Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Master of Science in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

Side channel attacks to LEDAcrypt: synthetic analysis and practical countermeasure validation

Supervisor: Prof. Gerardo PELOSI
Co-supervisor: Prof. Alessandro BARENGHI

Master Thesis by:
Simone BERGONZI Matr. 899250

Anno Accademico 2019–2020

Sommario

Con la diffusione di dispositivi embedded che gestiscono informazioni sensibili, la crittografia moderna non può più evitare di considerare la dimensione fisica nel progettare una primitiva crittografica. Infatti, ogni dispositivo introduce nuove superfici di attacco dovute ai così detti *side channels*, l'insieme di parametri fisici che possono rivelare dati sensibili dall'esecuzione di un algoritmo, tra cui la potenza dissipata, il tempo di calcolo, le emissioni elettromagnetiche.

Un'altra minaccia per la crittografia moderna è rappresentata dall'avvento del computer quantistico. Questa nuova tecnologia infatti, sarebbe in grado di inficiare la sicurezza computazionale delle moderne primitive crittografiche basate su due noti problemi matematici: la scomposizione in fattori primi e il calcolo del logaritmo discreto. In risposta a questa minaccia, i sistemi crittografici basati su codici a correzione di errori rappresentano una promettente area di ricerca.

Il seguente studio vuole investigare lo stato dell'arte degli attacchi che sfruttano la potenza dissipata durante l'esecuzione di sistemi crittografici basati su codici (in particolare, codici di tipo QC-MDPC/LDPC), col fine ultimo di valutare la sicurezza sotto questo fronte di LEDAcrypt. LEDAcrypt è un sistema crittografico che offre sia una primitiva di incapsulamento per chiavi di sessione, sia un cifrario a chiave pubblica basati su codici di tipo QC-LDPC.

Dall'analisi dello stato dell'arte, abbiamo selezionato l'attacco proposto da Sim et al. (2019, [46]). Si tratta di un attacco basato sulla traccia di una singola esecuzione dell'algoritmo, e quindi, in grado di minacciare anche l'uso di chiavi effimere. Abbiamo successivamente simulato l'attacco e studiato una contromisura per mettere in sicurezza LEDAcrypt. In fine,

abbiamo eseguito l'attacco su un dispositivo fisico e quindi analizzato gli effetti della nostra contromisura.

In conclusione, da una parte abbiamo validato l'attacco a traccia singola di Sim et al., spiegando anche il modello di leakage utilizzato dagli autori, e dall'altra abbiamo sviluppato una contromisura basata sul *random precharging* in grado di mettere in sicurezza l'algoritmo.

Abstract

With the diffusion of cryptographic devices such as smart cards, which handle sensitive information every day, modern cryptography must take into account also the physical aspects concerning the execution of a cryptographic primitive on such devices. In fact, a well-known family of attacks, called *side-channel attacks*, exploit physical measures such as power consumption, electromagnetic emissions, time computation, to recover secret data.

Another important threat to modern cryptography is the coming of quantum computers. This technology would ensure the computational power required to solve two important mathematical problems which are at the basis of the modern cryptography: integer factorization problem and discrete logarithm problem. To overcome this threat, researchers are studying new cryptosystems based on different mathematical problems. Among the new proposals, QC-MDPC/LDPC code-based cryptosystems seem to be a promising research area.

This work aims to investigate the state-of-the-art power analysis attacks against QC-MDPC/LDPC code-based cryptosystems and, in this way, evaluate the power analysis resistance of LEDAcrypt. LEDAcrypt is a QC-LDPC code-based cryptosystems that provides both a public key cryptoscheme and a key encapsulation method (also optimized for use in an ephemeral key scenario).

Therefore, we analyzed the state-of-the-art power analysis attacks and selected the one proposed by Sim et al. (2019, [46]) for our study. It is a single trace attack, a characteristic that turns it into a threat even for key ephemeral scenarios. We then simulated the attack and designed a countermeasure to secure the LEDAcrypt primitives. Finally, we carried out the attack against a physical device and validated our countermeasure.

In conclusion, on one hand we validated the single trace attack of Sim et al., while explaining the use of a Hamming weight leakage model. On the other hand, we designed a countermeasure based on *random precharging* able to secure the algorithm under threat.

Contents

List of Figures

List of Tables

List of Algorithms

Introduction

Since human beings started organizing in communities ages ago, the need of communication has been at the core of human life. Soon another need laid the foundation to what we now know as *cryptography*: the need to communicate selectively.

The word cryptography comes from ancient Greek: *kryptó s* meaning hidden and *graphein* meaning to write. In other words, the art of coding the messages in such a way that only some designed people could have access to the original information.

Among the most famous ciphers in history there are the *scytale*, a *transposition cipher* used by Spartans to communicate during military campaigns. A transposition cipher is an encryption technique by which the position of the letters in the original message is changed accordingly to a reversible scheme producing a permutation of the original message. Another famous example of ciphers in history is the *Caesar cipher*, a form of *substitution cipher* used by Julius Caesar. A substitution cipher encrypts the message by replacing the letters with other symbols (from the same or different alphabet) accordingly to a fixed scheme. In this case, the order of the letter does not change (as in the case of transposition ciphers) but the symbols do. Together with cryptography, another important discipline has been developed which is *cryptanalysis*, the art of breaking cipher to recover the secret message (or key). For example the *frequency analysis*, which studies the frequency of letters or groups of letters in a protected message, can retrieve a message obfuscated by means of transposition and substitution which do not modify the frequency of the symbols.

What we now refer to as *modern cryptography* has its bases in the *Kerckhoffs's principle* (end of 19^{th} century) which states: A cryptosystem should

be secure even if everything about the system, except the key, is public knowledge. In the 1940s C. E. Shannon proposed a more mathematical approach to the study of cryptography in communication theory opening the way for modern cryptography. In his paper "Communication theory of secrecy systems" published in 1949 [43], defines the concept of *perfect secrecy* for secret-key systems and proves their existence formulating the following theorem:
Let

$$\langle \mathcal{A}, \mathcal{M}, \mathcal{K}, \mathcal{C}, \{Enc_k(), k \in \mathcal{K}\}, \{Dec_k(), k \in \mathcal{K}\}\rangle$$

denote a symmetric key cryptosystem where the keys are picked independently of plaintexts values and $|\mathcal{K}| = |\mathcal{C}| = |\mathcal{M}|$.
The cryptosystem is perfectly secure *if and only if*

i. every key is used with probability $\frac{1}{|\mathcal{K}|}$

ii. $\forall (m, c) \in \mathcal{M} \times \mathcal{C}$ there is a *unique* $k \in \mathcal{K}$ such that $Enc_k(m) = c$.

Plaintext is the name assigned to the original message before encryption takes place. A plaintext belongs by definition to the message space $\mathcal{M}$. When the message has been manipulated by the cipher and has become incomprehensible, it takes the name of *ciphertext* and it belongs to the ciphertext space $\mathcal{C}$. The symbol $\mathcal{A}$ denotes the alphabet, that is the set of symbols which can form a message. The key k is the parameter taken by the cryptographic algorithm to manipulate the plaintext producing the ciphertext. The key is usually produced by the cryptographic algorithm by means of a *key-generation algorithm*, `Gen()` $= k$, and it belongs to the key space $\mathcal{K}$. Finally, $\mathbb{E}_k()$ and $\mathbb{D}_k()$ denotes respectively the encryption algorithm and the decryption algorithm such that:

$$Enc_{k_1}(m) = c, \quad Dec_{k_2}(c) = m, \qquad k_1, k_2 \in \mathcal{K},\ m \in \mathcal{M},\ c \in \mathcal{C}$$

Another important Shannon's contribution to the modern cryptography is the formulation of two general design principles:

Principle (Confusion)**.** Make the relation between the key, plaintext (ptx) and ciphertext (ctx) as complex as possible. Ideally, each digit of the key influences the correspondence between ptx and ctx letters in a non-predictable way

Principle (Diffusion)**.** Refers to the property that the statistical distribution of groups of ptx letters frequencies (due to the redundancy of the ptx language) should be dissipated, as much as possible, into flat distribution statistics, i.e. the ctx should appear as random data.
Ideally, keeping the same key, the change of a single bit in the plaintext drives the change of all bits in ciphertext.

Cryptosystems are divided into two classes on the basis of how the key is involved in the cipher.

Symmetric-key cryptosystems

Symmetric-key cryptosystems, also known as private-key cryptosystems, are characterized by the usage of a single key for both the encryption and decryption algorithm, that is, the sender and the recipient of the message shares the same secret key.

There are two kinds of symmetric-key cryptosystems:

- **block ciphers**, they are ciphers which operate on a block of plaintext at a time, producing a block of ciphertext by means of a key-parametric transformation. AES (Advance Encryption Standard) is an example of block cipher.

- **stream cipher**, they are ciphers which operate on individual plaintext bits or digits. A well known example of stream cipher (although nowadays its use is deprecated) is RC4 (Rivest Cipher 4).

Asymmetric-key cryptosystems

The major problem of symmetric-key cryptosystems is the key exchange. The two parties need to find a secure channel on which they can share the secret key.

In 1976, W. Diffie and M. Hellman proposed a solution to overcome this issue, presenting the concept of asymmetric-key cryptography (also known as public-key cryptography) [19], revolutionizing the modern cryptography.

This scryptographic scheme employs a pair of keys (K_{priv}, K_{pub}) consisting of *private key*, which must be kept secret and never shared, and a *public key*, which is distributed to the other parties by means of a public channel.

The public key K_{pub} of the recipient is used by the sender to encrypt the plaintext, in this way, the recipient will decrypt the ciphertext using his own private key K_{priv}.

Encryption and decryption functions for public-key cryptosystems (respectively, $Enc_{k_{pub}}()$ and $Dec_{K_{priv}}()$) are designed in practice making use of number theoretic problems. The most common ones are:

- **Integer Factorization Problem**: given a composite integer n, compute its factorization $\prod_i p_i^{e_i}$, $e_i \geq 1$.
- **Discrete Logarithm Problem**: given a cyclic group $(\langle g \rangle, \cdot)$ and $g_1 = g^x$, find $x \in \{0, 1, \ldots, |g| - 1\}$.

These problems allows both the public-key encryption and decryption functions to be designed in such a way that

i. the public key and the private key are linked in a mathematical way

ii. the knowledge of the public key tells you nothing about the private key

iii. the knowledge of the private key allows you to decrypt messages encrypted with the correspondent public key.

The encryption functions are also called *one-way trapdoor functions* which are functions easy to compute in one direction (i.e., computing the public key knowing the private key) and computationally hard to invert (i.e, computing the private key, knowing the public key).

A well known cryptographic algorithm based on the integer factorization problem is the RSA algorithm (Rivest, Shamir and Adleman). While important results based on the discrete logarithm problem are the public-key agreement protocol known as Diffie-Hellman key exchange, the Elgamal encryption algorithm and the ElGamal signature algorithm, from which has been derived the more widely used DSA (Digital Signature Algorithm).

Quantum Computing and Post-Quantum Cryptography

In 1982, Feynman was the first suggesting that the computational power of quantum mechanical processes might overcome that of the classical computation models [21]. He also reasoned about the possibility for traditional

computers of efficiently simulate these quantum processes, eventually giving a negative answer: a "quantum computing" might be imagined that could perform such simulations efficiently. Few years later, Deutsch [18] proposed a theoretically physically realizable model for the "quantum computer", which he speculated might be more efficient than a deterministic Turing Machine for certain types of computations.

An important result in cryptography, related to quantum computing, was achieved in 1994 by Shor [44] who proposed an algorithm able to solve the discrete logarithm and the integer factorization problems in polynomial time, given a sufficiently large-scale quantum computer. This result drawn the attention of the community since the most widely used public-key cryptographic algorithms were (and still are) based on those problems, raising the need for post-quantum cryptography.

Code-based cryptosystems are the most investigated ones to ensure a new standard for cryptographic algorithms which must be secure against both quantum and classical computers. Basic concepts of coding theory as well as code-based cryptography are given in section 1.1, while section 2.1 presents, in chronological order, the main results in post-quantum cryptography and cryptanalysis.

Chapter 1

Theoretical Background

This chapter introduces theoretical principles which will be recalled during this document. First it is presented a list of basic concepts about code-based cryptography, as well as the formulation of the two code-based cryptosystems at the basis of the modern post quantum cryptography. Later, it will be proposed a theoretical introduction to side channel attacks, further deepening the techniques based on the exploitation of power consumption leakages and corresponding countermeasures.

1.1 Code-based cryptography

Binary error correcting codes rely on a redundant representation of information in the form of binary strings used for controlling errors in data over unreliable or noisy communication channels.

Let $\mathbb{F}_2$ denote the binary finite field with the addition and multiplication operations corresponding respectively to the exclusive-or and logical product between two Boolean values. Let $\mathbb{F}_2^k$ denote the k-dimensional vector space defined on $\mathbb{F}_2$. A binary code, denoted as $C(n,k)$, is defined as a bijiective map $C(n,k) : \mathbb{F}_2^k \to \mathbb{F}_2^n$, $n,k \in \mathbb{N}, 0 < k < n$, between any binary k-tuple (i.e., an information word) and a binary n-tuple (denoted as codeword). The value n is known as the length of the code, while k is denoted as its dimension.

Encoding through $C(n,k)$ means converting an information word $u \in \mathbb{F}_2^k$ into its corresponding codeword $c \in \mathbb{F}_2^n$. The decoding process, instead, given a codeword $\hat{c}$ corrupted by an error vector $e \in \mathbb{F}_2^n$ with Hamming

weight $t > 0$ ($\hat{c} = c + e$), recovers both the value of the information word u and the value of the error vector e. A code is said to be t-error correcting if, for any value of e, given $\tilde{c}$ there is a decoding procedure to retrieve both the error vector e and the original information word u.

Code-based cryptography is based on the decoding problem of random error-correcting codes which is known to be NP-hard [5]. The problem consists of finding the closest codeword c to a given $\hat{c} \in \mathbb{F}_2^n$, assuming that there is a unique closest codeword.

Definition 1.1 (Circulant Matrix)**.** A $r \times r$ matrix $H \in \mathbb{F}_2^{r \times r}$ is a *circulant matrix* if its rows are successive cyclic shifts of its first one. From the definition, it follows that any circulant matrix has a constant row (and column) weight[1]. The top row (or the leftmost column) of a circulant matrix is the generator of the circulant matrix.

Definition 1.2 (Quasi-Cyclic Matrix)**.** A matrix $H = (H_0|\dots|H_{n_0-1})$ is a *quasi-cyclic* (QC) matrix if the n_0 sub-matrices $H_0, \dots, H_{n_0-1}$ are circulant matrices.

Definition 1.3 (Linear Code)**.** The binary code $C(n, k)$ of length n and dimension k over a field $\mathbb{F}_2$ is linear if and only if the set of its 2^k code-words is a k-dimensional subspace of the vector space $\mathbb{F}_2^n$.

Definition 1.4 (Minimum Distance)**.** Given a linear binary code $C(n, k)$, the *minimum distance* $d(C)$ of $C(n, k)$ is the minimum Hamming distance among all the ones which can be computed between a pair of its codewords. The minimum distance gives the smallest number of errors needed to change one codeword into another. As a result, it defines the error correction capability of the linear code C. In coding theory, depending on the adopted code, it is possible to correct up to $\lfloor (d(C) + 1)/2 \rfloor$ errors over a codeword $c \in \mathbb{F}_2^n$. Namely, linear code C can correct up to t errors if $d(C) \geq 2t + 1$.

Definition 1.5 (Quasi-Cyclic Code)**.** A QC-code is defined as a linear block code $C(n, k)$ having information word size $k = rk_0$ and codeword size $n = rn_0$, where n_0 is denoted as basic block length of the code and each cyclic shift of a codeword by n_0 symbols results in another valid codeword.

[1] The number of bits set to 1.

There is a ring isomorphism denoted as φ between the $r \times r$ circulant matrices and the quotient polynomial ring $R = \mathbb{F}_q[x]/\langle x^r - 1\rangle$. Thus, a circulant matrix A whose first row is $\mathbf{a_0} = [a_{0,0}, \ldots, a_{0,r-1}]$ is mapped to the polynomial $\varphi(A) = a_{0,0} + a_1 x + \ldots + a_{0,r-1}x_{0,r-1}$, and the (n, k)-QC code can be viewed as a cyclic code over the ring $R = \mathbb{F}_q[x]/\langle x^r - 1\rangle$.

Definition 1.6 (Generator Matrix)**.** The code C can be specified by providing a generator matrix $G \in \mathbb{F}_2^{k\times n}$, i.e., a matrix whose rows form a basis of C.

Definition 1.7 (Parity-Check Matrix)**.** A *parity-check matrix* $H \in \mathbb{F}^{r\times n}$ is a matrix which characterizes the linear code as $C = \{c \in \mathbb{F}_2^n | cH^T = 0_r\}$.

Definition 1.8 (Syndrome)**.** We refer to $s = Hx^T$ as *syndrome* of x. A vector x from $\mathbb{F}_2^n$ is contained in C (i.e., is a codeword of C) if and only if its *syndrome* is 0^r.

Definition 1.9 (QC-MDPC/LDPC)**.** C is a QC-MDPC code if each row of the parity-check matrix H which defines the code C has the same density $w = O(\sqrt{n\log(n)})$ (as defined in [46]). In case of a QC-LDPC code, each row of the parity check will have smaller constant row weights, usually less than 10.

Without knowing the QC-MDPC parity check matrix $H = (H_0|H_1)$, decoding a corrupted codeword (i.e., removing its errors) from a random binary linear code is an NP-hard problem (as already mentioned above). However, if H is known and the Hamming weight of e is not too large, there are efficient algorithms for decoding corrupted QC-MDPC codewords. The most commonly used decoding algorithm is the *probabilistic bit-flipping* algorithm introduced by Gallager in [22].
Given a corrupted codeword with at most t errors, the algorithm will output the (nearest) codeword after a sequence of iterations. Each iteration decides statically which of the n positions of the input vector v might have a higher chance to be in error and flips the bits at those positions. The flipped vector then becomes the input vector to the next iteration. The basic version of the algorithm stops when the *syndrome* (computed as, $H \times v^T$) becomes zero. Based on the count of unsatisfied parity-check equations[2] (denoted as upc),

[2]Each equation given by $H_i \times v^T = s_i$ is denoted as parity-check equation. A parity-check equation is unsatisfied when $s_i = 1$.

the algorithm selects the positions that are the most likely of being in error. The higher the count is, the higher the probability of a position being in error.

Let $\mathbf{u}$ be the vector storing the count of upc for each position j of v^T. Then, the count of ups for position j is expressed as follows

$$u_j = |\{i|H_{i,j} = H_i \times v^T = 1\}|.$$

Evaluating the vector $\mathbf{u}$, there are two possibilities to determine which bits should be flipped:

- Flip all positions that violate at least $max(\{u_i\}) - \delta$ parity checks, where δ is a small integer, say 5.
- Flip all positions that violate at least T_i parity checks, where T_i is a precomputed threshold for iteration i.

1.1.1 McEliece cryptosystem

The McEliece cryptosystem is a PKC scheme proposed in 1978 by Robert McEliece [30] and exploiting the hardness of the problem of decoding a random-like linear block code. Key-generation, encryption and decryption operations can be generalized as follows:

- The *key-generation* algorithm considers a binary linear block code $C(n, k)$, with codeword length n, information word length k and outputs a secret key sk defined as the generator matrix $G \in \mathbb{F}_2^{k\times n}$ of a code $C(n, k)$ able to correct $t \geq 1$ or less bit errors, plus a randomly chosen invertible binary matrix $S \in \mathbb{F}_2^{k\times k}$, named scrambling matrix, and a binary permutation matrix $P \in \mathbb{F}_2^{n\times n}$:

$$s_k \leftarrow \{S, G, P\}. \tag{1.1}$$

  The corresponding public key pk is computed as the generator matrix $G' \in \mathbb{F}_2^{k\times n}$ of a permutation-equivalent code with the same size and correction capability of the original code:

$$pk \leftarrow \{G'\}, with G' = SGP \tag{1.2}$$

- The *encryption* algorithm takes as input a public key pk and a message vector $m \in \mathbb{F}_2^k$, and outputs a ciphertext c computed as:

$$c = (m \cdot G' \oplus e) \in \mathbb{F}_2^n, \tag{1.3}$$

where $e \in \mathbb{F}_2^n$ is a random binary vector with weight t, named error vector.

- The *decryption* algorithm takes as input a secret key sk and a ciphertext c and outputs a message m' computed as the result of a known error correcting decoding algorithm able to remove t errors present in cP^{-1} and subsequently multiplying by the inverse of the matrix S:

$$\begin{aligned} C' = Decode(cP^{-1})S^{-1} = Decode((cP^{-1})G + eP^{-1})S^{-1} \\ = (mS)S^{-1} = m \end{aligned} \tag{1.4}$$

1.1.2 Niederreiter cryptosystem

The Niederreiter cryptosystem [34] is a code-based cryptosystem exploiting the same trapdoor introduced in the McEliece [30] with an alternative formulation. The encryption employs syndromes and parity-check matrices in place of the codewords and generator matrices employed by the encryption algorithm in the McEliece. When the same family of codes is used, Niederreiter and McEliece cryptosystems exhibit the same cryptographic guaranties [53]. Key-generation, encryption and decryption defining Niederreiter cryptosystem are as follows:

- The *key-generation* algorithm considers a binary linear block code $C(n, k)$, with codeword length n, information word length k and outputs a secret key sk defined as the parity-check matrix $H \in \mathbb{F}_2^{r \times n}$ of a code $C(n, k)$, $r = n - k$ able to correct $t \geq 1$ or more bit errors, plus a randomly chosen invertible matrix $S \in \mathbb{F}_2^{r \times r}$, named scrambling matrix:

$$sk \leftarrow \{H, S\}. \tag{1.5}$$

The corresponding public key pk is computed as the parity-check ma-

trix $H' \in \mathbb{F}_2^{r \times n}$ obtained as the product of the two secret matrices:

$$pk \leftarrow \{H'\}, with H' = SH \tag{1.6}$$

- The *encryption* algorithm takes as input a public key pk and a message binary vector $e \in \mathbb{F}_2^n$ with exactly t asserted bits, and outputs a ciphertext c computed as the syndrome of the original message:

$$c = H'e^T = SHe^T \tag{1.7}$$

- The *decryption* algorithm takes as input a secret key sk and a ciphertext c and outputs a message e computed as the result of a known error correction syndrome decoding algorithm applied the vector $S^{-1}c$ and able to recover the original error vector e:

$$e = SynDecoding(S^{-1}c) = SynDecoding(He^T) \tag{1.8}$$

1.2 Side Channel Attacks

A cryptographic primitive can be looked at in two different ways: it can be seen from an abstract mathematical point of view (as a mathematical relation transforming some input in some output); or it can be considered from a physical point of view, since eventually, this primitive will be implemented in a program and it will run on a specific hardware.

