algorithms.

These initiatives will expand the knowledge about the detected vulnerability and seek more effective solutions to protect cryptographic implementations based on the Mbed TLS library.

Bibliography

[Amiel 2009] Frederic Amiel, Benoit Feix, Michael Tunstall, Claire Whelan and William P. Marnane. *Distinguishing Multiplications from Squaring Operations*. In Roberto Maria Avanzi, Liam Keliher and Francesco Sica, editeurs, Selected Areas in Cryptography, Lecture Notes in Computer Science, pages 346–360, Berlin, Heidelberg, 2009. Springer. (Cited on page 8.)

[Aumasson 2017] Jean-Philippe Aumasson and Matthew D. Green. Serious cryptography: a practical introduction to modern encryption. No Starch Press, San Francisco, 2017. OCLC: ocn986236585. (Cited on page 1.)

[Bauer 2013] Aurélie Bauer, Eliane Jaulmes, Emmanuel Prouff and Justine Wild. *Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations*. In Ed Dawson, editeur, Topics in Cryptology – CT-RSA 2013, Lecture Notes in Computer Science, pages 1–17, Berlin, Heidelberg, 2013. Springer. (Cited on pages vii and 18.)

[Boer 2002] Bert Boer, Kerstin Lemke and Guntram Wicke. *A DPA attack against the modular reduction within a CRT implementation of RSA*. volume 2523, pages 228–243, August 2002. (Cited on page 15.)

[Clavier 2010] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet and Vincent Verneuil. *Horizontal Correlation Analysis on Exponentiation*. In Miguel Soriano, Sihan Qing and Javier López, editeurs, Information and Communications Security, Lecture Notes in Computer Science, pages 46–61, Berlin, Heidelberg, 2010. Springer. (Cited on page 19.)

[Clavier 2013] Christophe Clavier and Benoit Feix. *Updated Recommendations for Blinded Exponentiation vs. Single Trace Analysis*. In Emmanuel Prouff, editeur, Constructive Side-Channel Analysis and Secure Design, Lecture Notes in Computer Science, pages 80–98, Berlin, Heidelberg, 2013. Springer. (Cited on page 17.)

[Coron 1999] Jean-Sébastien Coron. *Resistance Against Differential Power Analysis For Elliptic Curve Cryptosystems*. In Çetin K. Koç and Christof Paar, editeurs, Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, pages 292–302, Berlin, Heidelberg, 1999. Springer. (Cited on page 7.)

[Heyszl 2013] Johann Heyszl, Andreas Ibing, Stefan Mangard, Fabrizio De Santis and Georg Sigl. *Clustering Algorithms for Non-Profiled Single-Execution Attacks on Exponentiations*. Technical report 438, 2013. (Cited on page 20.)

[Järvinen 2017] Kimmo Järvinen and Josep Balasch. *Single-Trace Side-Channel Attacks on Scalar Multiplications with Precomputations*. In Kerstin Lemke-Rust and Michael Tunstall, editeurs, Smart Card Research and Advanced Applications, Lecture Notes in Computer Science, pages 137–155, Cham, 2017. Springer International Publishing. (Cited on pages 8 and 20.)

[Kocher 1999] Paul Kocher, Joshua Jaffe and Benjamin Jun. *Differential Power Analysis*. In Michael Wiener, editeur, Advances in Cryptology — CRYPTO' 99, Lecture Notes in Computer Science, pages 388–397, Berlin, Heidelberg, 1999. Springer. (Cited on pages 6 and 15.)

[Mangard 2007] Stefan Mangard, Elisabeth Oswald and Thomas Popp. Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer US, 2007. (Cited on pages vii, 15 and 16.)

[Menezes 2007] Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone. Handbook of applied cryptography. CRC Press, 2007. (Cited on pages 6 and 9.)

[Messerges 1999] Thomas S. Messerges, Ezzy A. Dabbish and Robert H. Sloan. *Power Analysis Attacks of Modular Exponentiation in Smartcards*. In Çetin K. Koç and Christof Paar, editeurs, Cryptographic Hardware and Embedded Systems, Lecture Notes in Computer Science, pages 144–157, Berlin, Heidelberg, 1999. Springer. (Cited on page 15.)

