Side-channel resistance of cryptographic primitives based on error-correcting codes

Agathe Cheriere

COLLEGE DOCTORAL BRETAGNE

MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systémes, Electronique
Spécialité : *Informatique*

Par

Agathe Cheriere

Side-Channel Resistance of Cryptographic Primitives Based on Error-Correcting Codes.

Thèse présentée et soutenue à l'IRISA, Rennes, le 19 Décembre 2023
Unité de recherche : UMR 6074

Rapporteurs avant soutenance :

Guénaël RENAULT — Chercheur associé INRIA Saclay, ANSSI, France
Nicolas SENDRIER — Directeur de recherche, INRIA, FRANCE

Composition du Jury :

Président :	Nicolas SENDRIER	Directeur de recherche, INRIA, FRANCE
Examinateurices:	Lejla BATINA	Professeur, Université de Radboud, Pays-Bas
	Antonia WATCHER-ZEH	Professeur, Université Technique de Munich, Allemagne
Dir. de thèse :	Pierre LOIDREAU	Ingénieur DGA et membre de l'IRMAR, IRMAR, France
Co-dir. de thèse :	Pierre-Alain FOUQUE	Professeur des universités, Université de Rennes, FRANCE
Encadrants :	Benoît GÉRARD	Chercheur associé IRISA, France
	Tania RICHMOND	Maîtresse de conférence Université de Nouvelle-Calédonie, Nouvelle-Calédonie

NOMENCLATURE

Acronyms

λ	Security level: 128,192 or 256
Dec	Decryption or Decapsulation
$Decap$	Decapsulation
$Decrypt$	Decryption
Enc	Encryption or Encapsulation
$Encap$	Encapsulation
$Encrypt$	Encryption
$IND-CCA$	Indistinguishably under chosen-ciphertext attack
$IND-CPA$	Indistinguishably under chosen-plaintext attack
KEM	Key-Encapsulation Mechanism
$KeyGen$	Key Generation
p_k	Private key
PKE	Public-Key Encryption
pu_k	Public key
RSR	Rank Support Recovery Algorithm
sk	Secret key
$SOST$	Sum Of Squared pairwise t-difference T-test
ss	Shared secret

Integers

ω	Weight of the code
ρ	Weight of the error

k_0	Index of a circulant block
l	One element in $\mathscr{L}$
m	Parameter in ROLLO scheme
n_0	Number of circulant blocks
q	A power of a prime number
r	Size of circulant block

Vectors/Matrices

σ'_i	Masks of the second loop for the $i-$th column
σ_i	Masks of the first loop for the $i-$th column
ds	Duplicate syndrome vector cut in blocks of 32-bits
$\mathbf{e} = (\mathbf{e}_0, \mathbf{e}_1)$	Error vector
$\mathbf{s}$	Syndrome vector
$\mathbf{s}^l$	Vector representation of s_l
$\mathbf{x}$	Vector
$\mathbf{x}_i$	i-th element of the vector x
G	Generator matrix
H	Parity-check matrix
$H_{*,j}$	Column j of H
$H_{i,*}$	Row i of H
$H_{i,j}$	Element of H at the $j-$th column and the $i-$th row
H_i	Matrix representation of h_i
J'_{col}	Matrix representing the elimination algorithm in the Gaussian elimination for the $col-$th column
J_{col}	Matrix representing the pivot setting algorithm in the Gaussian elimination for the $col-$th column
M	Matrix of size $m \times n$

S	Syndrome matrix
S_{col}	Matrix representative of the syndrome after the execution of Gaussian elimination for th column *col*

Polynomials

$\mathcal{R}$	Polynomial quotient ring
P	Polynomial in $\mathbb{F}_q[X]$ of degree r
P_m	Polynomial of degree P_m
s	Polynomial of the syndrome vector $\mathbf{s}$
s^l	Product of the polynomial S by Z^l
x	Polynomial
x_i	Coefficient of the polynomial x

Operators

$\cdot$	Multiplication between a vector and a matrix
$\stackrel{\circ}{=}$	Definition
$\oplus$	Exclusive OR
$\otimes$	Scalar multiplication row

Error-correcting codes

$\boldsymbol{c}$	Codeword
$\mathscr{C}$	Linear code
d_H	Hamming distance
d_R	Rank distance
n	Length of a code
$Supp_H$	Hamming support
$Supp_R = \langle \ldots \rangle$	Rank support
w_H	Hamming weight
w_R	Rank weight

Cryptography

$(\mathbf{h}_0, \mathbf{h}_1)$	Private key composed of two vectors
$\mathbf{H}$	Hash function
$\mathbf{K}$	Hash function in BIKE
$\mathbf{L}$	Hash function in BIKE
$\mathcal{H}_\omega$	Set of the private key (h_0, h_1) *s.t.* $w_H(h_0) = w_H(h_1) = \frac{\omega}{2}$
$\mathcal{K}$	Shared secret in BIKE and ROLLO schemes
$\mathcal{m}$	Message
$\mathcal{E}_t$	Set of the error vectors (e_0, e_1) *s.t.* $w_H(e_0) + w_H(e_1) = t$
$\mathcal{L}$	Set of non-zero coordinates in a sparse polynomial
$\mathcal{S}_\omega^r(\mathbb{F}_{2^m})$	Set of vectors in $\mathbb{F}_{q^m}^r$ with a rank weight ω
c	Ciphertext
E	Rank support of $(\mathbf{e}_0, \mathbf{e}_1)$ in ROLLO
F	Rank support of $(\mathbf{h}_0, \mathbf{h}_1)$ in ROLLO
h	Public key, product of $h_1 * h_0^{-1}$

Other

$\mathbb{F}_q$	Finite field of q elements
$a \xleftarrow{\$} A$	Set a random element from A to a
Rx or Ry	32-bit element

Contents

Résumé

L'ordinateur quantique est un sujet captivant qui a fait l'objet de nombreux articles et vidéos. L'idée de créer un ordinateur basé sur la mécanique quantique capable d'atteindre la suprématie quantique, c'est-à-dire le moment où l'ordinateur quantique est suffisamment puissant pour résoudre un problème qu'un superordinateur ne peut pas résoudre, fascine.

L'intérêt pour ce domaine est tel que les grandes entreprises se sont lancées dans une course pour créer un puissant ordinateur quantique capable d'exécuter des algorithmes quantiques, c'est-à-dire des algorithmes utilisant les caractéristiques des bits quantique (quantum bits "qubits"). Dans les faits, la recherche sur les ordinateurs et les algorithmes quantiques n'est pas nouvelle. Le défi consiste désormais à les mettre en pratique et à repousser les limites de l'informatique.

Néanmoins, il s'agit, d'une part, d'une amélioration considérable des capacités et, d'autre part, d'une menace sérieuse pour la sécurité de nos informations. En effet, la cryptographie sera sérieusement affectée par le développement d'un ordinateur quantique aux performances excellentes. Depuis les années 1990, il existe deux algorithmes quantiques, les algorithmes de Grover et de Shor (du nom de leurs auteurs), qui réduisent la sécurité des systèmes cryptographiques actuels. L'algorithme de Grover pourrait être utilisé comme une attaque par force brute contre la cryptographie symétrique. Cette attaque peut être contrée en augmentant la taille des clés. S'il est utilisé comme attaque, l'algorithme de Shor est plus puissant que celui de Grover. Les algorithmes cryptographiques à clé publique actuellement utilisés sont basés sur des problèmes de la théorie des nombres et, par exemple, sur la difficulté de factoriser un nombre avec un ordinateur conventionnel. Cependant, l'algorithme de Shor permet de trouver les nombres premiers composants un entier N en un temps polynomial. Par conséquent, l'algorithme de Shor casse la majorité des cryptosystèmes asymétriques déployés aujourd'hui.

Cela pose le problème du remplacement de ces algorithmes cryptographiques essentiels par d'autres algorithmes cryptographiques qui ne sont pas menacés par l'algorithme de Shor. Ces algorithmes sont alors dits post-quantiques. Pour se préparer à l'arrivée potentielle des ordinateurs quantiques, le National Institute of Standards and Technology (NIST) a lancé, fin 2016, un processus de standardisation post-quantique [60, 62]. À la fin de l'année 2017, il y avait plus de 23 schémas de signature et 59 schémas de chiffrement/encapsulation de clé candidats. Il s'en est suivi plusieurs années d'étude de ces schémas

par la communauté scientifique. L'objectif était de tester leur sécurité face à des attaques mathématiques ou physiques afin de sélectionner les prochains standards. Les travaux menés au cours de cette thèse s'inscrivent dans ce cadre, plus précisément, entre la fin du deuxième et du quatrième tour de cette standardisation post-quantique.