Several techniques have been studied for testing cryptographic algorithms in isolation in order to tackle the mathematical structure of the primitive. For example, linear cryptanalysis and differential cryptanalysis. These approaches have the advantage of being more general, since they do not relay on a given implementation, but still, physical attacks on cryptographic devices are often much more effective than classical cryptanalysis.

Such attacks exploit the physically observable environmental parameters of a device performing a computation known as side channels. Examples of side channels which can be related to the ongoing computation are: power consumption, electromagnetic (EM) emissions, computation time and erroneous output obtained from fault inducing factors.

Side channel attacks are techniques designed to force cryptosystem exploiting the information leaking from the side channels. Among the above mentioned techniques, the most important results have been achieved in power consumption analysis and EM emissions analysis.

1.2.1 Modeling Power Consumption

At the basis of the digital circuits there are logic cells. These can be divided into two categories: *combinatorial cells*, that is, logic that does not have memory and thus its output depends only on the input at the present clock cycle; and *sequential cells*, whose output depends not only on the present value of its inputs but on the input history as well. The former are used in computer circuits to perform Boolean algebra (such as `end`, `xor` ...), the latter are used to store data (for example a D-latch).

The most common *logic style* (that is the way logic levels 0 and 1 are physically represented by the logic cells) is the Voltage-Mode Logic style, in which the zeros and ones are represented in terms of a voltage level. The ground voltage level GND is associated to 0 and the reference voltage level V_{DD} is associated to 1. Internally, the cells of digital circuits are realized utilizing *transistors* as voltage driven switches. The most common transistor technology is the Complementary Metal-Oxide-Semiconductor (CMOS), which employs pairs of P-type and N-type metal-oxide-semiconductor field effect transistors (MOSFET) to implement logic functions.

The power absorbed by a cell can be split into two components: a static component P_{st} dissipated in a steady state (not directly influenced by the switching activity), and a dynamic component P_{dyn} that is absorbed when the cell is switching its logic state. In other words, if a cell is not switching its input state, the only contribution to the global power consumption will be the one of P_{st}, vice versa, if a cell changes logic level given in input, the cell will draw $P_{st} + P_{dyn}$. It follows that a reasonable model for the power consumption of a group of cells representing binary values is given by the *Hamming distance*[3] between the two outputs of the cells at time $t-1$ and t.

In case the cells are toggling from an all-zero or all-one pre-charged state, a better fitting model for the power consumption is given by the *Hamming*

[3]The Hamming distance between two binary values is defined as the minimum number of single bit flips which should be applied to the first value in order to obtain the second.

weight[4] (HW) of the value being computed.

1.2.2 Power Analysis Attacks

Power analysis attacks are built on the information leaked by a device through the *power consumption* correlated to the execution of a cryptographic algorithm. The key idea is to build a relation between the power consumption measured from the device and either a key-dependent control flow in the algorithm, or specific operations in the data flow.

Power Consumption Measurement

First, to mount an attack, the attacker needs to gather samples of the actual power consumption of the device while running a cryptographic algorithm. The power consumption can be either acquired from a real measure session by means of a digital sampling oscilloscope or obtained from a simulation by a power estimation tool such as the ones in common EDA tools.

In both cases, the power consumption associated to an execution i is stored as a *power trace* t_i, which is considered as a vector of M *power samples*, as shown in Equation 1.9.

$$\mathbf{t_i} = [t_{i,1}, t_{i,2}, \cdots, t_{i,M}] \tag{1.9}$$

Where each power sample $t_{i,j}$ of a power trace $\mathbf{t_i}$ is the sum of different contributions that can be identified in

$$t_{i,j} = t_{i,j}^{Op} + t_{i,j}^{Data} + t_{i,j}^{Noise} + t_{i,j}^{Stat} \tag{1.10}$$

where the first contribution $t_{i,j}^{Op}$ is the power consumption due to the specific operation executed, the second $t_{i,j}^{Data}$ is the power consumption due to the processed data values, the third $t_{i,j}^{Noise}$ is the due to the environmental noise and the last $t_{i,j}^{Stat}$ is the static power consumption of the device.

The $t_{i,j}^{Stat}$ does not depend on the device activities, thus it is irrelevant to the purposes of power analysis. Differently, the $t_{i,j}^{Noise}$ contribution should be minimized as much as possible in order to achieve good performance in the results. Since $t_{i,j}^{Noise}$ is not affected by the ongoing operations, it can

[4]The Hamming weight of a binary value is defined as the number of bits set to 1 in the value.

be modeled as a random variable following a normal distribution with zero mean $\mathcal{N}(0, \sigma)$.

The other three contributions, given a fixed input and a fixed implementation platform, are constant. This means that $t_{i,j}$ will follow a normal distribution $\mathcal{N}(\mu_{i,j}, \sigma)$, and it is thus possible to reduce the noise through averaging a reasonable amount of measurements of the same encryption.

Attacks principles

Power analysis attacks focus on exploiting meaningful dependencies between the power consumption of the device and the cryptographic algorithm flow, that will turn eventually into a key recovery.

There are two kinds of dependencies that can be used in a power analysis attack. The first dependency is associated to the *control flow* of the algorithm, that is, different operations are executed for different key bit value. The second dependency is related to the *data flow* of the algorithm, in this case the difference is not about the operations, but it interests the data depending on the key bit value. These dependencies are exploited differently resulting in two different families of attacks which will be presented later in the subsection.

Another characteristic which is used to classify the attacks is the number of traces that is necessary to analyze for the attack to succeed. The so called *vertical attack* exploits a dependency on the key bit once per trace and needs many traces (typically hundreds or thousands) to get enough information in order to recover the secret key. The *horizontal attack*, contrariwise, exploits a single power information repeated many times along the same trace and thus it needs only few traces (sometimes just one trace) to mount the attack. The two approaches are illustrated in Figure 1.1.

Simple Power Analysis (SPA)

The most straightforward attack technique relying on the power consumption of a device is the Simple Power Analysis (SPA). This technique leverages the specific key-dependent points in the control flow of a cryptographic algorithm, for which the measurement of the dynamic power con-

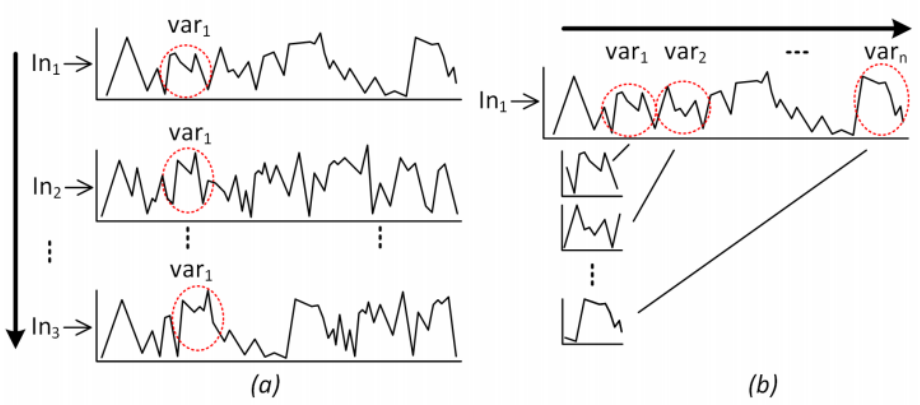


Figure 1.1: Vertical DPA (a) targets a single intermediate computation and seeks correlation across multiple traces each using a distinct input, while Horizontal DPA (b) targets multiple intermediate computations within a trace and seeks a correlation among them [2].

sumption of the circuit can leak the key. In other words, SPA exploits points in the algorithm where an instruction is executed depending directly on a specific value of the secret key, as it happens in key-dependent branches which can be found, for example, in the schoolbook implementation of the straightforward square and multiply (or double and add) exponentiation (multiplication) algorithm. In cases where the squaring operation has a different power consumption with respect to the multiplication, it is possible for an attacker to distinguish the two operations simply looking at the recorded power trace of an exponentiation computation. Since the operations are strictly key-dependent, this would be enough to recover the secret key.

Figure 1.2 illustrates a practical power trace example that clearly exposes the vulnerability. It is evident the difference in the power consumption that is recorded when the multiplication is performed rather than the squaring operation.

It is thus possible for an attacker to recover, in a straightforward way, each single bit composing the secret key, by simply decoding the information captured in the recorded power trace.

Since this attack requires a single power trace in order to be accomplished, it can be considered belonging to the horizontal class.

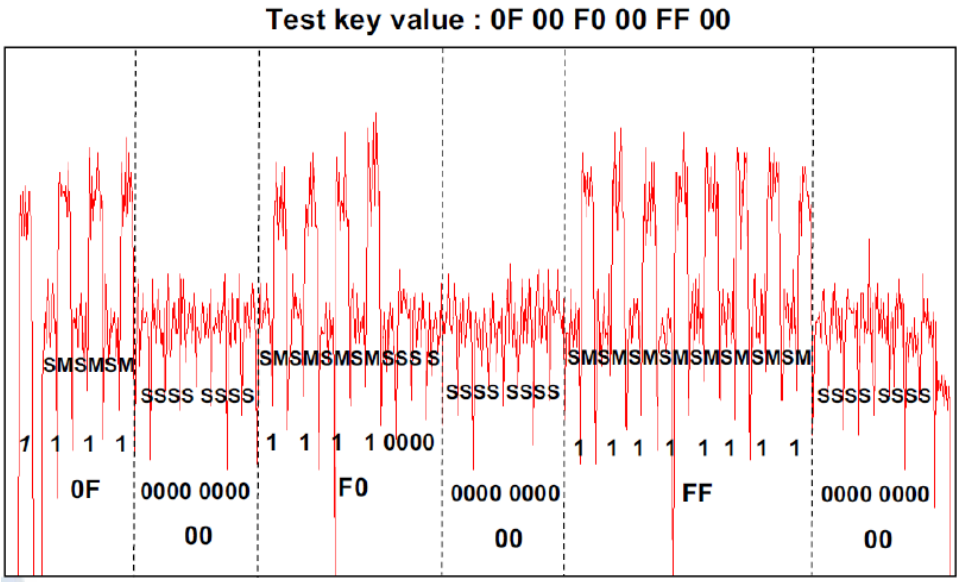


Figure 1.2: Simple Power Analysis against an RSA Left-To-Right Square and multiply.

Differential Power Analysis (DPA)

Differential power analysis is a statistical power analysis technique, first introduced by P. Kocher et al. [27], that relies on the *difference of means* (DOM) statistical test to recover the secret key from a device.

The essential difference between SPA and DPA attacks is that, the former take advantage of the difference in power consumption related to different key-dependent *operations* executing on the device whereas DPA attacks exploit the difference in power consumption as a result of using key-dependent *data*.

The basic idea of DPA is to predict the portion of the power consumption which depends on the key, for a small amount of key values, and distinguish the correct prediction employing the actual measurements as reference. A statistical test is applied in order to identify the dependencies between the measurements and the predictions: once the correct prediction is detected, the value for a portion of the secret key is retrieved. Depending on the specific statistical test employed to verify the hypothesis, DPA attacks are referred to by different denomination in the literature.

In the practice, DPA attacks are organized in five steps:

i. In the first step, an intermediate value of the cryptographic algorithm which is computed as a known function $f(d, k)$ is identified. d is a known non-constant data value (usually either the plaintext or the ciphertext) and k is a small portion of the secret key (typically $6 - 8$ bits in order to keep the hypothesis space $\mathcal{K}$ small).

ii. Subsequently, N power traces $\mathbf{t_i}$ are collected using N different known data d_i over a fixed key $\hat{k}$. The measurements are gathered in the form of a matrix of power traces $\mathbf{T}$ of size $N \times M$, where each power trace represents a row vector, while the input data are stored in the form of a vector $\mathbf{d}$

$$\mathbf{T} = \begin{pmatrix} \mathbf{t_1} \\ \mathbf{t_2} \\ \vdots \\ \mathbf{t_i} \\ \vdots \\ \mathbf{t_N} \end{pmatrix} = \begin{pmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,j} & \cdots & t_{1,M} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,j} & \cdots & t_{2,M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ t_{i,1} & t_{i,2} & \cdots & t_{i,j} & \cdots & t_{i,M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ t_{N,1} & t_{N,2} & \cdots & t_{N,j} & \cdots & t_{N,M} \end{pmatrix} \quad \mathbf{d} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ d_i \\ \vdots \\ d_N \end{pmatrix}$$

iii. In the third step, the hypothetical intermediate values $v_{i,l}$ for each input data d_i and every possible guess $k_j \in \{k_1, k_2, \ldots, k_l, \ldots, k_\mathcal{K}\}$ such that $v_{i,l} = f(d_i, k_l)$, are computed. These values are gathered in a matrix $\mathbf{V}$ of size $N \times |\mathcal{K}|$ which then contains every intermediate value for each possible key guess k_l

$$\mathbf{V} = \begin{pmatrix} v_{1,1} = f(d_1, k_1) & \cdots & v_{1,l} = f(d_1, k_l) & \cdots & v_{1,|\mathcal{K}|} = f(d_1, k_{|\mathcal{K}|}) \\ v_{2,1} = f(d_2, k_1) & \cdots & v_{2,l} = f(d_2, k_l) & \cdots & v_{2,|\mathcal{K}|} = f(d_2, k_{|\mathcal{K}|}) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ v_{i,1} = f(d_i, k_1) & \cdots & v_{i,l} = f(d_i, k_l) & \cdots & v_{i,|\mathcal{K}|} = f(d_i, k_{|\mathcal{K}|}) \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ v_{N,1} = f(d_N, k_1) & \cdots & v_{N,l} = f(d_N, k_l) & \cdots & v_{N,|\mathcal{K}|} = f(d_N, k_{|\mathcal{K}|}) \end{pmatrix}$$

In this representation, each column l contains the intermediate values that have been computed under the hypothesis of a specific key guess k_i and the plaintext d_i. At this point, the idea of DPA is to determine which column actually contains the intermediate values corresponding

to the ones computed by the device. In this way, the portion k_l of the secret key is retrieved.

iv. In the fourth step, a power model f_s (chosen accordingly to the cipher implementation) is applied to map the hypothetical intermediate values contained in matrix $\mathbf{V}$ to a matrix $\mathbf{P}$ of hypothetical power consumption values

$$\mathbf{P} = \begin{pmatrix} p_{1,1} = f(d_1, k_1) \cdots p_{1,l} = f(d_1, k_l) \cdots p_{1,|\mathcal{K}|} = f(d_1, k_{|\mathcal{K}|}) \\ p_{2,1} = f(d_2, k_1) \cdots p_{2,l} = f(d_2, k_l) \cdots p_{2,|\mathcal{K}|} = f(d_2, k_{|\mathcal{K}|}) \\ \vdots \quad \ddots \quad \vdots \quad \ddots \quad \vdots \\ p_{i,1} = f(d_i, k_1) \cdots p_{i,l} = f(d_i, k_l) \cdots p_{i,|\mathcal{K}|} = f(d_i, k_{|\mathcal{K}|}) \\ \vdots \quad \ddots \quad \vdots \quad \ddots \quad \vdots \\ p_{N,1} = f(d_N, k_1) \cdots p_{N,l} = f(d_N, k_l) \cdots p_{N,|\mathcal{K}|} = f(d_N, k_{|\mathcal{K}|}) \end{pmatrix}$$

v. In the last step, a statistical test is employed to compare each predicted power consumption (given a fixed key value k_l) against the actual ones stored in the power traces. When a predicted power consumption matches, with a high statistical confidence, the actual consumption of the device for a specific time instant, it means that the guessed key value is the correct one.

In the following paragraphs, the two most common statistical test employed in DPA will be presented.

Difference of Means - Common DPA attack. The difference of means is the first proposed statistical test used to validate the power consumption prediction. This tool is applied on data classified into two different sets S_0, S_1, in such a way that the sample-wise mean consumption of the two sets presents a significant difference for some time instant j.

The attacker first chooses a *selection function* f_s to decide to which set, $S_{0,l}$ or $S_{1,l}$, a trace $\mathbf{t_i}$ belongs, depending on the predicted power consumption $p_{i,l} = f_s(v_i, l) = f_s(f(d_i, k_l))$ already computed with the data input d_i related to the trace $\mathbf{t_i}$ and the key hypothesis k_l. This selection operation is repeated for each key guess k_l , since they will produce different partitions of the traces.

The simple-wise mean m_{0_l} of all the traces belonging to S_{0_l}, and the sample-wise mean m_{1_l} of all the ones belonging to S_{1_l} are calculated.

The attacker computes the sample-wise difference $\delta_l = m_{0_l} - m_{1_l}$ for all possible key hypothesis k_l. The correct key hypothesis will produce a significantly large value of δ_l for some time instant i. This is due to the fact that, if the key is correct, the selection function performs a correct partitioning of the traces into two sets where the mean consumption of the operation fits the predictions. Vice versa, if the key hypothesis is wrong, the selection function performs a random partitioning, resulting in two sets with roughly the same mean consumption.

Pearson's Linear Correlation Coefficient - (CPA). Correlation power analysis (CPA) employs the Pearson's (linear) correlation coefficient as a statistical test to detect the correct key hypothesis. Pearson's linear correlation coefficient is the measure of the linear correlation between two random variables X and Y.

The Pearson's linear correlation coefficient assumes values between -1 and 1. Values of the correlation coefficient close to 1 (or -1) indicates a high positive (or negative) linear correlation, whereas values close to zero means that the two variables are not linearly correlated.

Pearson's linear correlation coefficient between two random variables X and Y (denoted as $\rho_{X,Y}$) is defined as

$$\rho_{X,Y} = \frac{\text{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X - \mu_X)(Y - \mu_Y)]}{\sigma_X \sigma_Y} \tag{1.11}$$

Where, cov denotes the covariance between the two variables and σ_X denotes the variance of a random variable X.

In order to use the Pearson's correlation coefficient as a statistical test in a DPA attack, the actual power consumption measured in a precise time instant for all the traces $\mathbf{t_j} = [t_{1,j}, t_{2,j}, \cdots, t_{i,j}, \cdots, t_{N,j}]$ and its prediction for a fixed key hypothesis $\mathbf{p_j} = [p_{1,j}, p_{2,j}, \cdots, p_{i,j}, \cdots, p_{N,j}]$ are considered to be modeled by two random variables. Since the attacker does not know the theoretical distribution of these variables, he will need to employ the sample Pearson correlation coefficient as an estimator of the correct value of $\rho_{\mathbf{t_j},\mathbf{p_l}}$. Given the samples contained in $\mathbf{p_l}$ and $\mathbf{t_j}$, the sample Pearson correlation

coefficient (commonly noted as $r_{\mathbf{t_j},\mathbf{p_l}}$) can be computed as

$$r_{\mathbf{t_j},\mathbf{p_l}} = \frac{\sum_i (t_{i,j} - \bar{t}_j)(p_{i,l} - \bar{p}_l)}{\sqrt{\sum_i (t_{i,j} - \bar{t}_j)^2 \sum_i (p_{i,l} - \bar{p}_l)^2}} \tag{1.12}$$

where $\bar{t}_j$ and $\bar{p}_l$ are the sample means over $\mathbf{t_j}$ and $\mathbf{p_l}$ respectively.

Computing the coefficient for all the time instants j and the key hypothesis k_l exposes the correct key in correspondence of the highest peak correlation coefficient over the whole encryption.

Template Attacks

Template attacks are the strongest form of side-channel attack from an information point of view. They belongs to the class of profiled side-channel attacks and they have been firstly proposed by Chari et al. in 2002 [12].These attacks rely on the assumption that the attacker has access to a reference device identical to the target device that he can program at his will.

Template attacks are mounted in two stages. The first stage is the *profiling stage* (or *training* stage) and the second stage is the *extraction stage*. In the profiling stage, the attacker collects some power traces from an identical experimental device and builds templates for each key-dependent operation. During the extraction stage, the attacker exploits the templates obtained from the profiling stage to classify the correct key from a single power trace (or at most few power traces) collected from the target device.

This technique thus requires a large number of power traces to profile the device, but only few power traces (if not one) to retrieve the key during the extraction phase against the target device. This feature makes the template attack able to succeed in breaking implementations and countermeasures whose security is dependent on the assumption that the attacker can use only one or limited number of power traces, as in case of ephemeral keys cryptographic protocols.

Profiling phase - Template creation. A template is a set of probability distributions describing the power traces shape for many different keys. A

trace is considered the realization of a *multivariate Gaussian* random variable.

The most important points in each trace are called *points of interest* (PoI), this specific instants should contain most of the useful information and are usually a few, which allows to keep relatively small the dimension of the multivariate distribution. To identify the most significant PoIs, there are many techniques. One of the most effective method of choosing these points is named SOST.