[Micó Biosca 2021] Victor Micó Biosca. *Clustering attacks on k-ary implementations of public-key crypto algorithms*. https://github.com/newaetech/chipwhisperer-contest-2021/issues/4, December 2021. original-date: 2021-12-02T15:10:31Z. (Cited on page 25.)

[Nascimento 2017] Erick Nascimento and Łukasz Chmielewski. *Applying Horizontal Clustering Side-Channel Attacks on Embedded ECC Implementations (Extended Version)*. Technical report, 2017. (Cited on page 20.)

[NewAE 2023a] NewAE. *NAE-CW305*. https://www.newae.com/products/NAE-CW305, 2023. (Cited on page 25.)

[NewAE 2023b] NewAE. *NAE-CW308*. https://www.newae.com/products/NAE-CW308, 2023. (Cited on pages vii and 13.)

[NewAE 2023c] NewAE. *NAE-CW308T-STM32F3*. https://www.newae.com/ufo-target-pages/NAE-CW308T-STM32F3, 2023. (Cited on pages vii and 14.)

[NewAE 2023d] NewAE. *NAE-CWHUSKY*. https://www.newae.com/products/NAE-CWHUSKY, 2023. (Cited on pages vii, 12 and 25.)

[O'Flynn 2017] Colin O'Flynn. *A Framework for Embedded Hardware Security Analysis*. July 2017. (Cited on page 10.)

[Paar 2010] Christof Paar and Jan Pelzl. Understanding cryptography: a textbook for students and practitioners. Springer, Heidelberg ; New York, 2010. OCLC: ocn527339793. (Cited on pages vii, 7 and 9.)

[Perin 2014] Guilherme Perin, Laurent Imbert, Lionel Torres and Philippe Maurine. *Attacking Randomized Exponentiations Using Unsupervised Learning*. In Emmanuel Prouff, editeur, Constructive Side-Channel Analysis and Secure Design, volume 8622, pages 144–160. Springer International Publishing, Cham, 2014. Series Title: Lecture Notes in Computer Science. (Cited on page 20.)

[Perin 2015] Guilherme Perin and Lukasz Chmielewski. *A Semi-Parametric Approach for Side-Channel Attacks on Protected RSA Implementations*. 2015. (Cited on page 20.)

[Perin 2020] Guilherme Perin, Lukasz Chmielewski, Lejla Batina and Stjepan Picek. *Keep it Unsupervised: Horizontal Attacks Meet Deep Learning*. Technical report 891, 2020. (Cited on page 20.)

[SciPy 2023a] SciPy. *scipy.signal.find_peaks — SciPy v1.10.1 Manual*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.find_peaks.html, 2023. (Cited on page 31.)

[SciPy 2023b] SciPy. *scipy.stats.pearsonr — SciPy v1.10.1 Manual*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.pearsonr.html, 2023. (Cited on page 29.)

[Vadnala 2017] Praveen Vadnala and Lukasz Chmielewski. *Attacking OpenSSL using Side-channel Attacks*, August 2017. (Cited on page 20.)

[van Tilborg 2014] H.C.A. van Tilborg and S. Jajodia. Encyclopedia of cryptography and security. Encyclopedia of Cryptography and Security. Springer US, 2014. (Cited on page 2.)

[Walter 2001] C. D. Walter. *Sliding Windows Succumbs to Big Mac Attack*. In Çetin K. Koç, David Naccache and Christof Paar, editeurs, Cryptographic Hardware and Embedded Systems — CHES 2001, Lecture Notes in Computer Science, pages 286–299, Berlin, Heidelberg, 2001. Springer. (Cited on page 18.)

[Witteman 2009] Marc Witteman. *A DPA attack on RSA in CRT mode*. 2009. (Cited on page 15.)