L'étude de la sécurité de schémas cryptographiques est complexe et peut être divisé en deux parties principales. La première concerne la résistance des problèmes sur lesquels se basent les cryptosystèmes et de leurs structures face à des attaques mathématiques. Autrement dit, vérifier qu'il n'y a pas de moyens de casser ou réduire la sécurité du cryptosystème avec nos connaissances actuelles en mathématiques. La seconde partie se focalise sur la sécurité une fois que les schémas sont implantés. Nous parlons alors de sécurité de l'implantation des schémas face à des attaques physiques, introduites dans les années 90 [47, 15], telles que les canaux auxiliaires et les injections de fautes. Outre la différence de technique entre les attaques mathématiques et les attaques physiques, il y a une différence sur l'impact plus globale des attaques sur la sécurité des schémas. C'est-à-dire, lorsqu'il y a une attaque sur la structure mathématique utilisée pour la clé privée ou secrète alors tous les cryptosystèmes se basant sur cette structure sont impactés. Alors que pour une attaque physique, cela ne va pas automatiquement impacter les autres implantations du même schéma,*i. e.* elle ne casse pas le cryptosystème. Bien qu'à première vue, les attaques physiques semblent être une menace plus faible pour le schéma en tant que tel, elles sont la source d'un risque important d'abolition de la protection rendant inefficace le cryptosystème. D'où le fort intérêt montré par le NIST pour ce sujet durant le processus de standardisation post-quantique [61].

Dans le cadre de cette thèse, nous nous sommes intéréssés à la résistance aux attaques par canaux auxiliaires des cryptosystèmes basés sur les codes correcteurs d'erreurs.

Au début du processus de standardisation post-quantique du NIST, il y avait peu de travaux sur la résistance des implantations de cryptosystèmes basés sur les codes-correcteurs d'erreurs. La première attaque sur une implantation du cryptosystème de McEliece ayant été publiée à peine une quinzaine d'années au par avant [77]. Une majorité des travaux sur les attaques physiques concerne des attaques temporelles [73, 75, 7, 76, 16] et des attaques exploitant les variations de la consommation électrique du programme exécuté[44, 19, 18, 64, 71]. Parallèlement, les premières implantations optimisées et en temps constant de cryptosystèmes basés sur les codes correcteurs ont été proposées [14, 26, 25, 33, 31, 32]. Nous nous sommes alors demandés si les contre-mesures permettant d'assurer le temps constant, c'est-à-dire empêcher des attaques temporelles, ainsi que les optimisations, n'étaient pas de potentielles portes d'entrée pour attaquer les cryptosystèmes avec d'autres attaques par canaux auxiliaires.

Contributions

Nous nous sommes focalisés sur deux schémas candidats à la standardisation du NIST : BIKE et ROLLO [1, 2]. En commençant par ce dernier.

ROLLO

ROLLO est un schéma basé sur la métrique rang. Il utilise plus particulièrement les codes LRPC (Low-Rank Parity-Check) apparut pour la première fois dans un schéma cryptographique en 2013 [36]. Notre approche fût d'analyser l'implantation dévéloppée par les auteurs de ROLLO, puis de construire une attaque en ne ciblant qu'une fonction identifiée comment étant en temps constant. Nous avons alors mis en évidence une fuite d'information dans une fonction essentielle du schéma, nous permettant ainsi de reconstruire le syndrome et de remonter vers la clé privée. Les données nécessaires sont obtenues grâce à une analyse de la consommation. Nous avons démontré grâce à l'expérimentation que l'attaque est réalisable en ne récupérant qu'une seule trace.

Ce premier travail sur ROLLO met en avant une vulnérabilité liée à l'application du temps constant dans l'implantation de ce schéma. Ce travail a soulevé une interrogation sur l'existence de failles similaires dans des implantations de schémas plus étudiés basés sur la métrique de Hamming tel que BIKE.

BIKE

BIKE est un cryptosystème basé sur la métrique de Hamming. Pour ce schéma, ce sont des codes QC-MDPC (Quasi-Cyclic Moderate-Density Parity-Check) qui sont utilisés. Bien qu'étudiés que depuis une dizaine d'années dans le cas des cryptosystèmes [55], ils furent rapidement implantés [52, 26]. Deux attaques par analyse de consommation furent alors proposées sur l'implantation en temps constant des codes QC-MDPC [71, 74]. Il est intéressant de noter que dans la dernière implantation en temps constant de BIKE pour Cortex-M4 [20], il n'est plus possible d'exploiter les fuites mises en avant par ces attaques.

En travaillant sur cette implantation, nous avons mis en évidence une faille dans une des fonctions. Ce qui est intéressant, c'est que cette faille existe à la fois dans l'implantation en C, mais aussi dans sa version optimisée utilisant de l'assembleur. Pour l'attaque sur BIKE, nous ne nous sommes pas arrêtés à l'exposition d'une fuite. Nous avons cherché à connaître les limites de notre attaque. Ce qui nous a amené à utiliser de l'apprentissage automatique (*machine learning* en anglais) combiné avec une attaque classique contre les cryptosystèmes basés sur les codes de Hamming.

Contre-mesures

Les contre-mesures peuvent être particulièrement complexes à développer et à mettre en place. En effet, de nombreux facteurs sont à prendre en compte, d'une part, il faut s'assurer que la mesure de protection élimine la faille ou au moins empêche son exploitation avec l'attaque. Mais il faut aussi s'assurer, dans la mesure du possible, qu'il n'y a pas de possibilités de trouver une autre attaque en utilisant d'autres techniques. Par exemple, avoir une implantation protégée des attaques par analyse simple de la consommation, mais pas des attaques par analyse différentielle de la consommation. Pour aller plus loin, il est possible de prendre en compte des contraintes sur la mémoire et le temps d'exécution, ce qui complique la tâche, car les contre-mesures sont par natures plus coûteuses que l'implantation d'origine.

Suite à nos attaques sur ROLLO et BIKE nous avons commencé à développer des contre-mesures les en empêchant. Nous nous sommes intéressés deux façons de protéger les implantations. La première concerne l'élimination totale de la faille dans l'implantation. Autrement dit, soit remplacer, soit cacher la ou les opérations à l'origine de la fuite par une ou plusieurs autres opérations, rendant ainsi impossible l'extraction des informations nécessaires aux attaques. Cependant cette méthode a souvent l'inconvénient de nécessiter plus de temps de calcul, entrainant l'augmentation du temps d'exécution du cryptosystème. La seconde contre-mesure a pour objectif de rendre impossible le fait de remonter jusqu'à la clé privée malgré les informations obtenues. Autrement dit, laisser la ou les opérations à l'origine de la faille, mais modifier l'implantation de façon à ce qu'elle ne soit pas exploitable. Par exemple en mélangeant l'ordre d'exécution.

Ces travaux ont mené aux publications suivantes.

Publications

Colloque international

* **Agathe Cheriere**, Lina Mortajine, Tania Richmond, Nadia El Mrabet. Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis. Proceedings of 2022, The twelfth International Workshop on Coding and Cryptography (WCC 2022), *Rostock, Allemagne*, Mars 2022.

Article de journal

* **Agathe Cheriere**, Lina Mortajine, Tania Richmond, Nadia El Mrabet. Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis. Designs, Codes and Cryptography (DCC), 2023.

Conférence internationale

* **Agathe Cheriere**, Nicolas Aragon, Tania Richmond, Benoît Gérard. BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding. Proceeding of 2023, 21th International Conference on Applied Cryptography and Network Security (ACNS 2023). *Kyoto, Japon.* **Récompensé du prix : Best student paper award**.

INTRODUCTION

The quantum computer is an enthralling subject of numerous articles and videos. The idea of creating a computer based on quantum mechanics capable of achieving quantum supremacy, *i. e.* the moment when the quantum computer is powerful enough to solve a problem that a supercomputer cannot solve, is fascinating. Interest in the field is so great that major companies are racing to create a powerful quantum computer able to run significant quantum algorithms, *i. e.* algorithm using characteristics of the qubits. In fact, research into quantum computers and algorithms is nothing new. The challenge is now to put them into practice and push back the computer computation limit.