The attacker runs the implementation of the target cryptographic algorithm n times obtaining n actual power traces $(\mathbf{s_1}, \mathbf{s_2}, \ldots, \mathbf{s_n})$ which will be needed to identify the interesting points. The n power traces are sampled from the implementation with a fixed (know) key and a (known) random input plaintext.

For a fixed instant point p_t, the power consumption values of the n power traces are collected in a vector $\mathbf{r}(t) := (\mathbf{s_1}[p_t], \mathbf{s_2}[p_t], \ldots, \mathbf{s_n}[p_t])$. Let's define the sets $\mathbf{G_i} := \{\mathbf{s_j} | g(m_j, key) = o_i\}$, $i = 0, 1, \ldots, \bar{k} - 1$, where $g(m_j, key)$ denotes a function (depending on the input plaintext m_j and the (sub)key key) which expresses the key-dependent operation o_i. For the point p_t, we let $\bar{m}_i[p_t] := \sum_{l=1}^{|\mathbf{G_i}|} \mathbf{s_l}[p_t] / |\mathbf{G_i}|$ $(i = 0, 1, \ldots, \bar{k} - 1)$, where $\mathbf{s_l} \in \mathbf{G_i}$.

Now, the SOST is based on the T-Test and is defined as the sum of squared pairwise t-differences. For a point p_t, let $\sigma_i^2(t)$ denote the variance of all the sample data $\mathbf{s_i}[p_t]$, where $\mathbf{s_i} \in \mathbf{G_i}$. Then, the signal-strength estimate of the SOST method is computed as follows:

$$f(t) = \sum_{i \neq j} \left(\frac{\bar{m}_i[t] - \bar{m}_j[t]}{\sqrt{\frac{\sigma_i^2(t)}{|\mathbf{G_i}|} + \frac{\sigma_j^2(t)}{|\mathbf{G_j}|}}} \right)^2 , \tag{1.13}$$

where $i, j \in \{0, 1, \ldots, \bar{k} - 1\}$.

Once the attacker has picked $\bar{i}$ points of interest, he computes the mean and the covariance matrix for every operation k (every choice of (sub)key). Let μ_i be the average power consumption at instant p_i; v_i denotes the variance of the power consumption at each point of interest p_i, and let $c_{i,j}$ be the covariance between the power consumption at every pair of points of interest (p_i and p_j).

For every operation $k \in \{0, 1, \ldots, \bar{k} - 1\}$ he will obtain the mean vector μ_k and the covariance matrix $\mathbf{\Sigma}_k$

$$\mu_k = \begin{bmatrix} \mu_{k,1} \\ \mu_{k,2} \\ \vdots \\ \mu_{k,i} \\ \vdots \\ \mu_{k,|I|} \end{bmatrix} \quad \mathbf{\Sigma}_k = \begin{bmatrix} v_{k,1} & c_{k,(1,2)} & \cdots & c_{k,(1,j)} & \cdots & c_{k,(1,|I|)} \\ c_{k,(2,1)} & v_{k,2} & \cdots & c_{k,(2,j)} & \cdots & c_{k,(2,|I|)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{k,(i,1)} & c_{k,(i,2)} & \cdots & c_{k,(i,j)} & \cdots & c_{k,(i,|I|)} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ c_{k,(|I|,1)} & c_{k,(|I|,2)} & \cdots & c_{k,(|I|,j)} & \cdots & v_{k,|I|} \end{bmatrix}$$

Extraction phase - Template matching. Assuming that the attacker obtains t actual power traces (labeled as $\mathbf{s}_1, \mathbf{s}_2, \ldots, \mathbf{s}_t$) from the target device in the extraction phase. When the power traces are statistically independent, the attacker can apply maximum likelihood approach on the product of conditional probabilities

$$key_{ck} := argmax_{key_i}\{\prod_{j=1}^{t} \Pr(\mathbf{s}_j | key_i),\ i = 0, 1, \ldots, \bar{k} - 1\},$$

where $\Pr(\mathbf{s}_j | key_i)$ gives a measure of how likely is it that key key_i is the correct one for trace j.

The key_{ck} reveals which guess for the (sub)key fits the templates the best and thus it is considered to be the correct (sub)key.

1.2.3 Countermeasures

Since side-channel attacks rely on the relationship between information leaked though a side channel during the computation of a cryptographic algorithm and the secret data, the countermeasures basic idea is to break those dependencies. Following this principle, countermeasures can be categorized in two classes that are called *hiding* and *masking*. The former tries to hide the information leaked without altering the computation, while the latter tries to mask the computation without altering the side-channels.

Countermeasures can be implemented at different levels of abstraction. At the lowest level, countermeasures can be implemented by using protected logic styles that try to hide the usual power consumption. At architectural

level the main idea is to reorganize the flow of the instructions by randomly modifying the order of execution or by randomly inserting dummy instructions in order to produce power traces that are not directly comparable. Finally, at algorithm level, the cryptographic algorithm can be modified in such a way that the information leaked is not any more correlated with the expected intermediate values of a standard computation [29].

Different countermeasures can also be combined together to achieve a higher level of security, typically by implementing hiding countermeasures after that masking countermeasures have been used. When implementing such countermeasures, it is important to keep in mind that they usually come at a price in terms of loss of speed, higher chip area or higher power consumption that make countermeasures hard to design in the practice.

Hiding.

The objective of hiding is to tamper with the dependencies between the computation and the power consumption by altering the latter. Usually it tries to produce a power consumption either constant or random.

The hiding countermeasure can alter either the time or the amplitude of the power consumption. In the first case, the operations of the cryptographic algorithm are executed in a different order or at a different time instant for each different execution, this can be accomplished by random insertion of dummy operations or by shuffling the order of some operations every execution. Whereas in the second case the power consumption of each operation is altered randomly, this can be achieved by introducing noise in the form of switching activity in the normal cryptographic computation altering the power consumption.

Masking.

Masking is the complementary countermeasure to hiding, its purpose is to invalidate the dependency of the computation from power consumption by randomizing the intermediate results calculated by the cryptographic algorithm.

Implementing a cryptographic algorithm with masking means taking every intermediate value v and masking it by an unknown random value m

(that is different for every execution) that is called *mask*. Every intermediate result must be masked all the time $v_m = v \odot m$ where $\odot$ is a known operation. The most common operations employed in masking countermeasure are the exclusive-or function, the modular addition or the modular multiplication.

Masking operation can be designed over many shares (typically two or three) to achieve a higher level of security, bearing in mind that the overhead rapidly increases with the number of the shares. These techniques are known as *higher-order masking* (second-order masking, third-order masking etc.).

Countermeasures based on masking have to ensure that the results of an operation that involves two masked intermediate values are masked as well. Thus, in the practice many different masks are adopted making it necessary to specify a *masking scheme* that describes the order in which the masks are employed.

Chapter 2

State of the Art

This chapter presents the state of the art in post quantum cryptography, in particular, code-based cryptosystems and side channel attacks against them. First, it will illustrate the related works, presenting the evolution of techniques in the area of code based cryptography in chronological order. Subsequently, it will describe QcBits, a state of the art code-based cryptosystem which employs a constant-time multiplication to secure the cipher against timing analysis. Finally, it will present three papers discussing power analysis side channel attacks against hardware and software implementation of code-based cryptosystems.

2.1 Post Quantum Cryptography

The security of the most commonly employed public key cryptosystems (PKCs) is based on the difficulty of number theory problems, such as the integer factorization problem or the discrete logarithm problem. In 1994, Shor [44] proposed an algorithm that can solve such problems in polynomial time using quantum computing.

In the past few years, quantum computing became a main topic, so that, in 2015, the National Security Agency (NSA) announced that it is planning to transition "in the not too distant future" to a new cipher suite that is resistant to quantum attacks. In December 2016, the National Institute of Standards and Technology (NIST) announced a call for proposals for post-quantum cryptography (PQC) standardization. Some of the most promising alternatives include cryptosystems based on lattices, error correcting codes,

hash functions, and multivariate quadratic equations. These mathematical problems are expected to remain intractable even for quantum computing. In the second-round of NIST competition, twenty-six candidates (over sixty-nine) have survived [36], and seven candidates are code-based cryptographic algorithms.

Code-based cryptography is based on coding theory, which aims to detect and correct errors on transmitted data through a noisy channel. The first code-based PKC was proposed by McEliece in 1978 [30]. Its security is based on the difficulty of the Syndrome Decoding (SD) problem and the Goppa Code Distinguishing (GCD) problem. The main drawback of the original McEliece cryptosystem is the large size of the public keys. For the 80-bit security level, the public key size of the McEliece cryptosystem requires about 500 Kbits. To address this problem, several variants of the McEliece cryptosystem have been proposed, by exploiting different efficient codes other than Goppa codes, for example, generalized Reed-Solomon (GRS), low-density parity-check (LDPC), and moderate-density parity-check (MDPC) [7, 9, 16, 17, 31, 34].

Using Quasi-cyclic MDPC (QC-MDPC) in the McEliece cryptosystem was first suggested by Misoczki et al. in 2012 [32]. For the 80-bit security level, the public key of QC-MDPC McEliece requires only 4801 bits. Some hardware implementations of this scheme followed in 2013 [26] and 2014 [51].

Bernstein et al. proposed a key encapsulation mechanism (KEM)/data encapsulation mechanism (DEM) called McBits [7], using the Niederreiter cryptosystems as the underlying scheme.

In 2016, Chou proposed a variant of the hybrid (KEM/DEM) Niederreiter encryption scheme called QcBits [16]. It operates in constant time and has very good speed results and small key sizes.

Another issue with the QC-MDPC cryptosystems is that they use a probabilistic decoder with a non-negligible decoding failure rate (DFR) depending on the security parameters. In the original proposal by Misoczki et al. [31], the DFR was around 10^{-7}. In [24], Guo et al., take advantage of the decryption failure to recover the secret key of Misoczki's original version in minutes. For QcBits [16], Chou claims a DFR of 10^{-8} for the 80-bit secure version.

Kocher first presented side-channel attacks (SCAs) [28], which enable

to recover cryptographic keys by analyzing side-channel leakages such as execution time, power consumption, electromagnetic emission, and photonic emission, when cryptographic algorithm are running on devices. These side-channel attacks include timing attack (TA), simple power analysis (SPA), differential power analysis (DPA), correlation power analysis (CPA), and profiling attack [29].

A SCA against the McEliece cryptosystem was first proposed by Strenzke et al. in 2008 [50]. Other TAs against McEliece have been followed in [10, 45, 47–49]. While various SPAs and DPAs against the McEliece cryptosystem can be found in [20, 25, 33, 40, 41, 52]. In [14] Chen et al. introduce the horizontal DPA attack on a lightweight FPGA implementation of QC-MDPC McEliece presented in [51]. Finally, fault injection attacks have been presented in [11, 48].

Chou suggested a CCA-secure constant-time implementation for QC-MDPC McEliece to mitigate TAs [16]. Rossi et al. [42] proved this countermeasure to be vulnerable to a DPA in private syndrome computation. The proposed attack, however, requires further solving linear equations to obtain the entire secret key. Sim et al. [46] proposed a novel attack against QcBits able to fully recover the secret key, improving the results of Rossi et al. [42]. They also provided a single-trace SCA able to recover the secret key even when using ephemeral keys or applying the DPA countermeasures suggested in [15, 42].

2.2 QcBits cryptosystem

One problem with QC-MDPC codes is that the most widely used decoding algorithm, when implemented naively, leaks information about secrets through timing. Even though decoding is only used for decryption, the same problem can also occur if the key-generation and encryption are not constant-time.

QcBits [16] is a fully constant-time implementation of a QC-MDPC-code-based encryption scheme. It follows the McBits [7] paper to use a variant of the hybrid KEM/DEM Niederreiter encryption scheme proposed in [39]. As a property of the KEM/DEM encryption scheme, the software is protected against adaptive chosen ciphertext attacks (aka it's CCA-secure), unlike the

plain McEliece or Niederreiter [34] encryption scheme. Other than being faster than most of the other implementations (also not constant-time implementations), another important result of QcBits is that, using a 2^{80}-security parameter set, the algorithm has a decryption failure rate lower than 10^{-8}.

QcBits uses (n, r, w)-QC-MDPC binary codes with $n = 2r$, where n, r and w denotes respectively the code length, the code dimension and the codeword density. The parity check matrix in its QC-MDPC form is then composed of two square sparse circulant matrices

$$H = (H_0|H_1) \in \mathbb{F}_2^{r \times n}. \tag{2.1}$$

The generator matrix in its systematic form is the $r \times n$ binary matrix

$$G = (I|P) \tag{2.2}$$

Where I is the $r \times r$ identity matrix and P is an $r \times r$ dense binary circulant matrix

$$P = (H_1^{-1} \cdot H_0)^T. \tag{2.3}$$

It easy to verify that $H \cdot G^T = 0$, so the rows of G form a basis for the codewords. An r-bit data vector x is encoded by multiplying it by G:

$$c = x \cdot G. \tag{2.4}$$

Let e be a n-bit error vector, and $\hat{c}$ the corrupted codeword

$$\hat{c} = c \oplus e = x \cdot G \oplus e. \tag{2.5}$$

The private key of QcBits is the QC-MDPC parity check matrix H_{priv}:

$$H_{priv} = (H_0|H_1) \tag{2.6}$$

where H_0, $H_1 \in \mathbb{F}_2^{r \times r}$ are randomly generated circulant matrix with weight $w/2$ in each row. The private key is sparse, so only the indices of the nonzero values of the first row are stored. Knowing the private key, one can use the bit-flipping decoding algorithm to recover a codeword which has been corrupted up to t errors.

The public key is computed directly from the private key as the dense circulant $r \times r$ matrix P:

$$P = (H_1^{-1} \cdot H_0)^T. \tag{2.7}$$

From P, anyone can derive the generator matrix in its systematic form G_{pub} and a parity-check matrix H_{pub}:

$$G_{pub} = (I|P) \tag{2.8}$$

$$H_{pub} = (I, P^{-T}). \tag{2.9}$$

In the following, it will be presented how the QcBits cryptosystem primitives work.

In QcBits, Niederreiter encryption is used to encrypt a random vector e of weight t, which is then fed into a key-derivation function to obtain the symmetric encryption and authentication key. The ciphertext is then the concatenation of the Niederreiter ciphertext, the symmetric ciphertext, and the authentication tag for the symmetric ciphertext.

By default, QcBits uses the following symmetric primitives:

- A hash function denoted *Hash*. QcBits uses Keccak with 512-bit outputs [38];

- A symmetric stream cipher denoted (*Senc,Sdec*). QcBits makes use of Salsa20 [8];

- An authentication function denoted (*Tag,Check*). QcBits makes use of Poly1305 [6];

The encryption of a message m using QcBits is shown in Algorithm 2.2.1.

We next describe the bit-flipping algorithm, which is used by the decryption algorithm. In QcBits, the bit-flipping algorithm performs a total of $j_{max} = 6$ iterations. It uses the precomputed thresholds $Thresh[0, \ldots, 5] = [29, 27, 25, 24, 23, 23]$ in each iteration to determine which bits should be

Algorithm 2.2.1: QcBits encryption

Input : Plaintext m Public matrix P
Output: Ciphertext $(c|d|g)$

$e \leftarrow \$$ // Drawing a random n-bit error vector with Hamming weight t
$key \leftarrow Hash(e)$
$c^T \leftarrow (I, P^{-T}) \cdot e^T \in \mathbb{F}_2^r$
$d \leftarrow Senc(key, m)$
$g \leftarrow Tag(key)$
return $(c|g|d)$

flipped. This process is shown in Algorithm 2.2.2.

Algorithm 2.2.2: Bit Flipping

Input : $H_{priv} \in \mathbb{F}_2^{r \times n}, x \in \mathbb{F}_2^n$
Output: Corrected codeword v

$v \leftarrow x$
$S \leftarrow H_{priv} \cdot v^T$ // Syndrome computation
for $j = 0$ **to** j_{max} **do**
    for $i = 0$ **to** $n - 1$ **do**
        $\sigma_i \leftarrow \langle S, h_i \rangle \in \mathbb{Z}$ // h_i denotes the i-th column of H
        if $\sigma_i \geq Thresh[j]$ **then**
            $v_i \leftarrow v_i \oplus 1$
    $S \leftarrow H_{priv} \cdot v^T$
return the codeword v

The decryption works in a similar way as encryption, Algorithm 2.2.3. First, $(c|0) \in \mathbb{F}_2^n$ gets decoded. The bit-flipping returns the error e. Then, the decryption hashes e to compute the symmetric key, verifies the tag g, and decrypts the second part of the ciphertext, d.

Sparse-Times-Dense Multiplications in $\mathbb{F}_2[x]/\langle x^r - 1\rangle$

Given the problem of computing $h(x) = f(x)g(x) \in \mathbb{F}_2[x]/\langle x^r - 1\rangle$, where $f(x)$ is represented as an array of indices in $I = \{i \mid f_i = 1\}$, where f_i represents the bit in position i of the dense representation of $f(x)$, and $g(x)$ is in the dense representation. Then we have

$$f(x)g(x) = \sum_{i \in I} x^i g(x). \tag{2.10}$$

Algorithm 2.2.3: QcBits decryption

Input : Ciphertext $(c|d|g)$, Private key $H_{priv} = (H_0|H_1)$
Output: Plaintext m or $\perp$

$s \leftarrow (c|0) \in \mathbb{F}_2^n$
$e \leftarrow$Bit-Flipping$(H_{priv}, s) \oplus s$
$key \leftarrow Hash(e)$
if $Check(key, g)$ **then**
    return $m \leftarrow Sdec(key, d)$
else
    return $\perp$

Therefore, the implementation first sets $h = 0$. Then, for each $i \in I$, $x^i g(x)$ is computed and accumulated in h. Note that $x^i g(x)$ is represented as an array of $\lceil r/b \rceil$ b-bit words, so adding $x^i g(x)$ to $h(x)$ can be implemented using $\lceil r/b \rceil$ bitwise-XOR instructions on b-bit words. $x^i g(x)$ can be obtained by rotating $g(x)$ by i bits. In order to perform a constant-time rotation, the implementation makes use of the idea of the Barrel shifter. The idea is to first represent i in binary representation

$$(i_{k-1} i_{k-2} \cdots i_0)_2. \tag{2.11}$$

Since $i \leq r - 1$, it suffices to use $k = \lfloor \lg (r-1) \rfloor + 1$. Then, for j from $k - 1$ to $\lg b$, a rotation by 2^i bits is performed.One of the unshifted vector and the shifted vector is chosen (in a constant-time way) and serves as the input of the next j. After dealing with all $i_{k-1}, i_{k-2}, \ldots, i_0$, a rotation of $(i_{\lg b - 1} i_{\lg b - 2} \cdots i_0)_2$ bits is performed using a sequence of logical instructions.

The constant-time multiplication algorithm which is used to secure private syndrome computation Hc^T as a countermeasure against timing attacks is shown (along with a toy example) in Appendix A.

2.3 Power Analysis Attacks

Concerning post quantum cryptography, QC-MDPC(/LDPC) cryptosystems are the most promising ones, therefore, they have drown the attention of many researchers.

In the following subsections, three different papers which explore the robustness against power analysis attacks of the state of the art of QC-

MDPC(/LDPC) cryptosystems will be presented.

In the first paper, by C. Chen et al. [13], the vulnerabilities exposed by a state of the art FPGA implementation of McEliece are exploited leading to a full key recover. The researchers introduce a vertical and an horizontal side-channel attack and subsequently they suggest possible countermeasures.

In the second paper, by M. Rossi et al. [42], the researches test a state of the art software implementation, QcBits (already presented in subsection 2.2). In particular, they mount a DPA attack targeting the constant time multiplication suggested by Tung Chou and they eventually present a mask-based countermeasure to secure the algorithm.

The third paper, by B.-Y. Sim et al. [46], extends the work of Rossi et al. presenting a multiple-trace attack and a single-trace attack against the constant-time multiplication adopted in QcBits.

For the rest of the section, the parity check matrix H represents the secret key of the cryptosystems under investigation. Let $H_i \in \mathbb{F}_2^{r\times r}$ be the *i-th* circulant sub-matrix of $H \in \mathbb{F}_2^{r\times n}$ such that $H = (H_0|\dots|H_{n_0-1})$, $n = r\times n_0$. The first row of each block H_i is referred to as $\mathbf{h_i}$ and $h_{i,j}$ represents the *j-th* bit of the first row of the *i-th* block, while w denotes the sum of the weights (number of bit set to 1) of the first row of each block (i.e., the weight of the first row of the matrix H).

2.3.1 Vertical and Horizontal Attack on FPGA McEliece (2016)

This subsection presents the work of C. Cheng et al. [13].

In this paper, Cheng et al. present a vertical and a horizontal SCA exploiting leakages during the syndrome computation of the decryption for a state of the art FPGA implementation of McEliece proposed in [51].

QC-MDPC McEliece

The QC-MDPC McEliece public key cryptosystem uses t-error correcting (n, r, w)-QC-MDPC codes, where $r = n - k$ the co-dimension of $C(n, k)$. Using such a code, key generation, encryption and decryption operations can be described as follows.