[Witteman 2011] Marc F. Witteman, Jasper G. J. van Woudenberg and Federico Menarini. *Defeating RSA Multiply-Always and Message Blinding Countermeasures*. In Aggelos Kiayias, editeur, Topics in Cryptology – CT-RSA 2011, Lecture Notes in Computer Science, pages 77–88, Berlin, Heidelberg, 2011. Springer. (Cited on page 20.)

[Woudenberg 2021] Jasper van Woudenberg and Colin O'Flynn. The Hardware Hacking Handbook: Breaking Embedded Security with Hardware Attacks. No Starch Press, December 2021. Google-Books-ID: DEqatAEACAAJ. (Cited on page 25.)

Annexes

6.3 Mbed TLS Modular Exponentiation Function

Listing 6.1: Mbed TLS modular exponentiation function

```
/*
 * Sliding-window exponentiation: X = A^E mod N  (HAC 14.85)
 */
int mbedtls_mpi_exp_mod( mbedtls_mpi *X, const mbedtls_mpi *A,
const mbedtls_mpi *E, const mbedtls_mpi *N, mbedtls_mpi *_RR )
{
int ret;
size_t wbits, wsize, one = 1;
size_t i, j, nblimbs;
size_t bufsize, nbits;
mbedtls_mpi_uint ei, mm, state;
mbedtls_mpi RR, T, W[ 2 << MBEDTLS_MPI_WINDOW_SIZE ], Apos;
int neg;

if( mbedtls_mpi_cmp_int( N, 0 ) < 0 || ( N->p[0] & 1 ) == 0 )
    return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );

if( mbedtls_mpi_cmp_int( E, 0 ) < 0 )
    return( MBEDTLS_ERR_MPI_BAD_INPUT_DATA );

/*
 * Init temps and window size
 */
mpi_montg_init( &mm, N );
mbedtls_mpi_init( &RR ); mbedtls_mpi_init( &T );
mbedtls_mpi_init( &Apos );
memset( W, 0, sizeof( W ) );

i = mbedtls_mpi_bitlen( E );

wsize = ( i > 671 ) ? 6 : ( i > 239 ) ? 5 :
```

```
        ( i >  79 ) ? 4 : ( i >  23 ) ? 3 : 1;

if( wsize > MBEDTLS_MPI_WINDOW_SIZE )
    wsize = MBEDTLS_MPI_WINDOW_SIZE;

j = N->n + 1;
MBEDTLS_MPI_CHK( mbedtls_mpi_grow( X, j ) );
MBEDTLS_MPI_CHK( mbedtls_mpi_grow( &W[1],  j ) );
MBEDTLS_MPI_CHK( mbedtls_mpi_grow( &T, j * 2 ) );

/*
 * Compensate for negative A (and correct at the end)
 */
neg = ( A->s == -1 );
if( neg )
{
    MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &Apos, A ) );
    Apos.s = 1;
    A = &Apos;
}

/*
 * If 1st call, pre-compute R^2 mod N
 */
if( _RR == NULL || _RR->p == NULL )
{
    MBEDTLS_MPI_CHK( mbedtls_mpi_lset( &RR, 1 ) );
    MBEDTLS_MPI_CHK( mbedtls_mpi_shift_l( &RR, N->n * 2 * biL ) );
    MBEDTLS_MPI_CHK( mbedtls_mpi_mod_mpi( &RR, &RR, N ) );

    if( _RR != NULL )
        memcpy( _RR, &RR, sizeof( mbedtls_mpi ) );
}
else
    memcpy( &RR, _RR, sizeof( mbedtls_mpi ) );

/*
 * W[1] = A * R^2 * R^-1 mod N = A * R mod N
 */
if( mbedtls_mpi_cmp_mpi( A, N ) >= 0 )
    MBEDTLS_MPI_CHK( mbedtls_mpi_mod_mpi( &W[1], A, N ) );
```

```
else
    MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &W[1], A ) );

MBEDTLS_MPI_CHK( mpi_montmul( &W[1], &RR, N, mm, &T ) );

/*
 * X = R^2 * R^-1 mod N = R mod N
 */
MBEDTLS_MPI_CHK( mbedtls_mpi_copy( X, &RR ) );
MBEDTLS_MPI_CHK( mpi_montred( X, N, mm, &T ) );

if( wsize > 1 )
{
    /*