Nevertheless, what is, on the one hand, a major improvement in capabilities and, on the other, a severe threat to the security of our information. Indeed, cryptography will be seriously impacted by the development of a quantum computer with excellent performance. Since the 1990s, there have been two quantum algorithms, Grover and Shor algorithms (named after their authors) [41, 72], which reduce the security of the current cryptographic schemes. Grover's algorithm could be used as a brute-force attack against symmetric cryptography, but increasing the keys sizes prevents this attack. If used as an attack, Shor's algorithm is more potent than Grover's. The public key cryptographic algorithms currently in use are based on number theory and, more specifically, on the difficulty of factoring a large number with a conventional computer. However, Shor's algorithm makes it possible to find the prime numbers composing an integer N in polynomial time. As a result, most of the asymmetric cryptosystems that are currently in use can be broken by Shor's algorithm.

Replacing these essential cryptographic algorithms with other cryptographic algorithms that are not threatened by Shor's algorithm is a challenge. These resistant algorithms are then said to be post-quantum. To prepare for the potential arrival of quantum computers, the National Institute of Standards and Technology (NIST) launched a post-quantum standardization process at the end of 2016 [60, 62]. By the end of 2017, there were more than 23 candidate signature schemes and 59 candidate encryption/KEM schemes. It was followed by several years of study of the schemes. The aim was to test their security against mathematical or physical attacks to select the next standards. The work carried out during this thesis falls within this framework, specifically between the end of the second and fourth rounds of post-quantum standardization.

Studying the security of cryptographic schemes is complex and can be divided into two main parts. The first concerns the resistance of the problems on which cryptosystems and their structures are based to mathematical attacks. In other words, we are checking that there are no ways of breaking or reducing the security of the implementation with our current knowledge of mathematics. The second part focuses on security once the schemes have been implemented. We then discuss the security of scheme implementation against physical attacks introduced in the 90s [47, 15], such as side channels and fault injections. Mathematical and physical attacks differ in technique and they both impact security schemes differently.In other words, when there is an attack on the mathematical structure used for the private or secret key, all the cryptosystems based on this structure are impacted. A physical attack, on the other hand, will not automatically impact other implementations of the same scheme since it does not break the cryptosystem. While physical attacks may seem to pose a lesser threat to cryptographic systems, they actually present a substantial risk of compromising their protection and making them ineffective. As a result, NIST has demonstrated a keen interest in this topic during the post-quantum standardization process [61].

In this thesis, we focus on side-channels resistance of cryptosystems based on error-correcting codes.

At the start of NIST's post-quantum standardization process, there was little work on the resistance of error-correcting code-based cryptosystem implementations. The first side-channel attack on an implementation of the McEliece cryptosystem had been published barely a decade before [77]. Most of the work on physical attacks concerns timing attacks [73, 75, 7, 76, 16] and power analysis attacks [44, 19, 18, 64, 71]. At the same time, the first optimized constant-time implementations of cryptosystems based on error-correcting codes were proposed [14, 26, 25, 33, 31, 32]. We then wondered whether countermeasures to ensure constant time, i.e. to prevent timing and caches attacks, as well as optimizations, could be vulnerabilities for other side-channel attacks on cryptosystems.

Contributions

We looked at two candidate schemes for NIST standardization: ROLLO and BIKE [2, 1]. We are starting with ROLLO.

ROLLO

ROLLO is a scheme based on rank metric. It uses LRPC (Low-rank parity-check) codes, which appeared for the first time in a cryptographic scheme in 2013 [36]. Our approach was to analyze the implementation developed by the authors of ROLLO. Then, we built an attack targeting only one function, which we identified as being in constant time. We then

identified an information leakage in an essential function of the scheme, enabling us to reconstruct the syndrome and trace it back to the private key. We obtained the necessary data by analyzing power consumption.n. We have demonstrated through experimentation that the attack can be carried out with just one trace.

Through our work on ROLLO, we discovered a vulnerability in the implementation of the constant time scheme. This has led us to question whether similar vulnerabilities exist in more widely studied schemes which are based on the Hamming metric.

BIKE

BIKE is a cryptosystem based on the Hamming metric. QC-MDPC (Quasi-Cyclic Moderate-Density Parity-Check) codes are used for this scheme. Although only studied for about ten years in the case of cryptosystems [55], they were quickly implemented [52, 26]. Two attacks by power analysis were then proposed on the implementation in constant time of the QC-MDPC codes [71, 74]. Interestingly, in the latest constant-time implementation of BIKE for Cortex-M4 [20], it is no longer possible to exploit the leakages highlighted by these attacks.

While working on this implementation, we discovered a leakage in one of the functions. Interestingly, this leakage exists both in the C implementation and in its assembly version. For the BIKE attack, we did not stop at exposing a leakage. We looked at the limits of our attack. It led us to use machine learning combined with a classic attack on cryptosystems based on Hamming codes.

Countermeasures

Countermeasures can be particularly complex to develop and implement. There are many factors to consider. We must ensure the protection measures eliminate or prevent data leakage from the proposed attacks. However, as far as possible, we must also ensure that there is no possibility of finding another attack using other techniques, for example, having an implementation that protects against attacks using simple power analysis but not against attacks using differential power analysis. Furthermore, it is possible to consider constraints on memory and execution time, which complicates the task, as countermeasures are more expensive on memory and execution time than the original implementation.

Following our attacks on ROLLO and BIKE, we developed countermeasures to prevent them. We looked at two ways of protecting the implementations. The first involves entirely eliminating the leakage in the implementation. In other words, either replace or hide the operation(s) at the origin of the flaw by one or more other operations, thus making it impossible to extract the information needed for the attacks. However, this method often has the drawback of requiring more computing time, leading to an increase in the execution time of the cryptosystem. The second countermeasure is to make it impossible to recover

the private key despite the information obtained. In other words, keep the operation(s) at the origin of the leakage but modify the implementation so that it cannot be exploited, for example, by mixing the execution order.

Our works led to the following publications.

Publications

International Workshop

* **Agathe Cheriere**, Lina Mortajine, Tania Richmond, Nadia El Mrabet. Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis. Proceedings of 2022, The twelfth International Workshop on Coding and Cryptography (WCC 2022), *Rostock, Germany*, March 2022.

Journal

* **Agathe Cheriere**, Lina Mortajine, Tania Richmond, Nadia El Mrabet. Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis. Designs, Codes and Cryptography (DCC), 2023.

International Conference

* **Agathe Cheriere**, Nicolas Aragon, Tania Richmond, Benoît Gérard. BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding. Proceeding of 2023, 21th International Conference on Applied Cryptography and Network Security (ACNS 2023). *Kyoto, Japan*. June 2023 **Best student award**.

Chapter 1

Cryptology

The definitions of the vocabulary introduced in this chapter are taken from the NIST glossary [59].

1.1 Cryptology and Various Definitions

Cryptography, previously viewed as a military art, refers to the science of secure and usually confidential information transmission. It includes cryptography, the creation and the usage of methods by allies to exchange messages illegible to anyone not concerned. It also involves cryptanalysis, which is all techniques used to break the code, protecting the messages and making them readable.

Definition 1. *(Cryptology) Cryptology is the collection and exploitation of communication and solutions, products, and services to ensure the availability, integrity, authentication, confidentiality, and non-repudiation of telecommunications and information systems. The field encompasses cryptography and cryptanalysis.*

Remark 1. *Derives from the Greek words* `kryptós` *for "hidden" and* `lógos` *for "word," cryptology etymologically means the hidden words. While cryptography, resp. cryptanalysis, express the notion "to write", resp. "to untie", with* `gráphen`*, resp.* `analýen`*.*

1.2 Cryptography

Definition 2. *(Cryptography) The discipline embodies the principles, means, and methods for the transformation of data in order to hide their semantic content, prevent their unauthorized use, or prevent their undetected modification.*

In other words, cryptography is the set of techniques used to transform a clear message into an encrypted message, known as a cipher. It includes the study and design of a cryptographic algorithm to ensure the following properties:

- Confidentiality: Preserving authorized restrictions on information access and disclosure,
- Integrity: Guarding against improper information modification or destruction,

- Authenticity: The property of being confident in the validity of a transmission, a message, or a message originator,
- Non-repudiation: Assurance the sender of data is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the data.

Only some of the properties are required each time, rarely all. Indeed, cryptographic algorithm designates various protocols, from authentification to encryption, and they all have different requirements.

1.3 Encryption and Decryption

Among all the protocols, the most commonly known is the encryption, *i. e.* the cryptographic transformation of data (plaintext) to produce ciphertext. Its related protocol, the decryption, *i. e.* the process of changing ciphertexts into plaintexts.

The encryption mechanism ensures confidentiality thanks to a key. The decryption mechanism removes confidentiality if the key is known. There is a necessity to generate a key. The mechanism is called key generation.