- *Key generation*: the secret key is comprised of the first rows $\mathbf{h_0}, \dots,$

$\mathbf{h_{n_0-1}} \in \mathbb{F}_2^r$ of the n_0 parity-check matrix blocks $H_0, \ldots, H_{n_0-1}$. These rows are chosen at random and it has to be ensured that their weights sum up to w. Iterated cyclic rotation of the $\mathbf{h_i}$ yields the parity-check matrix blocks $H_0, \ldots, H_{n_0-1} \in \mathbb{F}_2^{r \times r}$ and thereby the secret parity-check matrix $H = (H_0 | \ldots | H_{n_0-1}) \in \mathbb{F}_2^{r \times n}$. Assuming the last to be non-singular, the public key is obtained as generator matrix $G = (I|P)$ in standard form, where, in case of $n_0 = 2$,

$$\mathbf{P} = \begin{pmatrix} (H_{n_0-1}^{-1} \cdot H_0)^T \\ (H_{n_0-1}^{-1} \cdot H_1)^T \\ \vdots \\ (H_{n_0-1}^{-1} \cdot H_{n_0-2})^T \end{pmatrix}$$

- *Encryption*: to encrypt a message $m \in \mathbb{F}_2^k$, an error vector $e \in \mathbb{F}_2^n$ of weight $wt(e) \leq t$ is chosen at random. The ciphertext is then computed as $c = (m \cdot G \oplus e) \in \mathbb{F}_2^n$.

- *Decryption*: to decrypt a ciphertext $c \in \mathbb{F}_2^n$, a t-error correcting QC-MDPC decoder is applied to c recovering $m \cdot G$. Since G is in systematic form, the message m can be simply read off from the first k positions of $m \cdot G$.

The target under investigation is a lightweight implementation of QC-MDPC McEliece for re-configurable devices by [51]. The chosen parameters are for an 80-bit security level: $n_0 = 2, n = 9602, r = 4801, w = 90, t = 84$.

Vertical attack on the syndrome computation

The syndrome s is computed by processing the ciphertext c in a bitwise fashion. If the j-th bit is set, i.e., $x_j = 1$, then the j-th row of H is added to the syndrome s. The implementation adds two 32-bit words in parallel: one word of the rotated $\mathbf{h_0}$ and one word of $\mathbf{h_1}$ are processed in each clock cycle. For the first set bit $c_i = 1$, the zeroed syndrome is overwritten with (a shifted version of) $\mathbf{h_0}$ or $\mathbf{h_1}$. Thus, assuming a Hamming distance function, each bit $h_{i,j}$ will leak in a 32-bit word. Exploiting this leakage, they mounted a vertical DPA analyzing power traces from chosen ciphertext of weight 1, i.e., all the possible c such that $wt(c) = 1$.

Horizontal attack on the key rotation

Since the Block Random Access Memories (BRAMs) store only the first row of each block of H, they need to be rotated by one bit to generate the next rows during the syndrome computation.

The key rotation is implemented as follows: in the first clock cycle, the least significant bit (LSB) is loaded from the last memory cell. The first 32-bit of the row to be rotated are loaded next. In all following clock cycles, the succeeding 32-bit blocks of the row are read and overwritten by the rotated preceding 32-bit block. The LSB of each 32-bit block is delayed by a flip-flop and becomes the most significant bit (MSB) of the following block. That is, in each clock cycle (151 clock cycles per rotation) one bit $h_{i,j}$ (the LSB of the last accessed word) is written to the carry register, causing a leakage λ_{carry}. In the following clock cycle, that bit is overwritten with the LSB of the next word, $h_{i,j+32}$. Assuming an Hamming distance leakage function, and assuming that $h_{i,j+32} = 0$, it is possible to distinguish the case when $h_{i,j} = 0$ from $h_{i,j} = 1$.

All rotations together result in a total of 4801×150 carry register overwrites for each $\mathbf{h_i}$. Since there are 4801 bits in $\mathbf{h_i}$, each bit is written to the carry register 150 times. The corresponding clock cycles l are then identified and their corresponding leakage $\lambda_i(j, l)$ is combined, allowing them to mount a horizontal SCA. A null ciphertext is used to avoid the leakage contribution of the syndrome calculation.

Rotating the two parts of the secret key is implemented in parallel, which means that the 4801-bit rows of the first and the second part of the parity-check matrix are rotated at the same time. This means that the attack retrieves information about $\mathbf{h_0} + \mathbf{h_1}$.

Full key recover

Due to noise observed in both attacks and leakage overlapping observed in the horizontal one, there are probably false positive errors in the recovered bits. To recover the full key correctly with either attack strategy, starting from equation

$$\begin{aligned} &\mathbf{h_0} = \mathbf{h_1} \times Q^T && \text{, vertical attack} \\ &\mathbf{h_0} \oplus \mathbf{h_1} = \mathbf{h_1} \times (Q^T \oplus I_{4801}) && \text{, horizontal attack} \end{aligned}$$

they ended up with a linear system of equations that possibly can be solved and yields a unique candidate for $\mathbf{h_1}$.

Countermeasure

The proposed countermeasure is presented in [15] which consists in a threshold implementation inspired masking with two to three shares to key and syndrome during syndrome computation and during the decoding step to achieve a protection against first-order side-channel attacks.

2.3.2 Vertical Attack on QcBits (2017)

This subsection presents the work of M. Rossi et al. [42].

In this paper Rossi et al. present a side-channel assisted cryptanalytic attack against QcBits (presented in subsec. 2.2). In contrast to Guo et al.'s attack in [24], this attack focuses on the first step of the decoding process and is independent of its failure probability. This attack only requires the attacker to observe about 200 power traces for the implementation under analysis. The attack also works for both the 80-bit and 128-bit security versions. The attack consists of two steps:

i. A DPA attack targeting the syndrome computation of the decryption operation that is able to recover some information about the half of the private key (H_0) employed during the computation;

ii. A linear algebra computation which takes advantage of the sparseness of the private key and that, in the end, allows the attacker to recover the entire secret key.

The DPA targets the syndrome computation at line 2 of the bit-flipping decoding (Alg. 2.2.2) to recover some partial information about the secret matrix H_0.

General leakage model

The computation of the syndrome in the bit-flipping decoding (Step 2 in Alg. 2.2.2) can be written as:

$$H_{priv} \cdot \begin{pmatrix} c^T \\ 0 \end{pmatrix} = (H_0|H_1) \cdot \begin{pmatrix} c^T \\ 0 \end{pmatrix} = H_0 \cdot c^T \tag{2.12}$$

Since H_0 is a circulant matrix with few ones, QcBits represents it as a list of the indices $\{x_0, \ldots, x_{(\frac{w}{2}-1)}\}$ of the nonzero elements of the first row $\mathbf{h_0}$. H_0 can be decomposed as a sum of $w/2$ rotation matrices

$$H_0 = R_{x_0} + \ldots + R_{x_{(\frac{w}{2}-1)}}. \tag{2.13}$$

Multiplying c^T by R_{x_i}, $0 \leq i \leq \frac{w}{2} - 1$, results in a left circular shift of c by x_i positions. Hence the syndrome computation at Step 2 (Alg. 2.2.2) can be accomplished by computing the rotated ciphertexts for each index x_i, and XORing them together. In fact, this is how QcBits implements the multiplication. In a loop, each rotated version of c is stored into a temporary memory location as it is calculated, and then XORed with the partial XOR sum from the previous loop iteration.

The SCA model assumes that the power consumption of the device depends on whether the leftmost bit (bit position 0) of each rotated version of c is either 0 or 1.

For the attack evaluation, Rossi et al. used the reference C version of QcBits [16] with 80 and 128 bits of security. To do so, they ported the code to run on ChipWhisperer evaluation platform designed by Colin O'Flynn [37].

To recover the information about H_0 they attacked the unknown indices $\{x_0, \ldots, x_{(\frac{w}{2}-1)}\}$ sequentially using standard DPA. They first made a guess for all possible values for the unknown x_0. For each of those guesses, they sorted the traces T_j into two partitions based on whether the leftmost bit of each rotated version of c was a zero or a one. They averaged the traces in the two partitions separately and computed the difference of the averages. Large spikes in the difference trace indicated a leak of information. The DPA process is then repeated for each of the unknowns x_i.

Since from the $(i+1)$ to $(i+W)$-th bits, where W is the word size of the device, are saved into the same register, there will be W candidates for the

Table 2.1: Approximate number of attempts in the worst case

Security level	**Architecture**			
	8-bit	**16-bit**	**32-bit**	**64-bit**
80-bit	22	950	2^{23}	2^{58}
128-bit	40	3500	2^{26}	2^{64}

index x_i. Hence, it is impossible to find accurate secret indices just with this process. For the full recovery of the secret key, it is thus required to solve linear equations.

Setting $Q = P^{-1}$ we can write

$$Q \cdot H_0^T = H_1^T. \tag{2.14}$$

The matrices H_0 and H_1 are sparse circulants defined by their first rows $\mathbf{h_0}$ and $\mathbf{h_1}$ respectively. We can therefore write the last equation as the system of linear equations

$$Q \cdot \mathbf{h_0}^T = \mathbf{h_1}^T. \tag{2.15}$$

Where Q is dense and known, $\mathbf{h_0}$ is sparse and partially known and $\mathbf{h_1}$ is sparse and unknown.

From this, they kept only the intervals of $\mathbf{h_0}$ where they knew there were at least a bit set to one, reducing the Q matrix accordingly. They added a parity equation for each of the interval and finally they obtained a square system of equations by randomly selecting the right number of entries from $\mathbf{h_1}$ and keeping the corresponding rows of Q (retaining all the parity equations). If all the selected entry from $\mathbf{h_1}$ are actually zeroes, then the value of $\mathbf{h_0}$ (such that $wt(\mathbf{h_0}) = \frac{w}{2}$ and $wt(Q \cdot \mathbf{h_0}^T) = \frac{w}{2}$) is among the solution of the resulting system of equations. If this is not the case, the final step is repeated with different random subvectors of $\mathbf{h_1}$ until a solution is found.

Table 2.2: Approximate solving times in SAGE (for the algebraic part of the attack) on one core of 2.9 GHz Core i5 MacBook Pro

Security level	**Architecture**			
	8-bit	**16-bit**	**32-bit**	**64-bit**
80-bit	0.4 sec	15 sec	16 h	$\approx 530y$
128-bit	2 sec	4 min	$\approx 7d$	$\approx 790,000y$

Countermeasure

They proposed a marking technique to help defending against SCA during the syndrome calculation in QcBits. Since QC-MDPC codes are linear, the XOR of two codewords is another codeword. Also, all codewords are in the nullspace of the parity check matrix H_{priv}. It is then possible to mask the corrupted codeword $(c|0)$ by XORing it with a random codeword c_m before passing it to the syndrome calculation:

$$H_{priv} \cdot ((c|0) \oplus c_m)^T = H_{priv} \cdot (c|0)^T \oplus H_{priv} \cdot c_m^T = H_{priv} \cdot (c|0)^T. \quad (2.16)$$

2.3.3 Multiple and Single Trace Attack to QcBits (2019)

This subsection presents the work of B.-Y. Sim et al. [46].

In this paper, Sim et al. present two different SCA against the constant-time multiplication introduced by Chou for secure syndrome computation in QcBits [16]. The first is a multiple-trace attack which enhances the result obtained by Rossi et al. [42]. The second attack is a novel single-trace attack which allows to recover the secret key even when using ephemeral keys or DPA countermeasures [15, 42]. Finally, they claim the vulnerability of BIKE [1] and LEDAcrypt [3, 4], two of the second-round candidates of the NIST PQC standardization, to their attacks.

Using QcBits, a syndrome of a vector $c' = (c|0) \in \mathbb{F}_2^n$ is calculated by

$$H_{priv} \cdot \begin{pmatrix} c_0^T \\ 0 \end{pmatrix} = (H_0|H_1) \cdot \begin{pmatrix} c_0^T \\ 0 \end{pmatrix} = H_0 \cdot c^T \quad (2.17)$$

$$H_0 \cdot c^T = \sum_{i \in I} R_i(c_0)^T, \tag{2.18}$$

where $R_i(c_{(k)})$ is an i-bit left rotation of $c_{(k)}$. For any vector $a \in \mathbb{F}_2^r$ there exists an isomorphism that maps a to the polynomial ring $\mathbb{F}_2[x]/\langle x^r - 1\rangle$. Thus, $c_{(k)}$ can be considered to be a polynomial, and $R_i(c_{(k)})$ can be calculated by the multiplication $x^d c_k$ in $\mathbb{F}_2[x]/\langle x^r - 1\rangle$, where $d = r-1$. Chou then suggested a *constant-time multiplication* $x^d c_{(k)}$, as shown in Algorithm 2.3.1, to secure private syndrome computation Hc^T against a TA.

Algorithm 2.3.1: Constant-Time Multiplication in $\mathbb{F}_2[x]/\langle x^r - 1\rangle$ (refer to [16])

Input : $d = (d_{l-1}, \ldots, d_0)_2$, $0 \le d \le r-1$, binary representation of the shift amount, i.e., an element of the sparse key.
$c_{(k)} = (c_{L-1}, \ldots, c_0)_{2^W}$, $L = \lceil r/W \rceil$, binary vector $1 \times r$ in dense representation

Output: $x^d c_{(k)}$, $1 \times r$ binary vector

Data: L: number of architecture words needed to represent the binary vector $c(k)$; W: word length of the architecture; $l = \log_2 r$: number of bits needed to represent the shift amount.

```
1  v ← 0, w ← c_(k)
2  for i = l − 1 down to log2 W do ▶ word unit rotation lines 2 to 10
3      d_i ← (d ≫ (l − 1 − i)) & 1
4      mask ← 0 − d_i
5      us ← 1 ≪ (i − log2 W)
6      ptr ← v, v ← w, w ← ptr
7      for j = 0 up to L − 1 − us do
8          w[j] ← (v[j + us] & mask) ⊕ (v[j] & ¬mask)
9      for j = 1 up to us do
10         w[j+L−1−us] ← (v[j−1] & mask)⊕(v[j+L−1−us] & ¬mask)
11 low ← d & ((1 ≪ log2 W) − 1)        ▶ bit rotation lines 11 to 18
12 high ← W − low
13 tmp ← w[0]
14 for j = 0 up to L − 2 do
15     w[j] ← w[j] ≫ low
16     w[j] ← w[j] | (w[j + 1] ≪ high)
17 w[L − 1] ← w[L − 1] ≫ low
18 w[L − 1] ← w[L − 1] | (temp ≪ high)
19 return w
```

Multiple-trace attack

Based on the structure of the constant-time multiplication shown in Algorithm 2.3.1, they divided the attack into two parts to find the secret index $d = (d_{l-1}, \ldots, d_0)$: the word unit rotation to find $(d_{l-1}, d_{l-2}, \ldots, d_{\log_2 W})$, and the bit rotation to find $(d_{\log_2 W - 1}, \ldots, d_1, d_0)$.

In Algorithm 2.3.1, in the steps 7 and 8, both the rotated value $v[j + us]$ and the unrotated value $v[j]$ are loaded but, depending on the value of d_i, only one of them is selected to be saved in $w[j]$. Thus, having the word size equal to 8 bits:

$$w[j] = \begin{cases} (v[j + us] \;\&\; \texttt{0x00}) \oplus (v[j] \;\&\; \texttt{0xff}) = v[j] & , \texttt{if } d_i = 0; \\ (v[j + us] \;\&\; \texttt{0xff}) \oplus (v[j] \;\&\; \texttt{0x00}) = v[j] & , \texttt{if } d_i = 1. \end{cases}$$

They defined 2 properties:

Property 1. The mask value is $0 - d_i$; therefore, it is `0x00` when $d_i = 0$. Consequently, in the steps 7 and 8 in Algorithm 2.3.1, $v[j]$ is saved to $w[j]$. Thus, $v[j]$ is loaded and saved, but $v[j + us]$ is only loaded. Contrariwise, when $d_i = 1$, the mask value is `0xff` on an 8-bit processor and $v[j + us]$ is saved to $w[j]$. Thus, $v[j + us]$ is loaded and saved, but $v[j]$ is only loaded.

Property 2. If $d_i = 0$, then the unrotated value is chosen, i.e. $v[j]$ is saved to $w[j]$, which has the same index. Contrariwise, when $d_i = 1$, the rotated value is chosen, i.e. $v[j + us]$ is saved to $w[j]$, which has a different index.

According to their leakage model, they remodeled P_{total} as $\epsilon \cdot wt(data) + P_{noise}$, where ϵ is a constant, i.e., they defined a linear relationship between P_{total} and $wt(data)$. They thus determined the positions where the $v[j]$ value was used by calculating the Pearson correlation coefficient between the Hamming weight of the $v[j]$ values and power consumption traces.

If $d_i = 0$, then the mask value is `0x00`; therefore, the power consumption with respect to the $v[j]$ value occurs sequentially twice in the steps 7 and 8 of Algorithm 2.3.1 according to the Property 1. Contrariwise, when $d_i = 1$, the mask value is `0xff` on an 8-bit processor; therefore, the power

consumption with respect to the $v[j]$ value occurs once in the steps 7 and 8 of Algorithm 2.3.1. In this way they identified d_{l-1}.

To recover the subsequent bits of the index (up to $d_{\log_2 W}$), they used Property 2, knowing that the power consumption related to the $v[j]$ value occurs sequentially twice in the same iteration where the loaded and saved operations are executed according to the prior key bits $d_i + 1$ when $d_i = 0$. Otherwise, the power consumption related to the $v[j]$ value occurs sequentially twice in a different iteration from where the loaded and saved operations are executed based on the prior key bits d_{i+1} when $d_i = 1$.
Algorithm 2.3.2 describes the attack procedure.

Algorithm 2.3.2: Multiple-Trace Attack on the Word Unit Rotation

Input : a trace set $T = \{T^1, \cdots, T^N\}$ and an input value set $C = \{c^1, c^2, \cdots, c^N\}$
Output: $(d_{l-1}, d_{l-2}, \cdots, d_{\log_2 W})$

1 Calculate the correlation coefficient between T and $C = \{c^1[0], c^2[0], \cdots, c^N[0]\}$
2 **if** high correlation occurs twice at 1st iteration **then** ▶ finding d_{l-1}
3 $d_{l-1} \leftarrow 0$
4 **else**
5 $d_{l-1} \leftarrow 1$
6 **for** $i = l - 2$ down to $\log_2 W$ **do** ▶ finding $(d_{l-2}, \cdots, d_{\log_2 W})$
7 **if** high correlation occurs twice at same position with d_{i+1} **then**
8 $d_i \leftarrow 0$
9 **else**
10 $d_i \leftarrow 1$
11 **return** $(d_{l-1}, d_{l-2}, \cdots, d_{\log_2 W})$

To derive the remaining bits $(d_{log_2 W-1}, \cdots, d_1, d_0)$, they made guesses for the leftmost word

$$(w[0] \gg (low)) | (w[1] \ll (W - low))$$

of the result of the bit rotation $x^d c_{(k)}$ of the Algorithm 2.3.1 (first iteration Steps 14 to 16). At this point the low value and the last $log_2 W-$bit value of

d are the same. With this information, they mounted a Correlation Power Analysis (CPA) finding the last log_2W-bit of d.

Single-trace attack

Also the single-trace attack is divided into two parts to find d: the word unit rotation to find $(d_{l-1}, d_{l-2}, \ldots, d_{\log_2 W})$, and the bit rotation to find $(d_{\log_2 W-1}, \ldots, d_1, d_0)$.

The $mask$ value determined by the value d_i is used to check whether the rotated value is saved or not. Therefore, there exists a phase, such as in the step 3 of Algorithm 2.3.1, in which d_i bit are extracted from the l-bit secret index string $d = (d_{l-1}, d_{l-2}, \cdots, d_0)_2$ and saved before performing the word unit rotation. Then, the values mask and $\neg mask$ are computed and saved. Besides, when the steps 7 and 8 of Algorithm 2.3.1 are executed, the values $mask$ and $\neg mask$ are loaded. They classified the power consumption properties of Algorithm 2.3.1 as follows:

s.1 $d_i \leftarrow (d \gg (l-1-i)) \,\&\, 1$	▶ d_i is saved;
s.2 $mask \leftarrow 0 - d_i$	▶ $mask$ is saved;
s.3 $\neg mask$ is calculated	▶ $mask$ is loaded, $\neg mask$ is saved;
s.4 $w[j] \leftarrow (v[j+us] \,\&\, mask) \oplus (v[j] \,\&\, \neg mask)$	▶ $mask$ and $\neg mask$ are loaded.

Furthermore, they introduced new properties:

Property 3. The secret bit d_i is 0 or 1. Thus, if $d_i = 0$, the power consumption is associated with 0 when extracting and saving the d_i value. Likewise, if $d_i = 1$, then the power consumption is associated with 1.

Property 4. The $mask$ value is $0-d_i$; therefore, it is `0x00` when $d_i = 0$, and the power consumption is related to 0. Contrariwise, when $d_i = 1$, the mask value is `0xff` on an 8-bit processor, and the power consumption is related to 8, which is the Hamming weight of the $mask$ value.

Property 5. The $\neg mask$ value is the bitwise inversion value of the mask value. Consequently, it is 0xff on an 8-bit processor when $d_i = 0$ and the power consumption is related to 8. Contrariwise, the power consumption is related to 0 when $d_i = 1$.

Starting from multiple traces measured from the syndrome computation (N runs with different ciphertexts $(C_0, \cdots, C_N)$ for each different known key), they recovered a PoI (maximum SOST [23] value) for each round corresponding to each of the bits, from d_{l-1} to $d_{\log_2 W}$ (representing the most significant part of the secret index d). These PoIs corresponds to an instant in the trace in which the operation (the one with the highest difference in power consumption depending on the value of d_i among the aforementioned operations **s.1**, **s.2**, **s.3**, **s.4**) is being processed.

They then classified this PoIs into two groups G_1 and G_2 using the k-means clustering algorithm. They finally distinguished which group was associated with $d_i = 0$ or $d_i = 1$ knowing the intermediate corresponding to the PoIs and computing the average values of each group. In this way, they recovered the first secret key bits $(d_{l-1}, d_{l-2}, \cdots, d_l og_2 W)$.

Algorithm 2.3.3 describes the attack procedure.