     * W[1 << (wsize - 1)] = W[1] ^ (wsize - 1)
     */
    j =  one << ( wsize - 1 );

    MBEDTLS_MPI_CHK( mbedtls_mpi_grow( &W[j], N->n + 1 ) );
    MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &W[j], &W[1]    ) );

    for( i = 0; i < wsize - 1; i++ )
        MBEDTLS_MPI_CHK( mpi_montmul( &W[j], &W[j], N, mm, &T ) );

    /*
     * W[i] = W[i - 1] * W[1]
     */
    for( i = j + 1; i < ( one << wsize ); i++ )
    {
        MBEDTLS_MPI_CHK( mbedtls_mpi_grow( &W[i], N->n + 1 ) );
        MBEDTLS_MPI_CHK( mbedtls_mpi_copy( &W[i], &W[i - 1] ) );

        MBEDTLS_MPI_CHK( mpi_montmul( &W[i], &W[1], N, mm, &T ) );
    }
}

nblimbs = E->n;
bufsize = 0;
nbits   = 0;
wbits   = 0;
state   = 0;
```

```
while( 1 )
{
    if( bufsize == 0 )
    {
        if( nblimbs == 0 )
            break;

        nblimbs--;

        bufsize = sizeof( mbedtls_mpi_uint ) << 3;
    }

    bufsize--;

    ei = (E->p[nblimbs] >> bufsize) & 1;

    /*
     * skip leading 0s
     */
    if( ei == 0 && state == 0 )
        continue;

    if( ei == 0 && state == 1 )
    {
        /*
         * out of window, square X
         */
        MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );
        continue;
    }

    /*
     * add ei to current window
     */
    state = 2;

    nbits++;
    wbits |= ( ei << ( wsize - nbits ) );

    if( nbits == wsize )
```

```
    {
        /*
         * X = X^wsize R^-1 mod N
         */
        for( i = 0; i < wsize; i++ )
            MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );

        /*
         * X = X * W[wbits] R^-1 mod N
         */
        MBEDTLS_MPI_CHK( mpi_montmul( X, &W[wbits], N, mm, &T ) );

        state--;
        nbits = 0;
        wbits = 0;
    }
}

/*
 * process the remaining bits
 */
for( i = 0; i < nbits; i++ )
{
    MBEDTLS_MPI_CHK( mpi_montmul( X, X, N, mm, &T ) );

    wbits <<= 1;

    if( ( wbits & ( one << wsize ) ) != 0 )
        MBEDTLS_MPI_CHK( mpi_montmul( X, &W[1], N, mm, &T ) );
}

/*
 * X = A^E * R * R^-1 mod N = A^E mod N
 */
MBEDTLS_MPI_CHK( mpi_montred( X, N, mm, &T ) );

if( neg && E->n != 0 && ( E->p[0] & 1 ) != 0 )
{
    X->s = -1;
    MBEDTLS_MPI_CHK( mbedtls_mpi_add_mpi( X, N, X ) );
}
```

```
cleanup:

for( i = ( one << ( wsize - 1 ) ); i < ( one << wsize ); i++ )
    mbedtls_mpi_free( &W[i] );

mbedtls_mpi_free( &W[1] );
mbedtls_mpi_free( &T );
mbedtls_mpi_free( &Apos );

if( _RR == NULL || _RR->p == NULL )
    mbedtls_mpi_free( &RR );

return( ret );
}
```

6.4 Signal Processing Algorithms

Algorithm 5 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 $result$

Algorithm 6 Pattern match

Require: t as trace
Require: ref as Reference pattern
Ensure: $scores$
  $N \leftarrow length(trace)$
  $n \leftarrow length(ref)$
  for $i \leftarrow 1 to N$ **do**
    $score[i] \leftarrow corr(ref, trace(i, i+n)$
  return $score$