Definition 3. *(Key generation (KeyGen)) The generation of a cryptographic key either as a single process using a random bit generator and an approved set of rules or as created during key agreement or key derivation.*

A key generation algorithm takes established parameters to ensure a security level λ and returns the keys. Sometimes, the algorithm returns some other elements randomly generated.

The keys generated for cryptography can take different forms depending on their use, whether symmetric or asymmetric.

1.3.1 Symmetric Cryptography

Symmetric cryptography, also known as secret-key cryptography, is the oldest encryption algorithm. To give a few examples: the Caesar cipher is one of the oldest forms of encryption, and, more recently, the Enigma machine, which was made famous to the general public with a movie named "The Imitation Games".

In symmetric cryptography, Alice and Bob share the same key to encrypt and decrypt the message. The key is unknown except for Alice and Bob; hence its name: secret key (sk), see Figure 1.1.

Definition 4. *(Symmetric cryptography) A cryptographic algorithm that uses the same secret key for its operation and, if applicable, for reversing the effects of the operation.*

Encryption (Encrypt) Cryptographic algorithm that, given a message m and a secret key sk, returns a ciphertext c. $Encrypt(m, sk) = c$.

Decryption (Decrypt) Cryptographic algorithm that, given a ciphertext c and a secret key sk, returns the message m. $Decrypt(c, sk) = m$.

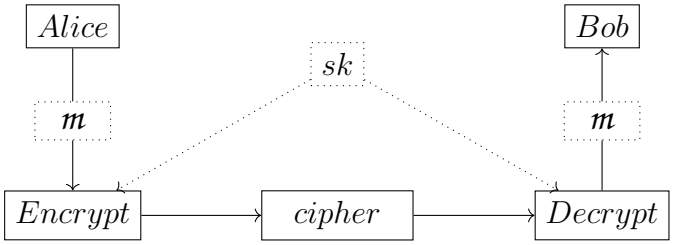


Figure 1.1: Symmetric cryptography

The effectiveness of symmetrical cryptography is well established. However, it has one major drawback: it requires the two parties communicating to have established a key beforehand. Symmetric cryptography makes setting up new secure communications between different people difficult.

Remark 2. *The algorithm to encrypt and decrypt in symmetric cryptography can be the same. For instance, it is the case with the advance encryption standard (AES) [68].*

1.3.2 Asymmetric Cryptography

Asymmetric cryptography was introduced in the mid-1970s and enabled the possibility of exchanging encrypted messages without establishing a secret key beforehand. It is possible thanks to the creation of new type of keys, the public and private keys. The idea behind asymmetric cryptography is to use two keys generated by Bob with distinct goals. In asymmetric encryption, Bob gives everyone the public key pu_k, and Alice uses pu_k to encrypt the message. On his side, Bob can decrypt the message using his private key p_k. p_k is only known by Bob, so only his can decrypt the message. This is possible because the private and public keys are related with the public key usually being derived from the private key. It is important to note that the public key cannot be used to recover the private key.Asymmetric cryptography is also known as public-key cryptography.

Definition 5. *(Asymmetric cryptography) Cryptography that uses two separate keys to exchange data, one to encrypt or digitally sign the data and one for decrypting the data or verifying the digital signature.*

The exchange of messages is not the only use of the public-key cryptography. Indeed, it can also be used to sign a document; it is called a digital signature.

Public-Key Encryption

Asymmetric encryption, the latter is often referred to as public-key encryption or PKE in order to distinguish it from symmetric encryption. In PKE, Bob generates the private key and the related public key with the key generation mechanism. The encryption and decryption mechanisms are described as follows:

Encryption (Encrypt) Cryptographic algorithm that, given a message m and a public key pu_k, returns a ciphertext c. $Encrypt(m, pu_k) = c$.

Decryption (Decrypt) Cryptographic algorithm that, given a ciphertext c and a private key p_k, returns the message m. $Decrypt(c, p_k) = m$.

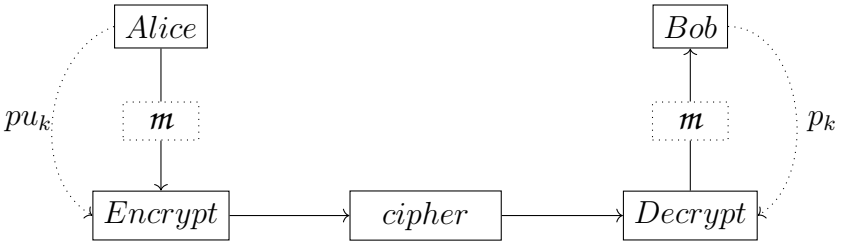


Figure 1.2: Public-key encryption.

Nonetheless, the public-key encryption requires more computations than the symmetric encryption. Therefore, it became usual to use PKE to exchange a secret key between Alice and Bod to use symmetric encryption afterward. Asymmetric cryptography includes mechanisms solely focused on key establishment.

Key-Encapsulation Mechanism

The key-encapsulation mechanism, KEM for short, is a procedure composed of two algorithms, encapsulation and decapsulation, in addition to KeyGen. Alice executes the encapsulation only with pu_k. No message is given as input parameters. The encapsulation generates the secret key, referred to as shared secret ss in KEM, and, Alice sends it in encrypt form c to Bob. The decapsulation algorithm reverses the encapsulation to obtain ss.

Encapsulation (Encap) Cryptographic algorithm that, given a public key pu_k, returns a ciphertext c and a share secret ss. $Encap(pu_k) = (c, ss)$.

Decapsulation (Decap) Cryptographic algorithm that, given a ciphertext c and a private key p_k, recovers the share secret ss. $Decap(c, p_k) = ss$.

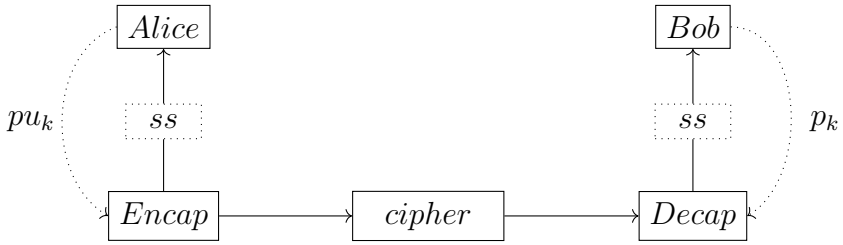


Figure 1.3: Key-encapsulation mechanism.

Classical and Post-Quantum Cryptography

There is a distinction between the classical and the post-quantum cryptography in public-key cryptography. The ones currently deployed are classical cryptography. It is based on the number theory problems, specifically on the factorization problem and discrete logarithm problem. RSA and Diffie-Hellman are two examples of classical cryptographic schemes [70, 30].

Nevertheless, Peter Shor introduced, in 1996, a quantum algorithm that finds the prime factors of an integer in polynomial time [72]. Therefore, Shor's algorithm breaks the problem on which classical cryptography relies. Although it depends on the existence of a powerful quantum computer, Shor's algorithm is a major threat to cryptography currently in use.

Here comes the notion of post-quantum cryptography. It refers to cryptography schemes that rely on mathematical problems not threatened by Shor's algorithm. For example, the schemes are based on problems on lattices, codes, isogenies, multivariates or hash.

Remark 3. *The notion of post-quantum cryptography does not mean recent cryptography. Indeed, the McEliece scheme in coding theory was proposed in 1978, one year after the RSA scheme (1977) [70].*

Remark 4. *Post-quantum cryptography should not be confused with quantum cryptography. The latter is based on quantum mechanics, whereas the former is simply resistant to currently known attacks using quantum algorithms.*

1.3.3 Other Cryptographic Schemes

Encryptions and KEMs are not the only cryptographic schemes and mechanisms. For example, there are identification protocols, digital signatures, hash functions, and random number generators. The security properties required by these systems are different because they have other purposes than sending a message. In this subsection, we formally defined the digital signature and the hash functions as we refer to them in this manuscript.

Digital Signature

The digital signature is one field of asymmetric cryptography. At the difference of the PKEs, the digital signature does not ensure the confidentiality of a message. The message is sent without encryption. The purpose of the digital signature is to certify the identity of the signer of a document and the integrity of the data. In the signature, Bob signs a message or document with its private key. The signature ($sign$) and the message, in clear, are sent to Alice. Alice verifies the signature with the public key. Figure 1.4 represents a digital signature scheme.