Algorithm 2.3.3: Single-Trace Attack on the Word Unit Rotation

Input : A trace T
Output: $(d_{l-1}, d_{l-2}, \cdots, d_{\log_2 W})$

for $i = l - 1$ down to $\log_2 W$ **do**
    Select PoIs p_i of **word unit rotation** operation associated with d_i
Classify p_i into two groups, G_1 and G_2, using the k-means clustering algorithm
Calculate the average values AVG_1 and AVG_2, respectively, of G_1 *and* G_2
for $i = l - 1$ down to $\log_2 W$ **do**
    if $p_i \in G_1$ **then** ▶ assume that $AVG_1 < AVG_2$
        $d_i \leftarrow 1$ ▶ $d_i = 1$ when it follows the property 4
    else
        $d_i \leftarrow 0$ ▶ $d_i = 0$ when it follows the property 4
return $(d_{l-1}, d_{l-2}, \cdots, d_{\log_2 W})$

To recover the remaining bits, $(d_{log_2 W-1}, \cdots, d_1, d_0)$, they applied a SPA.

If the processors only provide single bit shift instructions, a 1-bit right shift operation is repeated *low* times, and a 1-bit left shift operation is repeated *high* times in the steps 14 to 18 of the Algorithm 2.3.1. A SPA thus allows to identify the number of 1-bit left shift operations that is the last $log_2 W-$bit value of d.

If the processors support a barrel shifter, as the most commonly used 32-bit and 64-bit processors do, W candidates remain, requiring to recover accurate indices with additional algebraic computations, similarly as discussed in [42].

2.4 LEDAcrypt

The LEDAcrypt [3] cryptosystem is the result of merging two cryptographic systems, LEDAkem and LEDApkc and provides three cryptographic primitives based on binary linear error-correcting codes:

i. An IND-CCA2 key encapsulation method, named LEDAcrypt-KEM.

ii. An IND-CCA2 public key encryption scheme, named LEDAcrypt-PKC.

iii. An IND-CPA key encapsulation method optimized for employment in an ephemeral key scenario, while providing resistance against accidental key reuse, named LEDAcrypt-KEM-CPA.

LEDAcrypt exploits the advantages of relying on Quasi-Cyclic Low-Density Parity-Check codes to provide high decoding speeds and compact key pairs.

Both key encapsulation methods are based on the OW-CPA Niederreiter scheme, while the public key encryption scheme is based on the McEliece encryption and decryption primitives.

Algorithm 2.4.1: DECRYPT^{Nie}

Input : s: *syndrome*; $1 \times p$ binary vector.
$sk^{\text{Nie}} = \{H, Q\}$ private key;
Output: $e = [e_0, \ldots, e_{n_0-1}]$: *error*; sequence of n_0 binary vectors with size $1 \times p$.
res: Boolean value denoting if the decryption ended successfully (**true**) or not (**false**)
Data: $p > 2$ prime, $ord_p(2) = p - 1$, $n_0 \geq 2$

1 $L \leftarrow HQ$
2 $s' \leftarrow L_{n_0-1}s$
3 $\{e, \textbf{error}\} \leftarrow$
4 LEDAdecoder(s', sk^{Nie})
5 **if** ***res*** $=$ ***false*** **then**
6 $e \leftarrow \perp$
7 **return** $(e, \textbf{res})$

Algorithm 2.4.2: DECRYPT^{McE}

Input : $c = [c_0, \ldots, c_{n_0-1}]$: *error* affected codeword; $1 \times pn_0$ binary vector, where each c_j is a $1 \times p$ vector with $0 \leq j < n_0$;
$sk^{\text{Nie}} = \{H, Q\}$ private key;
Output: $u = [u_0, \ldots, u_{n_0-1}]$: *message*; sequence of n_0 binary vectors with size $1 \times p$.
$e = [e_0, \ldots, e_{n_0-1}]$: *error* sequence of n_0 binary vectors with size $1 \times p$.
res: Boolean value denoting if the decryption ended successfully (**true**) or not (**false**)
Data: $p > 2$ prime, $ord_p(2) = p - 1$, $n_0 \geq 2$

1 $L \leftarrow HQ$
2 $s \leftarrow Lc^T$
3 $\{e, \textbf{error}\} \leftarrow$
4 LEDAdecoder(s, sk^{McE})
5 **if** ***res*** $=$ ***true*** **then**
6 **for** $j = 0$ **to** $n_0 - 1$ **do**
7 $u_j \leftarrow c_j + e_j$
8 **else**
9 $e \leftarrow \perp; u \leftarrow \perp$
10 **return** $(e, \textbf{res})$

Algorithm 2.4.1 and Algorithm 2.4.2 describes the decryption algorithm of Niederreiter and McEliece cryptosystems, instantiated with QC-LDPC codes. These algorithms are employed respectively in the decapsulation algorithm of LEDAkem and in the decryption transformation in LEDApkc.

Both Algorithm 2.4.1 and Algorithm 2.4.2 compute a sparse to dense multiplication at line 2. This kind of operation, if not carefully implemented, may be vulnerable to side channel attacks based on timing analysis. The constant-time multiplication proposed by T. Chou at CHES 2016 [16] represents the state of the art to secure this computation against timing attacks.

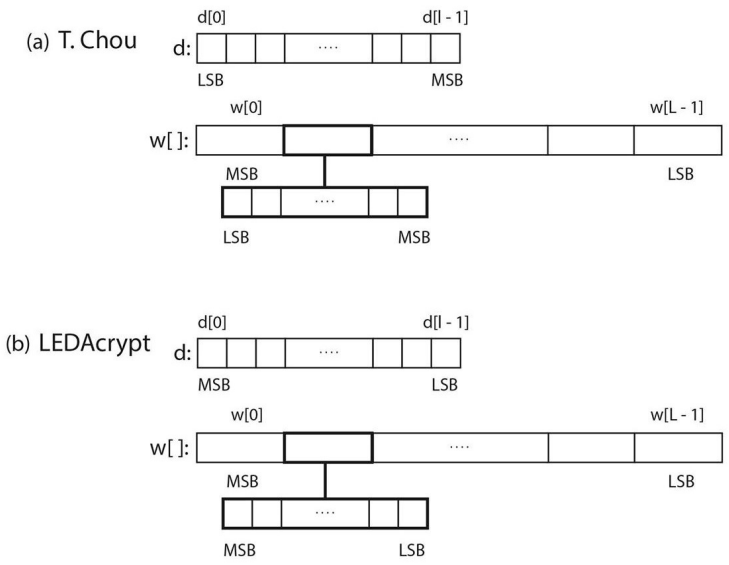


Figure 2.1: Comparison between the convention adopted by T. Chou (a) and by LEDAcrypt showing the representation of an index d of the secret key and the ciphertext vector $w[\,]$.

Algorithm 2.4.3 is the constant-time multiplication proposed by T. Chou and rewritten using the LEDAcrypt [3] convention.

In 2017 Rossi et al. described a differential power analysis against the Chou implementation of the constant-time multiplication, but provided also a valid countermeasure to mitigate the attack (refer to [42]).

In 2019 Sim et al. proposed a novel single trace attack based on a different power consumption leakage, but leaving open the question of a sound countermeasure to secure the multiplication against this threat. Moreover, they stated that this attack might be able to allow a full recover of the secret matrix L of LEDAcrypt.

Algorithm 2.4.3: Constant-Time Multiplication in $\mathbb{F}_2[\mathrm{x}]/\langle x^r - 1\rangle$

Input : $d = (d_{l-1}, \ldots, d_0)_2$, $0 \leq d \leq r-1$, binary representation of the shift amount, i.e., an element of the sparse key.
$c_{(k)} = (c_{L-1}, \ldots, c_0)_{2^W}$, $L = \lceil r/W \rceil$, binary vector $1 \times r$ in dense representation

Output: $x^d c_{(k)}$, $1 \times r$ binary vector

Data: L: number of architecture words needed to represent the binary vector $c(k)$; W: word length of the architecture; $l = \log_2 r$: number of bits needed to represent the shift amount.

1 $w \leftarrow c_{(k)}$, $tail \leftarrow r \bmod W$
2 **for** $i = 0$ up to $l - \log_2 W - 1$ **do** ▶ **word unit rotation** lines 2 to 14
3 $\quad v \leftarrow w$
4 $\quad d_i \leftarrow (d \gg (l - 1 - i)) \,\&\, 1$
5 $\quad mask \leftarrow 0 - d_i$
6 $\quad us \leftarrow 1 \ll (l - 1 - \log_2 W - i)$
7 $\quad w[0] \leftarrow (v[us] \,\&\, mask) \oplus (v[0] \,\&\, \neg mask)$
8 $\quad$ **for** $j = 1$ up to $L - us - 1$ **do**
9 $\quad\quad w[j] \leftarrow (v[j + us] \,\&\, mask) \oplus (v[j] \,\&\, \neg mask)$
10 $\quad$ **for** $j = 1$ up to $us - 1$ **do**
11 $\quad\quad w[j + L - 1 - us] \leftarrow (((v[j - 1] \ll (W - tail)) \mid (v[j] \gg tail))$
12 $\quad\quad\quad\quad \,\&\, mask) \oplus (v[j + L - 1 - us] \,\&\, \neg mask)$
13 $\quad w[L - 1] \leftarrow (((v[us - 1] \ll (W - tail)) \mid (v[us] \gg tail)) \,\&\, mask)$
14 $\quad\quad\quad\quad\quad\quad\quad\quad \oplus (v[L - 1] \,\&\, \neg mask)$
15 $low \leftarrow d \,\&\, ((1 \ll \log_2 W) - 1)$ ▶ **bit rotation** lines 15 to 29
16 $mask \leftarrow ((low - 1) \gg (W - 1)) - 1$
17 $high \leftarrow W - low$
18 $tmp \leftarrow w[1]$
19 $tmp_2 \leftarrow w[0]$
20 **for** $j = 1$ up to $L - 2$ **do**
21 $\quad w[j] \leftarrow w[j] \ll low$
22 $\quad w[j] \leftarrow w[j] \,|\, ((w[j + 1] \gg high) \,\&\, mask)$
23 $w[L - 1] \leftarrow w[L - 1] \ll low$
24 $tmp_2 \leftarrow tmp_2 \ll ((W - tail) \,\&\, mask)$
25 $tmp_2 \leftarrow tmp_2 \,|\, ((tmp \gg tail) \,\&\, mask)$
26 $w[L - 1] \leftarrow w[L - 1] \,|\, ((tmp_2 \gg high) \,\&\, mask)$
27 $w[0] \leftarrow w[0] \ll low$
28 $w[0] \leftarrow w[0] \,|\, ((tmp \gg high) \,\&\, mask)$
29 $w[0] \leftarrow w[0] \,\&\, ((1 \ll tail) - 1)$
30 **return** w

Chapter 3

Implementation

The aim of this work is to investigate possible attack surfaces exposed by the LEDAcrypt cryptosystem and to design and validate a countermeasure in order to secure it against the state of the art of power analysis attacks.

The state of the art for power analysis attacks against QC-LDPC cryptosystems is described in Section 2.3. Three different papers are reported, each of which discusses a different attack based on power analysis.

For the first two papers, the authors them self proposed a valid countermeasure to secure the cryptosystems against their own attack. These countermeasures are discussed in [15] and in [42]. The first of the two attacks proposed by Sim et al. in [46] can be prevented by the same countermeasures as reported by the authors. Instead, for the single trace attack, Sim et al. suggest the implementation of hiding methods, such as random noise and dummy operations, to increase attack complexity. However, they do not provide any theoretically-sound countermeasure against this attack, they just list it as "one of the interesting future research topics", leaving the question open.

All the above considerations have contributed to draw the focus of this work on the analysis of the single trace attack and the design and validation of a practical countermeasure. Particular attention was given to the evaluation of the reproducibility of the attack, in synthesis and on board, with different Signal to Noise Ratio (SNR).

The different stages of this thesis project are described in the following sections dividing the work into two environments: the analysis in a simu-

lation environment and the study of the attack against real traces acquired from a microcontroller.

3.1 Simulation Environment

This section describes the steps we followed to reproduce and validate the attack in a simulation environment.

Single trace attack based on a Hamming weight (HW) power consumption model. The first step is to simulate the single trace attack as it is described by Sim et al. in [46]. This is done to validate the concept at the basis of the attack and prove it's theoretical reproducibility. This means building the framework to perform the constant-time multiplication proposed by T. Chou integrating it with the LEDAcrypt framework. The Hamming weight is employed to simulate the power consumption during the execution of the algorithm. In this way, several simulated power traces are collected. Only the samples related to the power consumption of the constant-time multiplication are gathered, since the attack does not concern the rest of the framework. After that, the attack is built following the two phases described by the authors: in the *training phase*, the SOST method is applied over the collected traces in order to select the points of interest; in the *evaluation phase*, the single trace attack is mounted over those points which are classified using the k-means clustering algorithm, finally retrieving the secret key.

Since we chose to focus only on the constant-time multiplication, rather then the entire cryptosystem, the only primitive of LEDAcrypt that has been used is the key-generation, along with the LEDAcrypt security parameters, taken directly from the LEDAcrypt source code.

For the constant-time multiplication algorithm, we used the C code provided by T. Chou and modified to be consistent with the convention adopted in LEDAcrypt (see Algorithm 2.4.3).

The implementation developed in order to carry out the training phase in a simulation environment can be summarized as follows:

- A function which generates a key k (employing the key-generation primitive by LEDAcrypt) and performs the constant-time multiplica-

tion between k and c_i, where c_i is the *i-th* randomly generated ciphertext, and $0 \leq i < 100$.

- A Hamming weight function. To simulate the Hamming weight leakage model during the multiplication we computed the Hamming weight corresponding to a number of variables, with enough granularity to characterize the power trace. These values have been collected in a suitable format representing the emulation of 100 power traces corresponding to 100 run of the sparse-to-dense multiplication (using constant-time multiplication) between a fixed key k and a variable ciphertext c_i.

- A function which performs the SOST computation over a collection of simulated power traces.

Since the algorithm of the constant-time multiplication repeats the same operations over each index of the key, we performed the training phase on the single index, identifying the points of interest at index level rather than trace level.

The SOST computation extrapolates the points of interest exploiting the differences in the leakage of multiple runs with two different keys. Working at index level, we collected 100 traces with a fixed key and then, we computed the SOST values processing the samples corresponding to two different key indices. The two indices have been chosen on the basis of the adopted power consumption model. With the Hamming weight model, to enhance the results given by the SOST function, one index have to be the bit-wise complement of the other (e.g, $index_0 = 001101\ldots$, $index_1 = 110010\ldots$), in this way, the sample corresponding to an operation strictly dependent on the value of the *i-th* key-bit will have a high (Hamming weight) value if the key bit is set to 1 (or 0, depending on the operation) and a low value with the usage of the other index, leading to a high SOST value for that time instant (that is, it will highlight as meaningful data that sample of the trace). As Sim et al. explained, the operation involving the two variables $mask$ and $\neg mask$ are the ones that are the most likely to leak information about the secret key (and consequentially, to produce a high SOST value). In fact, we have $mask =$ `0x00`, when the key bit $d_i = 0$ and $mask =$ `0xff...ff` when $d_i = 1$, with $\neg mask$ behaving in a complement fashion. This is particularly clear in the simulation (on a 64-bit architecture) where the simulated power

consumption of $mask$ is given by $\mathsf{HW}(mask)$, that is 0 in one case and 64 in the other (the same considerations hold for $\neg mask$).

To simulate the error present in practical measurements of the traces, a random value extracted from a normal distribution is added to each sample. The experiments are repeated enhancing the standard deviation of the simulated white noise to evaluate the attack under different SNR scenarios.

In Table 3.1 are given the value of some parameters used in the simulation of the constant-time multiplication. Note that the number of blocks of the secret key matrix, the weight of each row, the size of the matrix and, accordingly the number of bits needed to represent an index belong to the security parameters of LEDAcrypt [1]. The last two parameters, depend also on the word length of the architecture.

Table 3.1: Parameters for the simulation on a 64-bit architecture

64-bit word parameters	**value**	**symbol**
Number of blocks N of the secret key L	2	n_0
Weight of a row of each block of L	71	V
Length of a row of each block of L	10253	r
Number of bits to represent an index of L	14	l
Number of bits used for the word unit rotation	8	$l - \log_2 W$
Number of words representing the ciphertext	161	L

To proceed with the evaluation phase of the attack, we have to collect a simulated power trace for the multiplication between the unknown key $\bar{k}$ and a random ciphertext c.

Once we have collected these data, we can extract the samples corresponding to the points of interest within the trace. With the training phase we obtain the time instants where are located the points of interest for the processing of a single index. Being a simulation, and having the trace perfectly aligned and divided by intervals, where each interval of data describes the processing of an index, it is sufficient to project the pattern that identifies the points of interest of one index over each interval to retrieve the most meaningful samples for the whole trace.

If the Hamming weight model is adopted to describe the power consump-

[1] These values refer to the LEDAcrypt specification before march 2020

tion (as suggested by the authors), the way to proceed for the single trace attack is described in Algorithm 2.3.3 by Sim et al. The k-means clustering algorithm is applied on the samples selected in correspondence of the points of interest. The sign of the SOST value (before the squaring) tells you if the point of interest corresponds to an operation which involves $mask$ or $\neg mask$. Depending on this information and the cluster to which the point belongs (after the k-means), the bit value is identified. Finally, the binary representation of the key is retrieved and compared against the secret key $\bar{k}$ used for the multiplication.

Single trace attack based on a Hamming distance (HD) power consumption model. Sim et al. have described the attack over the assumption of the Hamming weight model being suitable to characterize the power consumption leakage. However, this assumption could not be the strongest one in this case scenario. As written in [29], " ... the power consumption of registers in hardware implementations of cryptographic algorithms can be described very well by the HD model ... An attacker can simulate the power consumption of a register by calculating the Hamming distance of the values that are stores in consecutive clock cycles.", moreover, " ... In case of the Hamming weight model, the attacker assumes that the power consumption is proportional to the number of bits that are set in the processed data value. The data values that are processed before and after this value are ignored. Therefore, this power model is in general not very well suited to describe the power consumption of a CMOS circuit. The power consumption of a CMOS circuit rather depends on the fact whether there occurs a transition in the circuit or not, and not on the processed value.", concluding that " ... due to the fact that $0 \rightarrow 1$ and $1 \leftarrow 0$ transitions always lead to slightly different power consumption, $\mathsf{HW}(v)$ is typically at least somehow related to the actual power consumption. This relationship can of course become weak. Attackers therefore use the HD model whenever possible".

The Hamming distance model seemed to better fit the power consumption leakage in this case, therefore, a second simulation of the attack has been carried out, this time using the Hamming distance model to emulate the power traces.

We therefore adapted the original attack to consider a Hamming distance model as follows.

During the training phase, we had to employ different keys with respect to the ones used in the simulation based on the HW.

Employing the HD model, the key indices suitable for the SOST computation have to match the following scheme: one index must be a sequence of the same bit value (i.e., $index_0 = 000\ldots$, or $index_0 = 111\ldots$) and the other must be an alternating sequence of '1' and '0' (i.e., $index_1 = 0101\ldots$, or $index_1 = 1010\ldots$), in this way, the samples corresponding to an operation dependent on the *i-th* key-bit will have a high (Hamming distance) value in case of the "alternating-bit" index and a low value in case the other index is employed, obtaining a high SOST value for that time instant.

Also the evaluation phase of the attack has been adjusted accordingly to the power model. In this case, the string of bits that we retrieve after the clustering process does not represents directly the secret key, but it describes it instead through a "toggle pattern". In other words, looking at this binary string, a bit set to 1 in position i tell us that $d_{i-1} \oplus d_i = 1$ (where d_{i-1} and $d_i = 1$ are respectively the bit of $\bar{k}$ in position $i-1$ and i), which means that, to obtain the bit in position i of the secret key, you have to toggle the bit in position $i-1$. On the other hand, a bit set to 0 in position i tells that $d_{i-1} \oplus d_i = 0$, which means that the bit of $\bar{k}$ in position i is equal to the bit of $\bar{k}$ in position $i-1$. The first bit of the sequence is unknown (since the Hamming distance related to the first bit of the key relays on the first bit it self and the bit at position "-1", which clearly does not exist), thus, we have to guess the first bit of the key, say $d_0 = 0$, apply the toggle pattern to retrieve a key, compare it against the secret key $\bar{k}$ and, if they do not match, we have to recompute the key changing the guess for the first bit, i.e., $d_0 = 1$.

Single trace attack countermeasure based on a Hamming distance power consumption model. To secure the constant-time multiplication under the hypothesis of the Hamming distance model, it has been designed a countermeasure based on the principal of random precharging on data registers. In this way, the power leakage is no more directly correlated to the key-dependent operations which are now masked by a random value.

To simulate the random precharging, a random value is assigned to each sensitive variable before performing any other assignment to it. Then, the Hamming distance on *mask* (to simulate the power consumption) is computed between the random value and the new value of *mask*. This introduces a distortion in the simulated leakage and breaks the correlation between *mask* at time $i-1$ and i.

Since the simulation has been performed on a 64-bit architecture, to generate a 64-bit random value from a uniform distribution, we employed a C code of Mersenne Twister for 64-bit machines, named MT19937-64, coded by Takuji Nishimura and Makoto Matsumoto [35].

3.2 Real Board

After the analysis of the single trace attack performed in a simulation environment, a practical study has been carried out with the main objective of achieving a practical reproducibility of the attack and the validation of the countermeasure designed in the previous phase.

Note that, in order to validate the countermeasure based on random precharging it was first necessary to validate the Hamming distance model, proposed in this work, over the Hamming weight model, suggested by Sim et al. [46].

The practical experiments have been performed on a `Cortex-M7` core mounted on a `STMicroelectronics NUCLEO-F746ZG` board[2]. Details on the measurement setup for power consumption traces can be found in Section 4.2

The rest of this section describes the the work carried out in a real environment, targeting the `STM32F746ZG` board.

Code porting for the `STM32F746ZG` board. The first thing we had to do in order to perform the attack on the microcontroller was to adapt the framework to the `STM32F746ZG` board.

The practical experiments have been carried out under the assumption that the attacker is able to perform the training phase and thus to extrapolate the points of interest needed to complete the single trace attack. According to this assumption, the training phase was not implemented for the real

[2] `https://www.st.com/en/evaluation-tools/nucleo-f746zg.html`.

board environment.

The architecture size of the adopted microcontroller is 32-bits, therefore some of the parameters are different from the ones used for the 64-bits architecture in the simulation. In Table 3.2 are reported the parameters involved in the multiplication algorithm for a 32-bit architecture. Refer to Algorithm 2.4.3 to see how the different parameters affect the computation.

Table 3.2: Parameters for the real board (32-bit architecture)

32-bit word parameters	**value**	**symbol**
Number of blocks N of the secret key L	2	n_0
Weight of a row of each block of L	71	V
Length of a row of each block of L	10253	r
Number of bits to represent an index of L	14	l
Number of bits used for the word unit rotation	9	$l - \log_2 W$
Number of words representing the ciphertext	321	L

The following adjustments were needed to successfully deploy the code on the `STM32F746ZG` microcontroller:

- To avoid the deployment of the LEDAcrypt code on the microcontroller, we simply precomputed the operands of the multiplication (the key k and the ciphertext c) and saved them in a C header.
- The code of the constant-time multiplication has been rewritten in a three-address form in order to highlight every operation which needed to be masked with the random precharging. Moreover, all the variables have been declared *volatile* to ensure a store operation each time a variable was written and a load operation each time a variable was read. This was made with the intention of implementing the random precharging at C level.
- To generate the random values needed for the countermeasure, we exploited the TRNG (True Random Number Generator) component embedded within the board.
- Since the training phase was not implemented, a block of nop[3] oper-

[3]A nop is an assembler instruction that does not execute any operation during its processing.

ations has been inserted before and after the *mask* computation (line 5, Algorithm 2.4.3), which has been chosen as the target operation of the attack, so that the points of interest were easier to identify.

Single trace attack with no countermeasure. Once we have captured the trace with the oscilloscope, we proceeded with the attack organized in the following steps:

i. Parsing of the acquired trace.

ii. Collection of the intervals of nop containing the samples associated with the computation of *mask*.

iii. Shape definition: from the acquired trace, one singular characteristic shape has been identified and used as a template to extrapolate the leakage of the target operation from the nop intervals.

iv. Shape detection: several filters are applied to the samples in order to deal with the noise present in the measured trace. In this way, the characteristic shape is identified and the meaningful samples are retrieved for each key bit.

v. Evaluation: the selected samples are compared against a threshold to determine if the corresponding key bits are set to 0 or 1. This step is repeated for different thresholds to find the value which maximizes the number of detected bits.

To recognize the leakage model, the binary string resulting from the attack was compared against both the key and the "toggle-pattern" describing the key, counting the number of positive matches for both cases.

Countermeasure validation. The single trace attack has been performed against the protected version of the algorithm to verify the applicability and efficacy of the suggested countermeasure (random precharging) which was designed validated in a simulation environment. Firstly, the random precharging was implemented by assigning a random value to each sensitive variable right before their usage. After the analysis of the traces we observed that working at C level (with the *volatile* attribute) was not enough to ensure the correct behavior of the countermeasure. For this reason, it was

necessary to implement the countermeasure in assembly, loading random values in registers and storing random variables in memory locations for the sensitive variables. Although the countermeasure was implemented on the entire algorithm, since we did not implemented the training phase for the board environment, the effect of the countermeasure has been analyzed in depth only for the target operation. However, visual evidence of the efficacy of the countermeasure on the rest of the algorithm was found on the profile of the acquired traces. More details are given along with the results of the practical experiments in Section 4.2.

Chapter 4

Experimental Evaluation

This chapter will present the results obtained with the simulation of the single trace attack and its application in a real board scenario.

4.1 Experiments in Simulation Environment

For the simulation, the attack has been implemented and tested on a 64-bit architecture. According to the parameters presented in Table 3.1, the number of bits, for each index, involved during the word unit rotation (see Algorithm 2.4.3) is equal to 8, i.e., the single trace attack presented by Sim et al. [46], performed against the constant-time multiplication implemented with these parameters, aims to recover the first 8 most significant bit of each index of the secret key employed in the operation (where each index represents the position of a set bit in the first row of a block of the quasi-cyclic key matrix).

In order to simulate the power traces, the leakage has been modeled over the Hamming weight (or Hamming distance, depending on the power model adopted) of the value written into the variables during the computation of the algorithm. In this way, the power traces are represented by a long series of number taking values between 0 and 64.

The noise on the power consumption measures has been modeled as a normal distribution $\mathcal{N}(\mu, \sigma^2)$ with mean $\mu = 0$ and standard deviation $\sigma \in \{1, 6\}$.

For the training phase, we produced 100 power traces to compute the SOST values identifying the points of interest.

Figure 4.1a shows the power consumption simulated under the assumption of a Hamming weight leakage model for the processing of the eight most significant bits of an index (i.e., the bits involved in the word unit rotation). The high peaks correspond to the $mask$ computation (if bit $d_i = 1$) or to the computation of its complement $\neg mask$ (where bit $d_i = 0$). These operations are easily recognizable in the simulated power trace since the Hamming weight of each word of the vector which has been rotated is quite low.

Figure 4.1b shows the power consumption related to the processing of the same index but simulating the trace under the assumption of a Hamming distance model. In this case, the peaks occur both for the computation of $mask$ and $\neg mask$ [1] but only when the corresponding bit is different from the previous one ($d_i \neq d_{i-1}$).

In Figure 4.1c, the eight points with the highest SOST values are where the mask computation is performed, or where the $mask$ complement ($\neg mask$) is computed. Since the SOST values are similar in both Hamming weight and Hamming distance model, only the latter is shown below.

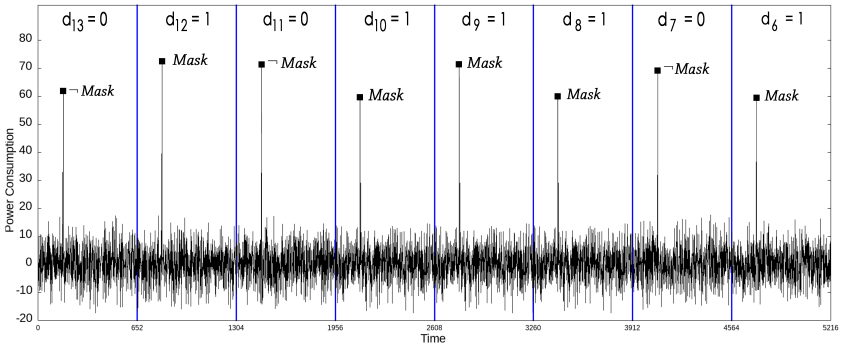
Under both leakage models, with the appropriate changes in the attack (as already explained in Section 3.1), the attack has been tested 1000 times, changing each time the key and the 100 random ciphertexts for the training phase and the key and the ciphertext for the evaluation phase. Both the training phase and the evaluation phase were always successful, leading every time to a full key recover.

4.1.1 Tests on countermeasure

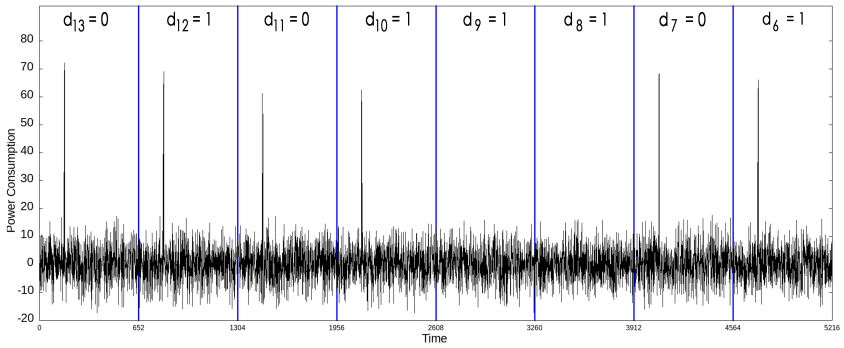
The introduction of a random precharging in the simulation makes it impossible to select meaningful points of interest, since the SOST computation is heavily affected by the countermeasure.

Figure 4.2 shows the SOST values computed over the traces simulated with the random precharging to mask the key-dependent operations. This

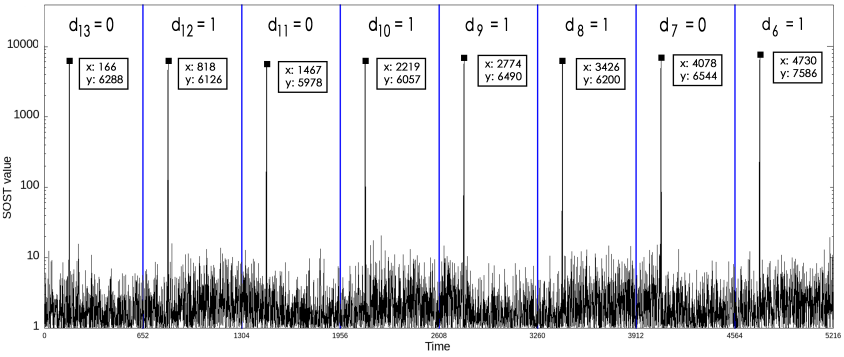
[1] the processing of $mask$ and $\neg mask$ are one right after the other, resulting in two contiguous peaks that can be mistaken for a single one



(a) Simulated power consumption trace, based on the Hamming weight, corresponding to the the word unit rotation of index $d = 5979 = (01011101011011)_2$. The noise added to the power trace is a white noise $\sim\mathcal{N}(0, 6)$.



(b) Simulated power consumption trace, based on the Hamming distance, corresponding to the the word unit rotation of index $d = 5979 = (01011101011011)_2$. The noise added to the power trace is a white noise $\sim\mathcal{N}(0, 6)$.



(c) SOST values computed over the samples from 100 traces and divided into two groups depending on the value of the key bit. The noise added yo the power trace is a white noise $\sim\mathcal{N}(0, 6)$.

Figure 4.1: Simulation of the power traces and SOST values for index $d = 5979 = (01011101011011)_2$.

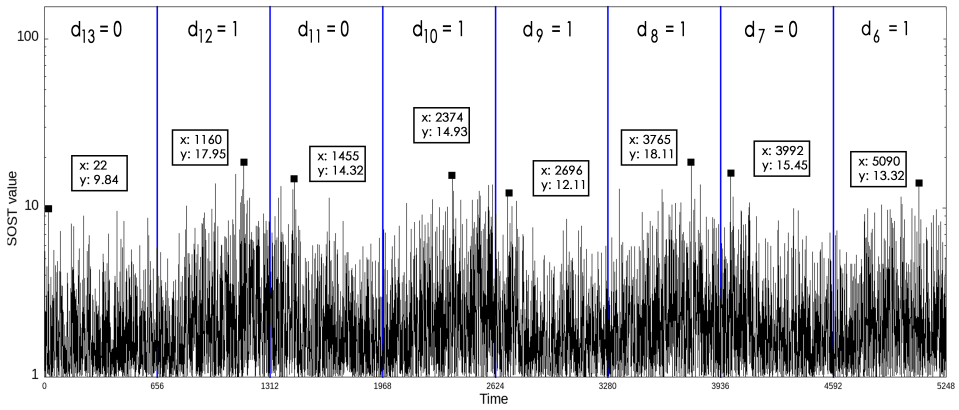


Figure 4.2: SOST values computed over 100 traces simulated with the random precharging on the sensitive operations. The noise added to the power trace is a white noise $\sim\mathcal{N}(0, 6)$.

time, the peaks which highlighted the sensitive operations visible in Figure 4.1c are canceled and the samples corresponding to the highest SOST values are meaningless.

To examine the effectiveness of the countermeasure, we modified the single trace attack in order to take into account the presence of the random precharging. We assumed the attacker to be able to select the samples of the trace which correspond to the operations of random precharging and the *mask* computations. We collected these pairs of samples for each bit of the key involved in the word unit rotation and mounted the single trace attack over the output of a preprocessing phase. As preprocessing function We adopted the *absolute-difference*[2] [29]. The idea is to process the pairs of values {*random precharging, sensitive operation*} in order to cancel the effect of the random precharging to make possible the attack.

Figure 4.3 shows the average number of key bits that have been correctly evaluated by the attack with respect to the standard deviation of the white noise added to the trace. In a simulation environment with zero noise, the attack can successfully classify about 95% of the key bits. However, as soon as a small noise is introduced to simulate the error on the measurements in a real world scenario, the effectiveness of the attack instantly drops and it quickly approaches the threshold of 50% (which means to perform a random guess on the key bits).

[2]i.e., the absolute value of the difference of two points.

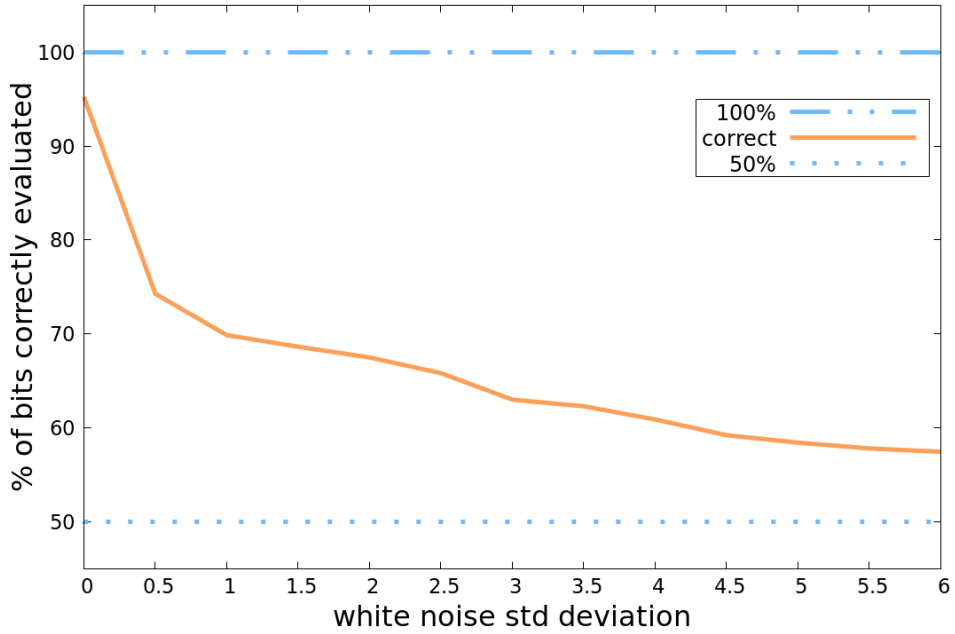


Figure 4.3: Number of key bits correctly evaluated against the random precharging countermeasure. The horizontal axis shows the standard deviation of the white noise introduced in the trace.

In the following, we analyze the effect of the preprocessing function showing its unsuitability the attack against the random precharging.

The different cases can be divided into four on the basis of two consecutive (at time instant $i-1$ and i) values of $mask$, m_{i-1} and m_i:

i. $\mathbf{m_{i-1} = 0x0}$, $\mathbf{m_i = 0xff \ldots ff}$[3].

$$\begin{aligned} |\hat{\mathsf{HD}}_i - \mathsf{HD}_i| &= |\mathsf{HW}(m_{i-1} \oplus r_i) - \mathsf{HW}(r_i \oplus m_i)| \\ &= |\mathsf{HW}(r_i) - (64 - \mathsf{HW}(r_i))| \\ &= |2 \cdot \mathsf{HW}(r_i) - 64| \end{aligned} \tag{4.1}$$

[3]The Hamming weight of 0xff...ff is equal to the word length, which is 64 in the case of the simulation environment.

ii. $\mathbf{m_{i-1} = 0xff \ldots ff}$, $\mathbf{m_i = 0x0}$.

$$\begin{aligned} |\hat{\mathsf{HD}}_i - \mathsf{HD}_i| &= |(\mathsf{HW}(m_{i-1} \oplus r_i) - \mathsf{HW}(r_i \oplus m_i)| \\ &= |(64 - \mathsf{HW}(r_i)) - \mathsf{HW}(r_i)| \\ &= |64 - 2 \cdot \mathsf{HW}(r_i)| \\ &= |2 \cdot \mathsf{HW}(r_i) - 64| \end{aligned} \tag{4.2}$$

iii. $\mathbf{m_{i-1} = 0x0}$, $\mathbf{m_i = 0x0}$.

$$\begin{aligned} |\hat{\mathsf{HD}}_i - \mathsf{HD}_i| &= |\mathsf{HW}(m_{i-1} \oplus r_i) - \mathsf{HW}(r_i \oplus m_i)| \\ &= |\mathsf{HW}(r_i) - \mathsf{HW}(r_i)| \\ &= 0 \end{aligned} \tag{4.3}$$

iv. $\mathbf{m_{i-1} = 0xff \ldots ff}$, $\mathbf{m_i = 0xff \ldots ff}$.

$$\begin{aligned} |\hat{\mathsf{HD}}_i - \mathsf{HD}_i| &= |\mathsf{HW}(m_{i-1} \oplus r_i) - \mathsf{HW}(r_i \oplus m_i)| \\ &= |(64 - \hat{\mathsf{HD}}(r_i)) - (64 - \mathsf{HW}(r_i))| \\ &= 0 \end{aligned} \tag{4.4}$$

$\hat{\mathsf{HD}}_i$ denotes the Hamming distance between the previous value of *mask* (m_{i-1}) and the value of the random precharging at time i (r_i). HD_i denotes the Hamming distance between the value of the random precharging at time i (r_i) and the current (time i) value of *mask* (m_1). These two values are the samples taken from the trace (simulated with the Hamming distance) and give to the preprocessing function. $\mathsf{HW}(x)$ denotes the Hamming weight of x.

Since Eq. 4.1 and 4.2 lead to the same result, and the same holds between Eq. 4.3 and 4.4, the absolute-difference seems to correctly characterize the cases in which $m_{i-1} = m_i$ and the cases in which $m_{i-1} \neq m_i$, i.e. the cases in which the Hamming distance between two consecutive values of *mask* is low or high.

Table 4.1 summarizes the results for different preprocessing functions.

However, since the values of the random precharging are uniformly dis-

Table 4.1: Results of the most commonly used preprocessing functions

	Value			
$\mathbf{m_{i-1}}$	0	0	64	64
$\mathbf{m_i}$	0	64	0	64
$\mathsf{HD}(\mathbf{m_{i-1}}, \mathbf{m_i})$	0	64	64	0
$\lvert\hat{\mathsf{HD}}_\mathbf{i} - \mathsf{HD}_\mathbf{i}\rvert$	0	$\lvert 2 \cdot \mathsf{HW}(r_i) - 64\rvert$	$\lvert 2 \cdot \mathsf{HW}(r_i) - 64\rvert$	0
$(\hat{\mathsf{HD}}_\mathbf{i} - \mathsf{HD}_\mathbf{i})^\mathbf{2}$	0	$(\lvert 2 \cdot \mathsf{HW}(r_i) - 64\rvert)^2$	$(\lvert 2 \cdot \mathsf{HW}(r_i) - 64\rvert)^2$	0
$\hat{\mathsf{HD}}_\mathbf{i} \times \mathsf{HD}_\mathbf{i}$	$(\mathsf{HW}(r_i))^2$	$Hw(r_i) \times (64 - \mathsf{HW}(r_i))$	$(64 - \mathsf{HW}(r_i)) \times \mathsf{HW}(r_i)$	$(64 - \mathsf{HW}(r_i))^2$

tributed, the expected value of $\mathsf{HW}(r_i)$ is given by:

$$\begin{aligned}\mathbf{E}[\mathsf{HW}(r_i)] &= \frac{0 \cdot \binom{64}{0} + 1 \cdot \binom{64}{1} + \ldots + 64 \cdot \binom{64}{64}}{2^{64}} \\ &= \frac{\sum_{b=0}^{64} b \cdot \binom{64}{b}}{2^{64}} \\ &= \frac{64}{2} \end{aligned} \tag{4.5}$$

This means that the expected value of Eq.4.1 and 4.2 is equal to

$$\begin{aligned}\mathbf{E}[\lvert 2 \cdot \mathsf{HW}(r_i) - 64\rvert] &= \lvert 2 \cdot \mathbf{E}[\mathsf{HW}(r_i)] - 64\rvert \\ &= \lvert 2 \cdot \frac{64}{2} - 64\rvert \\ &= 0 \end{aligned} \tag{4.6}$$

and that the preprocessing phase is not really able to distinguish the different cases.

The same (or equivalent) result is obtained with other commonly used preprocessing functions[4] (see Table 4.1), preventing other possible attacks in this direction.

4.2 Experiments on real Board

For the experiments on real board, the attack has been implemented and tested on a `Cortex-M7` (32-bit architecture) mounted on a `STM32F746ZG` board. All the experiments have been carried out at a clock frequency of

[4]including, but not limited to, the product of the values, the square of the difference, the cube of the difference.

Figure 4.4: The experimental board, STM32F746ZG.

Figure 4.5: The custom loop probe used for the measurements.

216 MHz, with the adaptive real time (ART) accelerator disabled. Also the interrupt requests have been disabled during the computation of the algorithm to limit as much as possible their interference on the trace.

Electromagnetic radiations measurements have been acquired with the digital oscilloscope `Pico Technology PicoScope 5244D` at a sampling rate of 500 MS/s and a 12-bit vertical resolution. The near field probe adopted was a custom loop probe (an electric field probe) and it was positioned over a microcontroller power supply line, near to the chip.