Definition 6. *(Digital signature) The result of a cryptographic transformation of data that, when properly implemented, provides a mechanism for verifying origin authentication, data integrity, and signatory non-repudiation.*

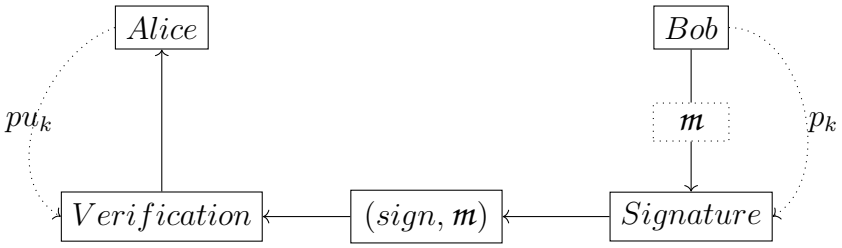


Figure 1.4: Digital signature.

From now on, the digital signature is shortened to signature.

Hash Functions

A hash function is a mathematical function that maps a bits string of arbitrary length to a fixed length bits string. Hash values (the result of the hash function) have two main security properties. It is impossible to recover the input from a hash value, and two different inputs have necessary different hash values. For cryptographic hash function, the latter property is ensured by collision and second preimages resistances.

Definition 7. *(Cryptographic Hash function) A function on bit strings in which the length of the output is fixed. Approved hash functions are designed to satisfy the following properties:*

1. *(One-way) It is computationally infeasible to find any input that maps to any new pre-specified output*
2. *(Collision-resistant) It is computationally infeasible to find two distinct inputs that map to the same output.*
3. *(Second preimage resistance) It is computationally infeasible to find a second preimage of a known message.*

1.4 Cryptanalysis

Definition 8. *(Cryptanalysis) The study of mathematical techniques for attempting to defeat cryptographic techniques and/or information systems security. It includes the process of looking for errors or weaknesses in the implementation of an algorithm or of the algorithm itself.*

In other words, cryptanalysis is a set of techniques that an attacker uses to recover the message or the keys of cryptographic mechanisms.

1.4.1 Attack Models

Testing the security of cryptographic schemes can be done by attempting to attack them. The level of resistance a scheme has against cryptanalysis attacks determines its level of security. The models of attack are classified from weakest to strongest as follows:

- Known-ciphertext attack: the attacker has only access to the ciphertexts to attack.
- Know-plaintext attack: the attacker has access to a limited number of plaintexts and their corresponding ciphertexts.
- Chosen-plaintext attack (CPA): the attacker is able to choose several plaintexts to be encrypted and receive the corresponding ciphertexts.
- Chosen-ciphertext attack (CCA): the attacker is able to choose several ciphertexts to be decrypted and receive the corresponding plaintexts.

To give an example of an attack, searching exhaustively the keys by trying all the possibilities to obtain the plaintext of a ciphertext is a known-ciphertext attack.

The CPA and CCA models have a second version called advanced chosen-plaintext attack (CPA2) and advanced chosen-ciphertext attack (CCA2). The difference is that after each call to the encryption or the decryption, the attacker analyses the result before choosing the next plaintext/ciphertext.

A scheme is said to be CPA secure if the scheme resists CPA attacks.

1.4.2 Indistinguishably Attack Model

The indistinguishability under CPA, resp. CCA, is an attack model with the idea that given two messages, resp. ciphertexts, and a ciphertext, resp. plaintext, the attacker cannot determine the message, resp. ciphertext, at the origin of the ciphertext, resp. plaintext.

The IND-CPA and IND-CCA are described as games as follows:

Indistinguishability under CPA

The indistinguishability under chosen-plaintext attack starts with the challenger generating the private and public keys. The adversary chooses two plaintexts p_0 and p_1 and sends them to the challenger, which randomly picks one and sends the challenge ciphertext. The adversary is free to encrypt any plaintext except the ones from the challenge to determine which one was chosen. The scheme is said to be IND-CPA is secure if the adversary has a negligible advantage in deciding which encrypted plaintext. Figure 1.5 formalizes the games for the IND-CPA model.

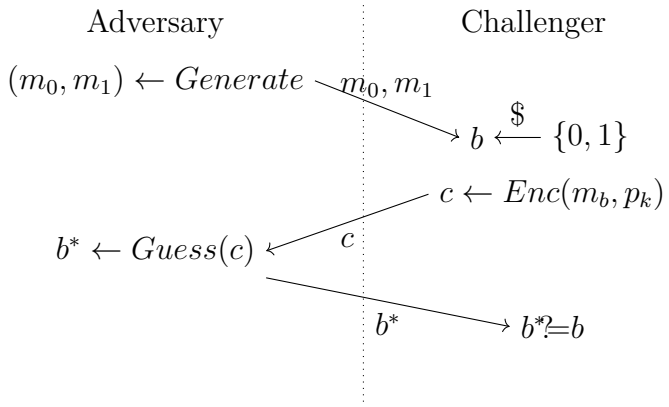


Figure 1.5: IND-CPA game

Indistinguishability under CCA

The game for indistinguishability under chosen-ciphertext attack is similar to the one for the IND-CPA game. The difference is that the adversary has access to a decryption oracle, which decrypts any ciphertext given by the adversary. The only ciphertext that cannot be given to the oracle is the challenge ciphertext.

In the IND-CCA1 (non-adaptive) model, the adversary is forbidden to use the oracle after the reception of the challenge ciphertext. While on the IND-CCA2 (adaptative) model, the adversary can call the oracle decryption after receiving the challenger ciphertext.

This manuscript's notation IND-CCA will refer to the IND-CCA2 security model.

Chapter 2

Error-Correcting Codes for Cryptography

In this chapter, we will be introducing error-correcting codes used in cryptography, specifically the Hamming and rank metrics. We will go through each metric and explain the different sets of codes that are required to fully comprehend the cryptographic schemes introduced in Chapter 4. The final section of this chapter will cover the McEliece and Niederreiter code-based cryptographic schemes.

2.1 Generality on Error-correcting Codes

Coding theory was introduced by Richard W.Hamming [43] in 1950. With the first error-correcting code, the Hamming (7,4) code, he proposed a solution to one of the main data transmission problems, the errors in the messages due to a noisy communication channel. Indeed, a communication channel, such as wires, may be disrupted by interference, leading to a modification between the message Alice sent and the one Bob received. Figure 2.1 represents a noisy communication channel. These modifications are called errors. As their name suggests, error-correcting codes are techniques to detect and correct errors using coding theory.

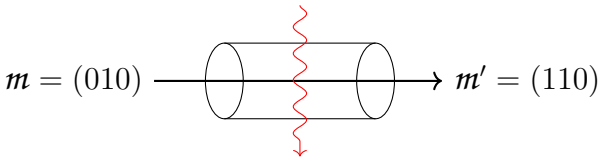


Figure 2.1: Noisy communication channel.

The idea is to add some information redundancy to the message during the encoding process. During the decoding process, the redundancy is used to determine if there are errors, their position, and correct them.

Example 1. *Let us have binary message* $m = (1010)$ *sent by Alice. We encode the message as follows:*

$$c = (m|m) = (10101010)$$

c is send through a noisy channel and Bob received $c' = (10001010)$. With the decoding, Bob can say that the third of the m bit was modified. However, Bob cannot determine if the right value is 0 or 1. However, if m is encoded as follows:

$$c = (m|m|m) = (101010101010)$$

Then, Bob can correct the error and retrieve the initial message.

Nonetheless, the capacity of correction by the codes is limited, and if too many errors occur, the codes are unable to correct. Even potentially return an incorrect correction. The capacity of correction is determined for each code family based on certaincharacteristics, including distance.The codes and vocabulary are formally described in the remainder of this chapter.

Since their introduction, error-correcting codes have proven their worth in different fields, such as communications, data storage, and the one we are looking for, cryptography.

2.1.1 Definition

In this chapter, we only introduce notions related to the work presented in the thesis.

Let $\mathbb{F}_q$ be a finite field of q elements, where q is a power of a prime number.

Definition 9. *(Linear code) A (n, k)-linear code $\mathscr{C}$ of length n, dimension k and co-dimension $(n-k)$ is a k-dimensional vector subspace of $\mathbb{F}_q^n$.*

In other words, a linear code is a set of vectors of length n, which is closed under addition and scalar multiplication. Given a message $m \in \mathbb{F}_q^k$, the codeword is the result of the encoding process of m with the code $\mathscr{C}$.

Definition 10. *(Codeword) Any vector belonging to $\mathscr{C}$ is called a codeword.*

The linear code $\mathscr{C}$ has a basis of k elements that, put in matrix form, compose the generator matrix of $\mathscr{C}$.

Definition 11. *(Generator matrix) A matrix $G \in \mathbb{F}_q^{k\times n}$ is called a generator matrix of the linear code $\mathscr{C}$ if its rows form a basis of $\mathscr{C}$.*

The code can then be written as follows:

$$\mathscr{C} \doteq \{m \cdot G | m \in \mathbb{F}_q^k\}$$

Note that the elementary operation on a matrix can be applied on G:

1. Suppression of rows with only 0,
2. Multiplication of a row by a scalar,

3. Add a combination of rows to another row,

4. Swap two rows.

Suppressing empty rows (with only 0) is unnecessary for G as it is a full-rank matrix.

Definition 12. *(Rank) The rank of a matrix A is the dimension of the vector space spanned by its rows (resp. columns). It corresponds to the maximal number of linearly independent rows (resp. columns). The rank is denoted as* $\mathbf{Rank}(A)$.

Definition 13. *(Full rank) A matrix is said to have full rank if it equals the largest possible dimension, the number of columns or rows.*

With the operations on matrices previously described, we modify G to have an identity matrix on the left part. G is in systematic form.

Definition 14. *(Systematic form) The generator matrix G is said to be in systematic form if G is written as:*

$$G \triangleq (I_k|A)$$

From the systematic form of G, we can obtain the parity-check matrix of $\mathscr{C}$, computed as follows:

$$H \triangleq \left(-(A)^T|I_{n-k}\right)$$

Definition 15. *(Parity-check matrix) A matrix $H \in \mathbb{F}_q^{(n-k)\times n}$ is called a parity-check matrix of $\mathscr{C}$ if and only if*

$$\mathscr{C} = \{\boldsymbol{c} \in \mathbb{F}_q^n | \boldsymbol{c} \cdot H^T = 0\}$$

Let us define $\mathbf{y} \in \mathbb{F}_q^n$ as $\mathbf{y} = \boldsymbol{c} + \mathbf{e}$, where $\mathbf{e}$ is the error vector. In other words, a vector in $\mathbb{F}_q^n$ containing the position and values of modification of the codeword $\boldsymbol{c}$. $\mathbf{y}$ is not a codeword so $\mathbf{y} \cdot H^T \neq 0$. In fact, $\mathbf{y} \cdot H^T = \mathbf{e} \cdot H^T$ as $\mathbf{y} \cdot H^T = \boldsymbol{c} \cdot H^T + \mathbf{e} \cdot H^T$ and $\boldsymbol{c} \cdot H^T = 0$. The $\mathbf{e} \cdot H^T$ is called the syndrome.

Definition 16. *(Syndrome) A syndrome $\mathbf{s} \in \mathbb{F}_q^k$ of a vector $\mathbf{e} \in \mathbb{F}_q^n$ is*

$$\mathbf{s} \triangleq \mathbf{e} \cdot H^T.$$

The syndrome is essential in coding theory as it is usually used to recover the error vector and decode the codeword.

Another essential notion in coding theory is the distance. When we talk about the Hamming or rank metrics, it actually refers to the method used to determine the distance between two codewords. The distance determines the error-correction capacity of any code.

Definition 17. *(Distance) Let us consider a set E. $d : E \times E \to \mathbb{R}_+$ is a distance function on E if and only if it satisfies the following properties for $x, y, z \in E$:*

1. *(Positivity) If $x \neq y$, then $d(x, y) > 0$.*

2. *(Symmetry) $d(x, y) = d(y, x)$.*

3. *(Triangle inequality) $d(x, y) + d(y, z) \geq d_i(x, z)$.*

The Hamming and rank distances are respectively defined in Section 2.2 and Section 2.3.

In a code $\mathscr{C}$, the minimum distance between two codewords determines the correction capacity of $\mathscr{C}$.

Definition 18. *(Minimum distance) Let d be a distance, the minimum distance of a code $\mathscr{C}$ is the minimum distance between two distinct codewords.*

$$d_{min} \triangleq min_{\substack{\mathbf{x},\mathbf{y}\in\mathscr{C} \\ \mathbf{x}\neq\mathbf{y}}} d(\mathbf{x}, \mathbf{y})$$

Definition 19. *(Correction capacity) The correction capacity is defined as follows:*

$$t \triangleq \left\lfloor \frac{d_{min} - 1}{2} \right\rfloor$$

The correction capacity is different for every family of code. Indeed, according to the metric and the structure of a family of codes, it does not have the same characteristics and it does have the same error-correction capacity.

In the next section, we introduce the first metric for coding theory used in cryptography: the Hamming metric.

2.2 Hamming Metric

2.2.1 Generalities

The principle behind the Hamming distance is quite simple. Indeed, given two vectors $(\mathbf{x}, \mathbf{y}) \in (\mathbb{F}_q^n)^2$, their distance $d_H(\mathbf{x}, \mathbf{y})$ is the number of distinct coordinates between $\mathbf{x}$ and $\mathbf{y}$.

Definition 20. *(Hamming distance) The Hamming distance d_H between two codewords, $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ in $\mathbb{F}_q^n$ and $\mathbf{y} = (\mathbf{y}_1, \ldots, \mathbf{y}_n)$ in $\mathbb{F}_q^n$, is defined as follows:*

$$d_H(\mathbf{x}, \mathbf{y}) \triangleq |\{i, \mathbf{x}_i \neq \mathbf{y}_i\}| .$$

If $\mathscr{C}$ is a binary code, then $\mathbf{x}$ and $\mathbf{y}$ are in $\mathbb{F}_2^n$ and their distance is the number of non-zero elements of $\mathbf{z} = \mathbf{x} \oplus \mathbf{y}$. In other words, the distance between $\mathbf{x}$ and $\mathbf{y}$ is the Hamming weight of $\mathbf{z}$ or the number of elements in the support of $\mathbf{z}$.

Definition 21. *(Support) The support of a vector* $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{F}_q^n$ *in Hamming metric, denoted* $Supp_H(\mathbf{x})$ *is the set of its non-zero coordinates.*

$$Supp_H(\mathbf{x}) \triangleq \{i | \mathbf{x}_i \neq 0\}.$$

Definition 22. *(Hamming weight) The Hamming weight of a vector* $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{F}_q^n$*, denoted* w_H*, is the number of its non-zero coordinates.*

$$w_H(\mathbf{x}) \triangleq |\{i | \mathbf{x}_i \neq 0\}| = |Supp_H(\mathbf{x})|.$$

The Hamming weight is sometimes used to constrain some code families.Those constraints are interesting for different reasons. One of them is the existence of specific decoder algorithms, making the decoding a more straightforward step by using the specificities of the families of codes. For instance, some decoders use the syndrome to decode $\mathbf{y}$.

2.2.2 Decoding Methods

The decoding, defined as the process of recovering the codeword $\boldsymbol{c}$ from the message received $\mathbf{y}$, is more complex than the encoding process. It has even been proven to be NP-complete if some constraints are put on the weight of the error vector.

One method is principally used due to its high efficiency in decoding linear with up to t error: the syndrome decoding. It can be seen as an adaptation of the minimum distance decoding.

Minimum Distance Decoding

Given a $\mathbf{y} \in \mathbb{F}_q^n$, the minimum distance decoding method search for the nearest codeword $\boldsymbol{c}$ to $\mathbf{y}$, *i. e.* find the codeword such that Hamming distance of $\mathbf{y}$ and $\boldsymbol{c}$ is minimum. This method can be executed by using a standard array.

Standard array Let us start by defining two notions: the coset and the coset leader.

Definition 23. *(Coset) Let us have* $\mathcal{ER}$ *an equivalence relation defined on* $\mathbb{F}_q^n$ *as follows:*

$$\forall \mathbf{x}, \mathbf{y} \in \mathbb{F}_q^n, \mathbf{x} \ \mathcal{ER} \ \mathbf{y} \equiv \mathbf{y} - \mathbf{e} \in \mathscr{C}$$

The coset of $\mathbf{e} \in \mathbb{F}_q^n$ *is* $\mathbf{e} + \mathbf{x}$*.*

Definition 24. *(Coset leader) The coset leader is the coset with the smallest weight.*

The standard array is a q^{n-k} by q^n array, constructed as follows:

1. The first row lists all the codewords.

2. Each row is a coset with the coset leader in the first column.

3. The entry in the i-th row and j-th column is the sum of the i-th coset leader and the j-th codeword.

Once the standard array is built, we search for $\mathbf{y}$ in the cosets. The coset leader is, therefore, the error vector, and to recover the nearest codeword $\boldsymbol{c}$, we compute $\mathbf{y} - \mathbf{e}$.