During the experiments, we employed also an impedance adapter ($50\Omega \rightarrow 1M\Omega$) and a 2-stage cascaded amplifier circuit (consisting of two `Agilent INA-10386` each of gain 26 dB, obtaining a total gain of 52 dB) to correct and enhance the signal acquisition.

Communication with the board has been enabled by the built-in serial interface.

Figure 4.6 shows the acquired trace for a constant-time multiplication performed over every index of the matrix to compute a complete sparse-to-dense multiplication. For this and the following figures, the square of the acquired voltage trace has been plotted to present a qualitative behavior for the power consumption. 142 repetitions of the same pattern (separated by a

high and a low peak), corresponding to a single constant-time multiplication (i.e., the processing of a single index of the key), can be easily recognized.

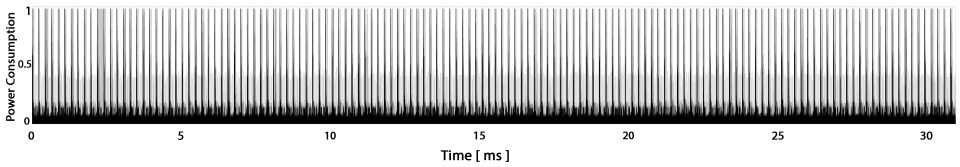


Figure 4.6: Power consumption trace of a constant-time multiplication performed on every index of the secret key.

According to the parameters presented in Table 3.2, the number of bits, for each index, involved during the word unit rotation is equal to 9, which are the bits that the single trace attack aims to recover.

An example of a constant-time multiplication is shown in Figure 4.7. These samples correspond to a single constant-time multiplication (Algorithm 2.4.3) which performs a circular shift over a vector by a shift amount given by an index of the key matrix. It is possible to recognize the *word unit rotation* and the *bit rotation* as well as the two for loops inside the *word unit rotation*. The two parts to the left and right of the peak in the magnification, correspond to the blocks of *nop* operations introduced to highlight the target operation. The target operation, which produces the peak in the center of the magnification, is the operation at line 5 of Algorithm 2.4.3, where the value of *mask* is computed.

The single trace attack, performed over a real trace acquired from the device electromagnetic leakage, exposed a leakage model following the Hamming weight principle.

The acquired single trace was really noisy and the attack didn't obtained the neat results obtained in the simulation environment. Due to the noise, the attack has not been able to evaluate 14% of the bits (which has been classified as "unknown") and a significant part of the others produced false positives. However, these results are enough to highlight the leakage model. In particular, over different thresholds, the Hamming weight model fits the leakage model at best at 65%, while the Hamming distance model is able to describe the leakage model ranging between 48 and 51%. According to these results, the Hamming distance seems to act as a series of random guesses on the key bits, and therefor it can not be used to describe the leakage model

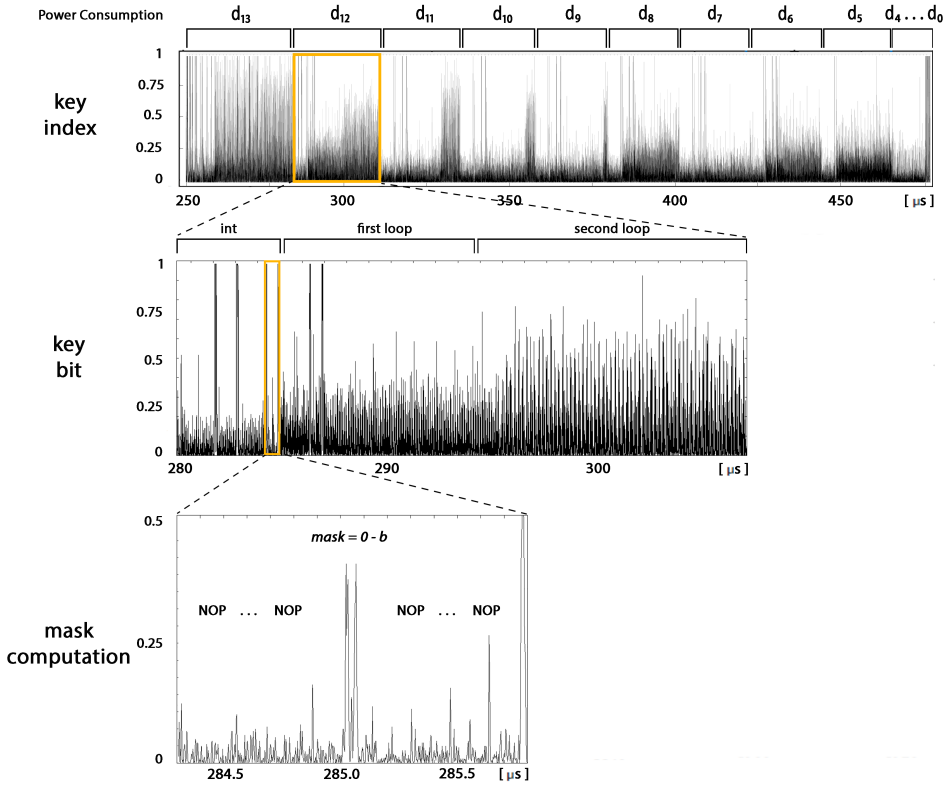


Figure 4.7: Power consumption trace of a constant-time multiplication. From the top, the processing of a single index of the key, along with a magnification of the processing of a single bit of the index and, finally, the *mask* computation (refer to Algorithm 2.4.3).

of the code under examination.

Figure 4.8 shows the percentages of recovered bits, along with the false positives, employing the Hamming distance or Hamming weight model with different thresholds (refer to Figure 4.9 for the percentage values computed without consider the "unknown" bits). It is easy to notice that the Hamming distance model is not affected by the value of the threshold and it always correctly identify about the half of the key bits. On the other hand, setting the right threshold, the Hamming weight model can reach better results.

Figure 4.10 and 4.11 show the number of recovered key bits (in percentage), under one leakage model or the other, distinguishing between bit set to 0 and bit set to 1.

To confirm the previous results, we measured several times the leakage in

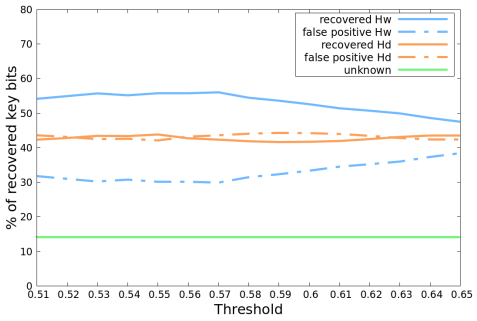


Figure 4.8: Percentages of recovered key bits (solid line) and false positives (dashed line) under the Hamming weight (blue) and Hamming distance (orange) leakage models, over different threshold values. The green line indicates the "unknown" bits.

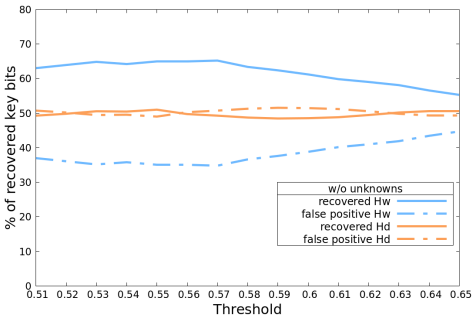


Figure 4.9: Percentages (computed without considering "unknown" bits) of recovered key bits under the Hamming weight (blue) and Hamming distance (orange) leakage models, along with the false positives (dashed lines), over different threshold values.

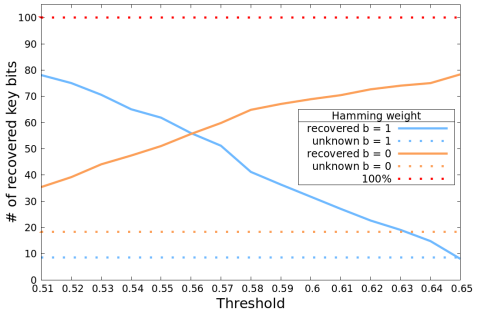


Figure 4.10: Percentages of recovered key bits set as 0 (orange) or 1 (blue) under the Hamming weight model, over different threshold values.

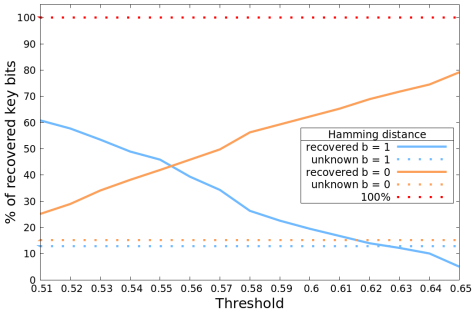
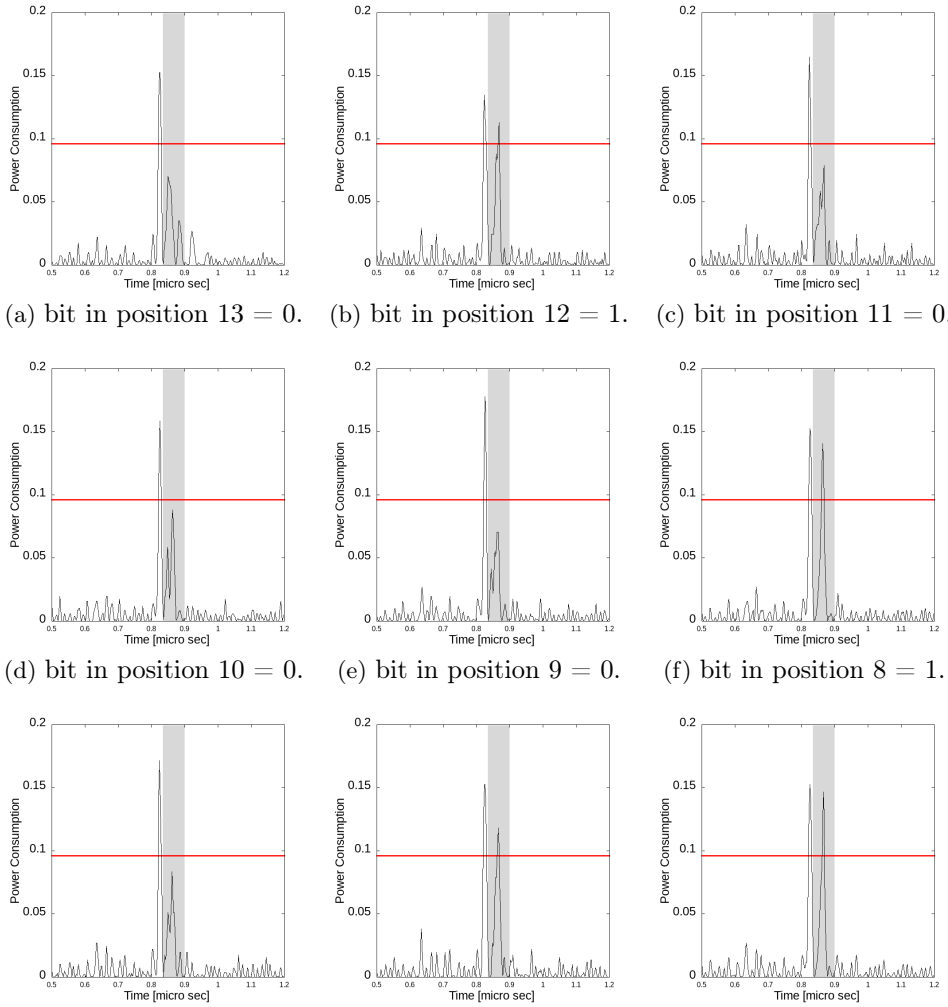


Figure 4.11: Percentages of recovered key bits set as 0 (orange) or 1 (blue) under the Hamming distance model, over different threshold values.

correspondence of the target operation for the processing of different indices and we examined the average value of the traces for each of them. The average of 16 traces was enough to significantly reduce the noise leading to more precise results. Under these circumstances 89% of the key bits has been correctly identified by the Hamming weight model, while the Hamming distance, once again, was able to correctly identify 51% of the key bits.

Figure 4.12 shows the leakage of the target operation surrounded by several nop operations on the left and right side. Each sub-figure represents the leakage for each one of the nine bits involved in the word unit rotation for



(a) bit in position 13 = 0. (b) bit in position 12 = 1. (c) bit in position 11 = 0.

(d) bit in position 10 = 0. (e) bit in position 9 = 0. (f) bit in position 8 = 1.

(g) bit in position 7 = 0. (h) bit in position 6 = 1. (i) bit in position 5 = 1.

Figure 4.12: Intervals of the trace corresponding to the processing of each bit involved in the word unit rotation of index $d = 4448 = (01000101100000)_2$

index $d = 4448 = (01000101100000)_2$. Sub-figure 4.12a shows the leakage in correspondence of the MSB (bit in position 13, first bit from the left) while Sub-figure 4.12i shows the ninth bit from the left of the binary representation (bit in position 5). Looking at peak in the grey box in these figures,it is clear how the leakage can be well described by the Hamming weight model. Each peak that exceeds the threshold corresponds to a key bit set to 1, while each peak that stays below the threshold corresponds to a key bit set to 0.

The Hamming weight leakage model is also well observable on the first

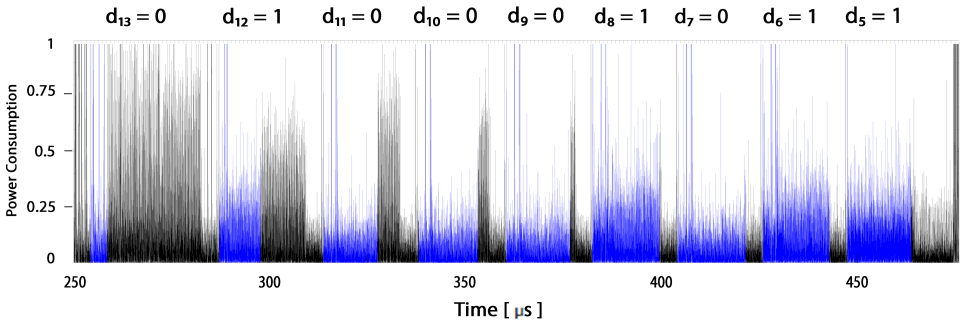


Figure 4.13: Leakage trace corresponding to the processing of index $d = 4448 = (01000101100000)_2$.

`for loop` of the word unit rotation (lines 8, 9 in Algorithm 2.4.3). Figure 4.13 shows the interval of the trace corresponding to index $d = 4448$. The first for loop of the word unit rotation has been highlighted in blue for each bit. It is possible to notice that the samples corresponding to key bits set to 1 reach a higher power consumption.

We investigated also the traces captured with the random precharging enabled on the sensitive operations.

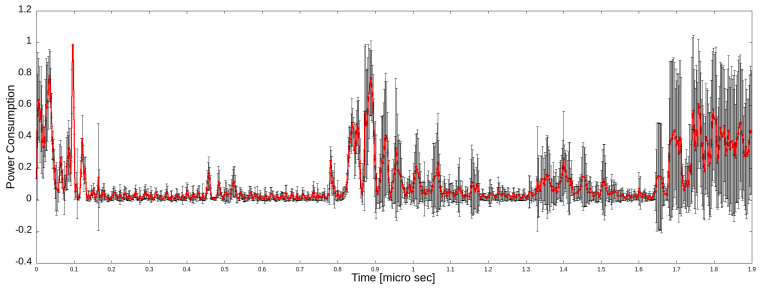
These traces ended up being noisier than the others with the plain multiplication, and it was not possible to correctly distinguish the value of the key bits. However, it was found that this supplementary noise was introduced by the background operations of the TRNG module of the board. In order to see this behavior, We examined the execution of four different versions of the code:

i. With the random precharging given by the RNG peripheral keeping its clock enabled throughout the entire run of the algorithm.

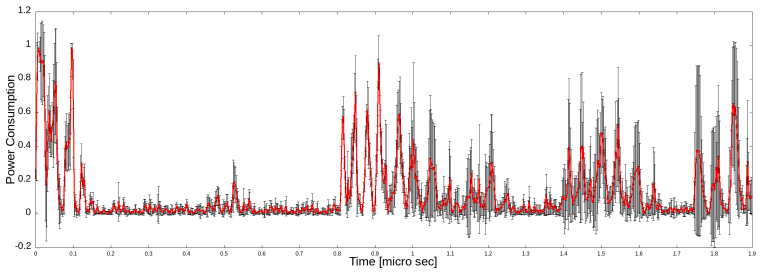
ii. In order to reduce the extra noise introduced by the background operations of the RNG, we captured several traces with the RNG peripheral enabled but disabling its clock after each call to the module.

iii. Since the second setup did not completely removed the noise carried by the RNG, we disabled the peripheral and simulated the random precharging with a constant value. For the constant precharging, we employed an arbitrary 32-bit value with a Hamming weight equal to 16, simulating an average value for a 32-bit true random number.

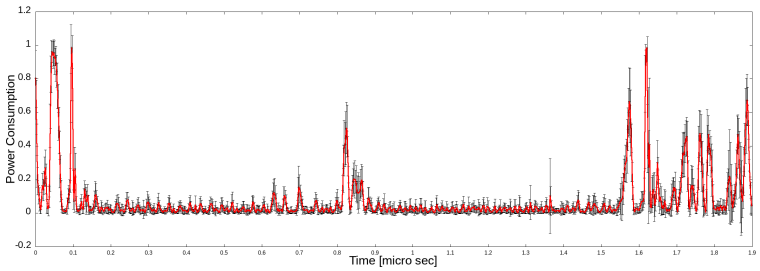
iv. Finally, as reference model, the same traces has been captured on the plain multiplication, without any kind of precharging.



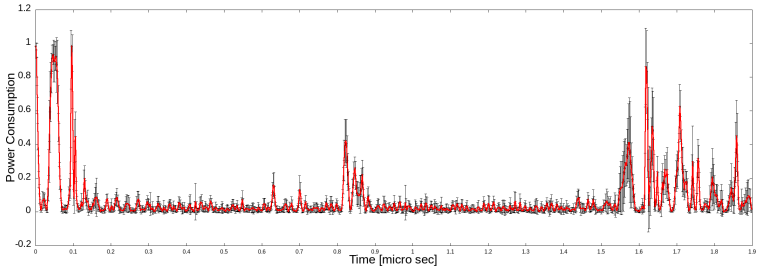
(a) The RNG peripheral is enabled along with its clock.



(b) The RNG peripheral is enabled, but its clock is disabled after each call to the RNG.



(c) The RNG peripheral is disabled and precharging is done with a constant value.



(d) Plain multiplication without any countermeasure enabled.

Figure 4.14: Average value (in red) of traces captured for a single bit processing, along with +/− standard deviation (in black). Each sub-figure describe a different scenario.

As shown in the pictures collected in Figure 4.14, a significant noise was introduced by the use of the RNG peripheral, while the precharging operation it self did not tampered the leakage at all.

These results are consistent with respect to the Hamming weight leakage model since the random precharging is able to mask only the Hamming distance between two values but not the Hamming weight of a single value.

After the validation of the Hamming weight model, the next step was to understand way it was the right model and why the Hamming distance was not.

Examining the assembly code generated by the compiler, it was easy to recognize the reason behind the Hamming weight leakage.

The following assembly code is the translation of line 5 in Algorithm 2.4.3.

```
ldr     r1, [sp, #36]    ; 0x24
negs    r1, r1
str     r1, [sp, #28]
```

The first line, loads the value of d_i (located at `[sp, #36]`). The second line saves in `r1` the result of $d_i - 1$. Finally, the third line, stores the result in *mask* (located at `[sp, #28]`).

Notice that d_i is 0 when the key bit is 0 and 1 when the key bit is 1. The `negs` instruction overwrites the value of d_i with the result of $0 - d_i$ which gives a word of 32 bits set to 1 (when the key bit is 1) or a word with 32 bit set to 0 (when the key bit is 0). Therefor, the `negs` instruction produces a switching activity in `r1` equal to 31 switches if the key bit is set to 1 and 0 switches if the key bit is set to 0. This behavior is perfectly modeled by the Hamming weight of the value of *mask*.

4.2.1 Countermeasure validation

Since the Hamming weight leakage model is validated by the re-usage of a register, the idea of using random precharging to secure the algorithm against power analysis is still valid. The problem with the implementation

was the limitation given by the only usage of the *volatile* keyword aiming at performing random precharging from a variable abstraction level that has proved to be not enough to ensure the expected results.

For the random precharging to be effective, it has to be done at register level. The following assembly listing shows an example for the application of the precharging with constant values to secure line 5 of Algorithm 2.4.3. Similar adjustments have been introduced also to secure the rest of the constant time multiplication.

```
movw    r1, #51526        // precharging r1
movt    r1, #46489        // precharging r1
ldr     r1, [sp, #36]     // loading d_i
movw    r4, #47764        // precharging r4
movt    r4, #52049        // precharging r4
negs    r4, r1            // saving 0 - d_i in r4
movw    r1, #45786
movt    r1, #38042
str     r1, [sp, #28]     // precharging mask
str     r4, [sp, #28]     // storing 'mask' in memory
```

`r1` is loaded with an arbitrary 32-bit value before loading d_i. Also `r4` is loaded with a arbitrary 32-bit value before receiving the result of $0 - d_i$, as well as the memory location of *mask* before storing the new value in it.

In Figure 4.15 are shown the results of the precharging at register level. The peaks corresponding to the nine bits of the index have been placed next to each other for easy comparison between them. The values on the top represents the actual value of the key bit. Since the leakage is the same for each mask computation, it is no longer possible to correctly guess the key bits involved in the operations.

Figure 4.16 shows how the leakage of the first loop in the multiplication algorithm is affected by the countermeasure. Differently from what is shown in Figure 4.13 (where the countermeasure is not employed), in this case, the samples corresponding to the first `for loop` of the word unit rotation (highlighted in blue) have similar values for each key bit, masking the leakage

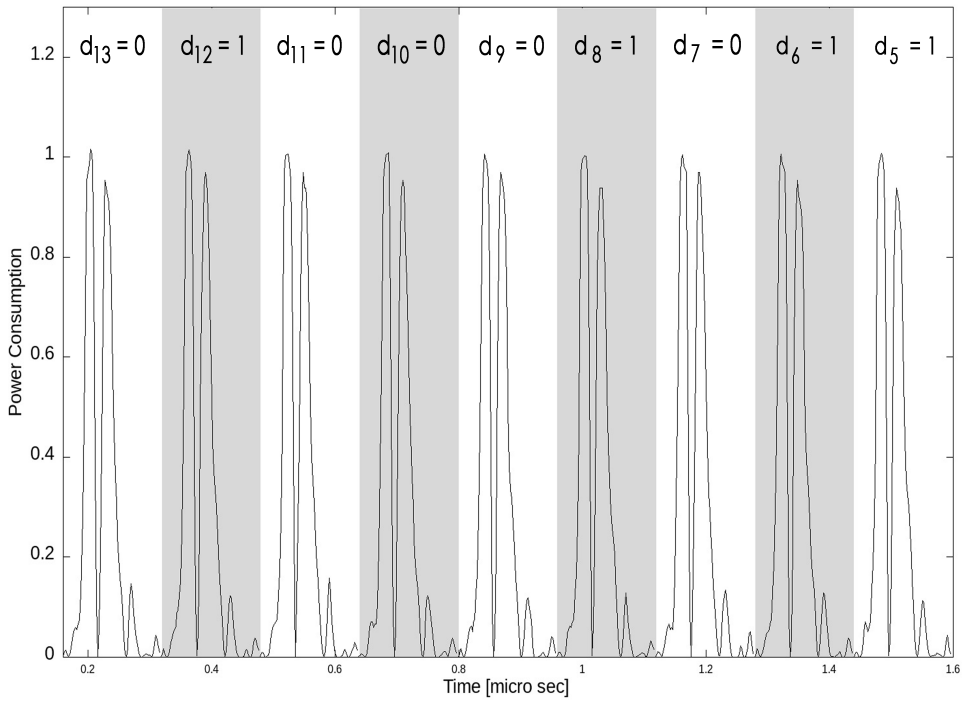


Figure 4.15: Leakage peaks corresponding to the *mask* computation for the nine bits in index $d = 4448 = (01000101100000)_2$ captured while performing register precharging.

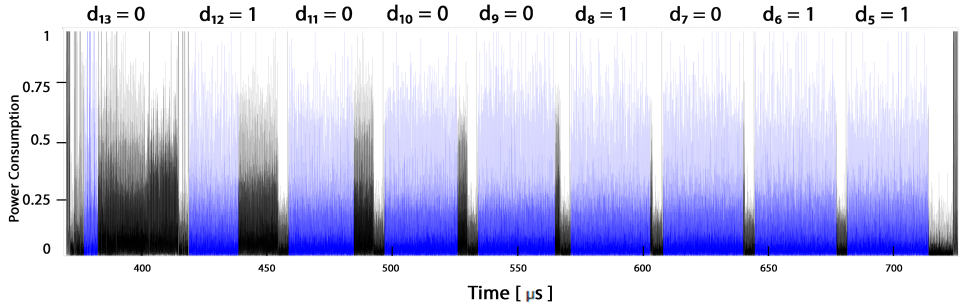


Figure 4.16: Leakage trace corresponding to the processing of index $d = 4448 = (01000101100000)_2$ captured while performing register precharging.

of the key-dependent operations.

All the practical experiments have been carried out on a `Cortex-M7` running at 216 MHz. A complete sparse-to-dense multiplication employing the constant-time multiplication algorithm (Algorithm 2.4.3) and the parameters given in Table 3.2 takes, on average, 6950120 cycles. The same operation, with the same parameters, employing the presented countermeasure, takes

11425031 cycles, plus the time spent in the random values generation (which depends on the RNG latency and the possibility of parallelization of this process with respect to the multiplication). These values correspond to a time overhead of 64%.

Conclusion

In this document we investigated the state-of-the-art power analysis attacks mounted against cryptosystems based on QC-MDPC/LDPC codes. Particular attention has been given to the ones which could find a possible attack surface in LEDAcrypt.

The single trace attack described by Sim et al. (2019, [46]) was found to be the most promising one. The target of this attack corresponds to the constant-time multiplication proposed by Tung Chou as the state-of-the-art countermeasure against timing attacks, and employed to perform a sparse-to-dense multiplication with the secret key, during the decryption phase. Moreover, they explicitly suggested the feasibility of this attack against LEDAcrypt without producing any countermeasure. For the other attacks which have been taken into account (see [13,42]), the authors themselves proposed valid countermeasures in [15, 42]. We thus proceeded in the analysis of the single trace attack.

First, we implemented a simulation framework to reason about the leakage model end evaluate the theoretical feasibility of the attack. On the results observed in the simulation, we designed a countermeasure in order to secure the algorithm. Then, we moved on a real device and tested both the attack and the countermeasure on leakage traces acquired with a digital oscilloscope and a custom loop probe. The results of the analysis are summarized below.

The proposed single trace attack was designed modeling the leakage with the Hamming weight, although the Hamming distance is usually a better fit for real case scenarios. In the simulation environment, we tested the two leakage models and proved the theoretical applicability of the attack based on both. We then designed a countermeasure based on random precharging,

aiming at securing the algorithm against the attack under the hypothesis of the more likely Hamming distance model. What we saw from the analysis of the traces acquired from the device, was that the Hamming weight was actually able to describe the leakage of the algorithm while the Hamming distance was not. In fact, the attack mounted against the `Cortex-M7` was able to recover most of the key bits assuming a Hamming weight model. However, it was found that the leakage behavior was given by the usage of the registers dictated by the compiler, hence, the random precharging was still a valuable countermeasure to invalidate the attack. In order for the random precharging to be effective, we had to redesign the countermeasure introducing a segment written in assembly to directly control the use of the registers. Implementing the random precharging at register level, gave the expected results: masking the leakage of a key-dependent operation and preventing the single trace attack.

In conclusion, the single trace attack proposed by Sim et al., although the acquired single traces was too noisy to allow a full key recover, seems to be a reasonable threat and it must be taken into account if we want to secure the constant-time-multiplication algorithm. At the same time, we proved that a random precharging, carefully implemented, can be considered as a valid countermeasure against such attack.

For practical reasons, the training phase of the attack has not been implemented for the evaluation of the acquired traces. It followed that we focused on the study of a single target operation (pointed out, by Sim et al., to be the most critical one, but not the only). Although visible improvements have been obtained over the whole trace, a future work may further investigate the soundness of this countermeasure starting from the SOST computation to see if there are still key-dependent operations which leak sensitive information. Moreover, it would be interesting to develop a more extensive study on the overhead introduced by the proposed countermeasure.

Appendix A

Constant-Time Multiplication (T. Chou convention)

The Algorithm A.0.1 is a detailed scheme for Algorithm 2.3.1.

Algorithm A.0.1: Constant-Time Multiplication in $\mathbb{F}_2[\mathrm{x}]/\langle x^r - 1\rangle$ (refer to [16])

Input : $d = (d_{l-1}, \ldots, d_0)_2$, $0 \le d \le r-1$,
$c_{(k)} = (c_{L-1}, \ldots, c_0)_{2^W}$, $L = \lceil r/W \rceil$
Output: $x^d c_{(k)}$

1 $v \leftarrow 0$, $w \leftarrow c_{(k)}$, $tail \leftarrow r \bmod W$
2 **for** $i = l-1$ down to $\log_2 W$ **do** ▶ **word unit rotation** lines 2 to 14
3 $d_i \leftarrow (d \gg (l-1-i)) \,\&\, 1$
4 $mask \leftarrow 0 - d_i$
5 $us \leftarrow 1 \ll (i - \log_2 W)$
6 $ptr \leftarrow v$, $v \leftarrow w$, $w \leftarrow ptr$
7 **for** $j = 0$ up to $L - 1 - us - 1$ **do**
8 $w[j] \leftarrow (v[j + us] \,\&\, mask) \oplus (v[j] \,\&\, \neg mask)$
9 $w[L-1-us] \leftarrow ((v[L-1] \mid (v[0] \ll tail)) \,\&\, mask)$
10 $\oplus (v[L-1-us] \,\&\, \neg mask)$
11 **for** $j = 1$ up to $us - 1$ **do**
12 $w[j+L-1-us] \leftarrow (((v[j] \ll tail) \mid (v[j-1] \gg (W - tail)))$
13 $\&\, mask) \oplus (v[j+L-1-us] \,\&\, \neg mask)$
14 $w[L-1] \leftarrow ((v[us-1] \gg (W-tail)) \,\&\, mask) \oplus (v[L-1] \,\&\, \neg mask)$

```
15 low ← d & ((1 ≪ log2 W) − 1)        ▶ bit rotation lines 15 to 27;
16 mask ← ((low − 1) ≫ (W − 1)) − 1;
17 high ← W − low;
18 tmp ← w[0];
19 for j = 0 up to L − 3 do
20     w[j] ← w[j] ≫ low;
21     w[j] ← w[j] | ((w[j + 1] ≪ high) & mask);
22 w[L − 2] ← w[L − 2] ≫ low;
23 w[L − 1] ← w[L − 1] | (tmp ≪ tail);
24 w[L − 2] ← w[L − 2] | ((w[L − 1] ≪ high) & mask);
25 w[L − 1] ← w[L − 1] ≫ low;
26 w[L − 1] ← w[L − 1] | ((temp ≪ high) & mask);
27 w[L − 1] ← w[L − 1] & ((1 ≪ tail) − 1);
28 return w;
```

Toy example for the case $r = 40$, $W = 8$, a vector $c_{(k)} = (c_0, c_1, \cdots, c_{39})$ $\in \mathbb{F}_2^{40}$ can be represented as the polynomial $c_{(k)} = c_0 + c_1x + c_2x^2 + \cdots + c_{39}x^{39}$ $\in \mathbb{F}_2[\mathrm{x}]/\langle x^{40} - 1\rangle$. Let the polynomial $c_{(k)}$ be

$$(x^8 + x^{10} + x^{12} + x^{14}) + (x^{16} + x^{17} + x^{20} + x^{21}) + (x^{24} + x^{25} + x^{26} + x^{27}) + (x^{36} + x^{37} + x^{38} + x^{39}),$$

which can be expressed as a 5-byte array as below:

v[0]	v[1]	v[2]	v[3]	v[4]
$(00001111)_2$	$(11110000)_2$	$(11001100)_2$	$(10101010)_2$	$(00000000)_2$

Following the convention, in the first cell ($v[0]$) are stored the 8 most significant bits. Inside the word, the bits are represented (from left to right) starting from the bit with the lowest weight. In other words, $v[0]$ contains the terms $(x^{32} + x^{33} + x^{34} + x^{35} + x^{36} + x^{37} + x^{38} + x^{39})$.

Let $d = 19$; then it is represented by 6-bit $(010011)_2$. Since $W = 8$ and $\log_2 W = 3$, it is possible to calculate the rotated intermediate values using 8-bit word unit rotation for d_i from d_5 to d_3. For the last 3-bit,

$(d_2, d_1, d_0)_2 = (011)_2$, a sequence of logical instructions is used, combining the most significant 5-bit of $v[i]$ and the least significant 3-bit of $v[(i+1) \bmod l]$. Accordingly, the multiplication $x^d = x^{(010011)_2} = x^{0 \cdot 2^5} \cdot x^{1 \cdot 2^4} \cdot x^{0 \cdot 2^3} \cdot x^{(011)_2}$ and $c_{(k)}$ is given by:

$$c_{(k)} \cdot (x^{0 \cdot 2^5} \cdot x^{1 \cdot 2^4} \cdot x^{0 \cdot 2^3} \cdot x^{(011)_2}) = ((((c_{(k)} \cdot x^{0 \cdot 2^5}) \cdot x^{1 \cdot 2^4}) \cdot x^{0 \cdot 2^3}) \cdot x^{(011)_2}).$$

Firstly, the computation is started from the multiplication with x^{2^5} which can be acquired by a 4-byte left rotation. However, the d_5 is 0, so the unrotated value is saved.

	v[0]	v[1]	v[2]	v[3]	v[4]
unrotated	$(00001111)_2$	$(11110000)_2$	$(11001100)_2$	$(10101010)_2$	$(00000000)_2$
rotated	$(00000000)_2$	$(00001111)_2$	$(11110000)_2$	$(11001100)_2$	$(10101010)_2$

Secondly, the multiplication with x^{2^4} can be acquired by a 2-byte left rotation. Since the d_4 is 1, the rotated value is saved.

	v[0]	v[1]	v[2]	v[3]	v[4]
unrotated	$(00001111)_2$	$(11110000)_2$	$(11001100)_2$	$(10101010)_2$	$(00000000)_2$
rotated	$(11001100)_2$	$(10101010)_2$	$(00000000)_2$	$(00001111)_2$	$(11110000)_2$

Thirdly, the multiplication with x^{2^3} can be acquired by a 1-byte left rotation. However, the d_3 is 0, so the unrotated value is saved.

	v[0]	v[1]	v[2]	v[3]	v[4]
unrotated	$(11001100)_2$	$(10101010)_2$	$(00000000)_2$	$(00001111)_2$	$(11110000)_2$
rotated	$(10101010)_2$	$(00000000)_2$	$(00001111)_2$	$(11110000)_2$	$(11001100)_2$

Lastly, the multiplication with $x^{(011)_2}$ can be acquired by the sequence of logical instructions which combines the most significant 5-bit of $v[i]$ and the least significant 3-bit of $v[(i+1) \bmod l]$.

	v[0]	v[1]	v[2]	v[3]	v[4]
unrotated	$(01011001)_2$	$(00010101)_2$	$(11100000)_2$	$(00000001)_2$	$(10011110)_2$

Bibliography

[1] Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillipe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, Gilles Zemor, and Valentin Vasseur. Bike (bit flipping key encapsulation).

[2] Aydin Aysu, Youssef Tobah, Mohit Tiwari, Andreas Gerstlauer, and Michael Orshansky. Horizontal side-channel vulnerabilities of post-quantum key exchange protocols. pages 81–88, 04 2018.

[3] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. Ledacrypt-kem and ledacrypt-pkc website. `https://www.ledacrypt.org`.

[4] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini. Ledakem: A post-quantum key encapsulation mechanism based on qc-ldpc codes. In Tanja Lange and Rainer Steinwandt, editors, *Post-Quantum Cryptography*, pages 3–24, Cham, 2018. Springer International Publishing.

[5] E. Berlekamp, R. McEliece, and H. Tilborg. On the inherent intractability of certain coding problems (corresp.). *Information Theory, IEEE Transactions on*, 24:384 – 386, 06 1978.

[6] Daniel J. Bernstein. The poly1305-aes message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, pages 32–49, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[7] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: Fast constant-time code-based cryptography. In Guido Bertoni and Jean-

Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 250–272, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[8] Daniel J. Bernstein and Peter Schwabe. New aes software speed records. In Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das, editors, *Progress in Cryptology - INDOCRYPT 2008*, pages 322–336, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[9] Bhaskar Biswas and Nicolas Sendrier. The hybrid mceliece encryption scheme (hymes). 2008.

[10] Dominic Bucerzan, Pierre-Louis Cayrel, Vlad Dragoi, and Tania Richmond. Improved timing attacks against the secret permutation in the mceliece pkc. *International Journal of Computers Communications & Control*, 12(1):7–25, 2016.

[11] P. Cayrel and P. Dusart. Mceliece/niederreiter pkc: Sensitivity to fault injection. In *2010 5th International Conference on Future Information Technology*, pages 1–6, May 2010.

[12] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In Burton S. Kaliski, çetin K. Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[13] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt. Horizontal and vertical side channel analysis of a mceliece cryptosystem. *IEEE Transactions on Information Forensics and Security*, 11(6):1093–1105, June 2016.

[14] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Differential power analysis of a mceliece cryptosystem. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *Applied Cryptography and Network Security*, pages 538–556, Cham, 2015. Springer International Publishing.

[15] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt. Masking large keys in hardware: A masked implementation of

mceliece. In Orr Dunkelman and Liam Keliher, editors, *Selected Areas in Cryptography – SAC 2015*, pages 293–309, Cham, 2016. Springer International Publishing.

[16] Tung Chou. Qcbits: Constant-time small-key code-based cryptography. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems – CHES 2016*, pages 280–300, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[17] Tung Chou. Mcbits revisited. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 213–231, Cham, 2017. Springer International Publishing.

[18] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 400:117 – 97, 1985.

[19] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[20] Tomáš Fabšič, Ondrej Gallo, and Viliam Hromada. Simple power analysis attack on the QC-LDPC McEliece cryptosystem. *Tatra Mt. Math. Publ.*, 67:85–92, 2016.

[21] Richard P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21:467 – 488, 1982.

[22] R. Gallager. Low-density parity-check codes. *IRE Transactions on Information Theory*, 8(1):21–28, January 1962.

[23] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 15–29, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[24] Qian Guo, Thomas Johansson, and Paul Stankovski. A key recovery attack on mdpc with cca security using decoding errors. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 789–815, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[25] Stefan Heyse, Amir Moradi, and Christof Paar. Practical power analysis attacks on software implementations of mceliece. In Nicolas Sendrier, editor, *Post-Quantum Cryptography*, pages 108–125, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] Stefan Heyse, Ingo von Maurich, and Tim Güneysu. Smaller keys for code-based cryptography: Qc-mdpc mceliece implementations on embedded devices. In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, pages 273–292, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[27] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[28] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, pages 104–113, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[29] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer-Verlag, Berlin, Heidelberg, 2007.

[30] Robert J. McEliece. A public key cryptosystem based on algebraic coding theory. 1978.

[31] R. Misoczki, J. Tillich, N. Sendrier, and P. S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *2013 IEEE International Symposium on Information Theory*, pages 2069–2073, July 2013.

[32] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo S. L. M. Barreto. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In *IACR Cryptology ePrint Archive, Report 2012/409*, 2012.

[33] H. Gregor Molter, Marc Stöttinger, Abdulhadi Shoufan, and Falko Strenzke. A simple power analysis attack on a mceliece cryptoprocessor. *Journal of Cryptographic Engineering*, 1(1):29–36, Apr 2011.

[34] H. Niederreiter. Knapsack-type cryptosystem based on algebraic coding theory. *Problems of Control and Information Theory*, 15(2):157–166, 1986.

[35] Takuji Nishimura and Makoto Matsumoto. Mersenne twister 64bit version, 2004. `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt64.html`.

[36] NIST. Post-quantum cryptography, round 2 submissions, nist computer security resource center. 2019. `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions`.

[37] Colin O'Flynn and Zhizhang (David) Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In Emmanuel Prouff, editor, *Constructive Side-Channel Analysis and Secure Design*, pages 243–260, Cham, 2014. Springer International Publishing.

[38] Michaël Peeters, Gilles Van Assche, Guido Bertoni, and Joan Daemen. Keccak and the sha-3 standardization. 2013.

[39] Edoardo Persichetti. Secure and anonymous hybrid encryption from coding theory. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, pages 174–187, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[40] M. Petrvalsky, T. Richmond, M. Drutarovsky, P. Cayrel, and V. Fischer. Countermeasure against the spa attack on an embedded mceliece cryptosystem. In *2015 25th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 462–466, April 2015.

[41] M. Petrvalsky, T. Richmond, M. Drutarovsky, P. Cayrel, and V. Fischer. Differential power analysis attack on the secure bit permutation in the mceliece cryptosystem. In *2016 26th International Conference Radioelektronika (RADIOELEKTRONIKA)*, pages 132–137, April 2016.

[42] Mélissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson. A side-channel assisted cryptanalytic attack against qcbits. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 3–23, Cham, 2017. Springer International Publishing.

[43] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, Oct 1949.

[44] P. W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, Nov 1994.

[45] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger. A timing attack against patterson algorithm in the mceliece pkc. In Donghoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology – ICISC 2009*, pages 161–175, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[46] Bo-Yeon Sim, Jihoon Kwon, Kyu Young Choi, Jihoon Cho, Aesun Park, and Dong-Guk Han. Novel side-channel attacks on quasi-cyclic code-based cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):180–212, Aug. 2019.

[47] Falko Strenzke. A timing attack against the secret permutation in the mceliece pkc. In Nicolas Sendrier, editor, *Post-Quantum Cryptography*, pages 95–107, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[48] Falko Strenzke. Message-aimed side channel and fault attacks against public key cryptosystems with homomorphic properties. *Journal of Cryptographic Engineering*, 1(4):283, Oct 2011.

[49] Falko Strenzke. Timing attacks against the syndrome inversion in code-based cryptosystems. In Philippe Gaborit, editor, *Post-Quantum Cryptography*, pages 217–230, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[50] Falko Strenzke, Erik Tews, H. Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan. Side channels in the mceliece pkc. In Johannes Buchmann and Jintai Ding, editors, *Post-Quantum Cryptography*, pages 216–229, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[51] I. von Maurich and T. Güneysu. Lightweight code-based cryptography: Qc-mdpc mceliece encryption on reconfigurable devices. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[52] Ingo von Maurich and Tim Güneysu. Towards side-channel resistant implementations of qc-mdpc mceliece encryption on constrained devices. In Michele Mosca, editor, *Post-Quantum Cryptography*, pages 266–282, Cham, 2014. Springer International Publishing.

[53] Yuan Xing Li, R. H. Deng, and Xin Mei Wang. On the equivalence of mceliece's and niederreiter's public-key cryptosystems. *IEEE Transactions on Information Theory*, 40(1):271–273, Jan 1994.