Example 2. *Let us have $\mathscr{C}$ a binary [4,2]-code such that $\mathscr{C} = \{0000, 1011, 0101, 1110\}$. And $\mathbf{y} = \boldsymbol{c} + \mathbf{e} = (1001)$ the message received. We construct the standard array of $\mathscr{C}$, see Table 2.2.*

0000	1011	0101	1110
1000	0011	0101	0111
0100	1111	0001	1010
0010	1001	0111	1100

Figure 2.2: Standard array generated by $\mathscr{C}$.

$\mathbf{y}$ *belongs to the third coset and has as coset leader* (0010) *thus the codeword is* $\boldsymbol{c} = (1011)$.

The problem with this method is that the standard array's size increases drastically, which makes it inefficient.

Syndrome Decoding

The syndrome decoding is a minimum distance decoder but with a smaller array. In this method, only the coset leader is kept. The syndrome corresponding to the coset leader replaces the other cosets. The array is constructed as follows:

1. On its first row, there is the zero coset leader and its syndrome .

2. Each following row starts with a coset leader of a small weight for which the syndrome is not already in the array.

To recover $\boldsymbol{c}$ from $\mathbf{y}$, *i. e.* $\mathbf{y} \cdot H^T$ and searched the corresponding coset leader $\mathbf{e}$. $\boldsymbol{c}$ is simply $\boldsymbol{c} = \mathbf{e} + \mathbf{y}$.

Example 3. *Let us have $\mathscr{C}$ a binary [4,2]-code such that* $H = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}$. *And* $\mathbf{y} = \boldsymbol{c} + \mathbf{e} = (1001)$ *the received message. We construct the array of $\mathscr{C}$, see Table 2.3.* $\mathbf{s} = \mathbf{y} \cdot H^T = (10)$*, so the error vector is* (0010)*. Thus the codeword is* $\boldsymbol{c} = (1011)$.

The syndrome decoding is a generic decoder. Many other decoders use the syndrome, but they are adapted to a specific family of codes to improve efficiency.

0000	00
1000	11
0100	01
0010	10

Figure 2.3: Coset leader and syndrome associated array generated by $\mathscr{C}$.

2.2.3 Syndrome Decoding Problem

Explain as previously, the syndrome decoding seems simple. However, it is misleading as the syndrome decoding problem has been proven NP-complete by Berlekamp, McEliece, and van Tilbord [12], given a syndrome and a weight for the error vector.

Definition 25. *(Syndrome Decoding Problem (SD problem)) Given a parity-check matrix* $H \in \mathbb{F}_q^{(n-k)\times k}$*, a syndrome* $\mathbf{s} \in \mathbb{F}_q^{(n-k)}$ *and a weight* ω*. Find a vector* $\mathbf{e} \in \mathbb{F}_q^n$ *such that* $\mathbf{e} \cdot H^T = \mathbf{s}$ *and* $w_H(\mathbf{e}) \leq \omega$*.*

The SD problem and its variants are highly significant for code-based cryptography. Indeed, many proofs of the security of schemes rely on this problem. Moreover, some generics decoding algorithms are considered threats to the security of the schemes. For instance, the Information-Set Decoding.

2.2.4 Information-Set Decoding

The Information-Set Decoding (ISD) was introduced in 1962 by Prange [65]. It is a generic algorithm (can be used for any family of codes) to recover the error vector for a syndrome and a weight ω by using linear algebra. There are different versions of ISD to improve its efficiency [49, 50, 11]. However, we only introduce Prange's algorithm in this Subsection.

Prange's ISD

The idea is to pick a set $\mathcal{I}$ of i elements into the code $\mathscr{C}$ such that the set contains only positions for which the error vector $\mathbf{e} \in \mathbb{F}_q^n$ equals zeros. If $\{\mathbf{x}_i = 0 | i \in \mathcal{I}\}$ then there exist a unique $\mathbf{x}$ that solves the following linear system of equations:

$$\begin{cases} \mathbf{x}_{\mathcal{I}} = 0 \\ \mathbf{x} \cdot H^T = s \end{cases}.$$

If $w_H(\mathbf{x}) = \omega$ then it is $\mathbf{x}$ is the vector error $\mathbf{e}$.

Algorithm 1 formally describes the Prange's ISD algorithm.

The success of Prange's ISD depends on the number of possible sets for $\mathcal{I}$. Indeed, the more there is of possibilities, the harder it is to find one with only zero positions. The algorithm selects i positions of $\mathbf{e}$ without errors. It is equivalent to selecting $n-k$ positions

Algorithm 1: Prange's ISD

Input: H,$\mathbf{s}$,ω

Output: $\mathbf{e}$ s.t. $w_H(\mathbf{e}) = \omega$ and $\mathbf{e} \cdot H^T = \mathbf{s}$

$\mathbf{x} \leftarrow \{0\}^n$

while $w_h(\mathbf{x}) \neq \omega$ **do**

  Pick $\mathcal{I}$

  Solve $\begin{cases} \mathbf{x}_{\mathcal{I}} = 0 \\ \mathbf{x} \cdot H^T = s \end{cases}$

$\mathbf{e} \leftarrow \mathbf{x}$

of $\mathbf{e}$ containing the ω errors. However, the error vector $\mathbf{e}$, of length n and Hamming weight ω has $\binom{n}{\omega}$ possible combinations. Thus, the Prange's algorithm has up to $\binom{n-k}{\omega}$ possible selections on the $\binom{n}{\omega}$ combination of $\mathbf{e}$. Therefore, the probability of success of Prange's ISD is :

$$\alpha = \frac{1}{\frac{\binom{n-k}{\omega}}{\binom{n}{\omega}}} = \frac{\binom{n}{\omega}}{\binom{n-k}{\omega}}$$

We must consider the cost of linear algebra to get the Algorithm 1 complexity. It is not a negligible cost. However, it depends on the technique chosen to find $\mathbf{e}$. For instance, in our attack on BIKE, Chapter 7, we use a Gaussian elimination instead of a system of linear equations.

Prange's ISD in $\mathbb{F}_2$

In the attack on BIKE scheme, see Chapter 7, we adapted the Algorithm 1 as follows:

1. We select $n-k$ columns of H to create a square submatrix $H_{Prange} \in \mathbb{F}_2^{(n-k)\times(n-k)}$
2. We apply the Gaussian elimination on H_{Prange} and we obtain the matrix M_{Gauss}
3. If the kernel of M_{Gauss} equals ω, *i. e.* $ker(M_{Gauss}) = \omega$, then, we return $\mathbf{e}$ which is composed of $(ker(M_{Gauss})|0)$ reversed to the initial positions of the columns.

Figure 2.4 is a simplified visual representation of Prange ISD, which we used in Chapter 7.

In our version of Prange ISD, the linear algebra cost is equivalent to the cost of inverting a square $(n-k)$ matrix over $\mathbb{F}_2$,*i. e.* $(n-k)^\zeta$ bit operation. In our computation, we use $\zeta = 2.8$.

At the beginning of this chapter, we talked about the (7,4)-Hamming codes, which belong to a larger family of codes called the Hamming codes. Since their introduction by Hamming in 1950, new families of code based on the Hamming metric have subsequently

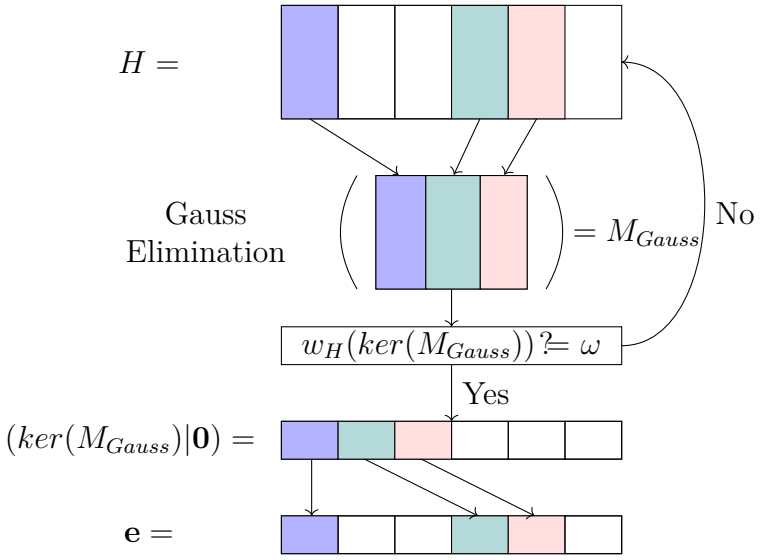


Figure 2.4: Prange's ISD for the attack on BIKE, Chapter 7

emerged, such as the Reed-Solomon codes, the Goppa codes, and so on. Each has its advantages and drawbacks, making them sometimes inappropriate for cryptography.

The next section focuses on the Quasi-Cyclic Moderate-Density Parity-Check codes as they are used on BIKE schemes.

2.2.5 Quasi-Cyclic Moderate-Density Parity-Check Codes

The Quasi-Cyclic Moderate-Density Parity-Check (QC-MDPC) code is a family of codes combining two other families with structural specificities. The first one is the Quasi-Cyclic codes for which the generator matrix G is generated thanks to a rotation of the first row. It is said to be quasi-cyclic and not cyclic because the first row is divided into at least two parts and rotated independently. They are creating at least two cyclic submatrices in G. The second family is the Moderate-Density Parity-Check codes. These codes have a weight constraint on the rows of their parity-check matrix H.

In code-based cryptography, the combination of the families has advantages in terms of security with the MDPC codes but also to reduce the size of the key with the QC codes. The binary QC-MDPC codes are formally introduced in the remainder of the subsection.

Quasi-Cyclic Codes

n_0 circulant matrices comprise the binary QC codes generator matrix. G is therefore a block-circulant matrix, *i. e.* a matrix composed of n_0 circulant matrices of same size r.

Definition 26. *(Circulant matrix) A circulant matrix is a square matrix in which all the rows are derived from the first row with a rotation of one position relative to the previous row.*

The QC code is defined as follows:

Definition 27. *(Quasi-Cyclic code) A $(n_0, k_0)-$QC code is a quasi-cyclic of length $n = n_0 r$ and dimension $k = k_0 r$ with a $k \times n$ block circulant matrix as generator matrix.*

Example 4. *Let us define a (2,1)-QC code with circulant matrices of length $r = 5$. Let have $h = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \end{pmatrix}$ the first row of G. Then, the generator matrix G is constructed as follows:*

$$G = \left(\begin{array}{ccccc:ccccc} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \end{array}\right)$$

By it is structure, a binary QC code can be represented only by the first row of its generator matrix. However, it can also be represented by (n_0) polynomials. Indeed, there exists a ring isomorphism ϕ between $r \times r$ circulant matrices and the quotient ring $\mathcal{R} = \mathbb{F}_2 \big/ (X^r - 1)$ that mapped the first row of a circulant matrix X to a polynomial as follows:

$$\phi : (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{r-1}) \mapsto x_0 + x_1 X + \ldots + x_{r-1} X^r$$

Hence, all matrix operations can be seen as polynomial operations. The transposition of a vector $\mathbf{x} = (\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{r-1})$ on a polynomial representation is defined as $x^{\mathbf{T}} = x_0 + x_{r-1} X + \ldots + x_1 X^{r-1}$.

Quasi-Cyclic Moderate-Density Parity-Check Codes

We explained in the previous subsection the notion of QC codes. Defining the notion of MDPC codes is necessary to obtain the QC-MDPC code. A binary Moderate-Density Parity-Check is a code that admits a sparse matrix as parity-check matrix H. The rows of H have a weight of order $\mathcal{O}(\sqrt{n})$.

Thus, the QC-MDPC codes are formally defined as follows:

Definition 28. *(Quasi-Cyclic Moderate-Density Parity-Check code) An (n_0, k_0, r, ω)-QC-MDPC code is an (n_0, k_0) quasi-cyclic code of length $n = n_0 r$, dimension $k = k_0 r$, order r with a parity-check matrix with constant row weight $\omega = \mathcal{O}(\sqrt{n})$*

In the case of a $(2, 1)$-QC-MDPC code generated by two polynomials $(\mathbf{h}_0, \mathbf{h}_1) \in \mathcal{R}$, the generator matrix is therefore construct as follows:

Shorten into $G = (\mathbf{h}_1 | \mathbf{h}_0)$. Therefore, the parity check matrix is defined by $H = (\mathbf{h}_0^{\mathbf{T}} | \mathbf{h}_1^{\mathbf{T}})$. Even though we defined the generator in this subsection, the QC-MDPC codes are, in practice, referred to by their parity-check matrix.

$$G = \left(\begin{array}{c|c} h_1 & h_0 \\ h_1X & h_0X \\ h_1X^2 & h_0X^2 \\ \vdots & \vdots \\ h_1X^{r-1} & h_0X^{r-1} \end{array}\right).$$

Example 5. *Let defined a binary (2,1,5,3)-QC-MDPC code $\mathscr{C}$. Let have $h_0 = X^4+X^2+X$ and $h_1 = X^3+X^1+1$, the two polynomials generator of the code $\mathscr{C}$. Then, the parity-check matrix H is constructed as follows:*

$$H = \left(\begin{array}{ccccc|ccccc} 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{array}\right)$$

In this work, we do not introduce the different decoders for the QC-MDPC. Two of them are explained in the description of the BIKE scheme, Section 4.2.

Quasi-Cyclic Syndrome Decoding Problem

In cryptography, the security of schemes based on QC-MDPC codes relies on variants of the general syndrome decoding problem.

In the schemes, the two polynomials h_0 and h_1, generating the (2,1,r,ω)-QC-MDPC codes, are the private key. The public key h is $h_1 * h_0^{-1}$, and the parity-check matrix H is constructed on a systematic form with the circulant matrix generated by $\mathbf{h}$ on the right side. However, we are not supposed to recover h_0 and h_1 from h. So here comes the first problem: the codeword finding problem.

Problem 1. *((2,1,r,ω)-QC Codeword Finding (QCCF)) Given $h \in \mathcal{R}$ and ω such that $\frac{\omega}{2}$ is odd. Find $(\mathbf{h}_0, \mathbf{h}_1) \in \mathcal{R}^2$ such that $w_H(\mathbf{h}_0) = w_H(\mathbf{h}_1) = \frac{\omega}{2}$ and $h_1 + h_0 * h = 0$.*

The QCCF problem is a variant of the SD problem as it uses the propriety of the syndrome equals zero if and only if the $(\mathbf{h}_1|\mathbf{h}_0)$ is a codeword. Thus, instead of looking for the error vector with a certain weight, we look directly for a codeword. And if we find one codeword, then we obtain the private key as the code is composed of rotation of $\mathbf{h}_0$ and $\mathbf{h}_1$.

The second variant for QC-code is more evident but required to define the error vector. In the schemes, the error vector is decomposed in two polynomials e_0 and e_1 such that $\mathbf{e} = (\mathbf{e}_0, \mathbf{e}_1)$. So, the syndrome decoding problem is described as follows:

Problem 2. *((2,1,r,ω)-QC Syndrome Decoding (QCSD)) Given $h \in \mathcal{R}$, $s \in \mathcal{R}$ and ρ an integer greater than 0. Find $(\mathbf{e}_0, \mathbf{e}_1) \in \mathcal{R}^2$ such that $w_H(e_0)+w_H(e_1) = \rho$ and $\mathbf{e}_0+\mathbf{e}_1*h = s$.*

One interesting point about QC-MDPC codes is that there is a variant of these codes but in Rank metric.

2.3 Rank Metric

In this section, we consider the finite field $\mathbb{F}_{q^m}$.

2.3.1 Generalities

In rank metric, the distance between two codewords belonging to the $\mathbb{F}_{q^m}$-linear code $\mathscr{C}$ of length n is given by the rank of their differences. However, a codeword of $\mathscr{C}$ is a vector in $\mathbb{F}_{q^m}^n$, so the rank cannot computed directly.

To obtain the rank of a vector, a matrix $X \in \mathbb{F}_p^{m\times n}$ associated to the vector $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \in \mathbb{F}_{q^m}^n$ is created. We define $\beta = (\beta_1, \beta_2, \ldots, \beta_m) \in \mathbb{F}_{q^m}^m$ a basis of $\mathbb{F}_q^m$. The coordinate $\mathbf{x}_j$ is associated to vector $\mathbf{x}_j = (\mathbf{x}_{j,1}, \ldots, \mathbf{x}_{j,m}) \in \mathbb{F}_q^m$ such that $x_j = \sum_{i=1}^m x_{j,i}\beta_i$. The matrix is, therefore, constructed as follows:

$$X = \begin{pmatrix} \mathbf{x}_{1,1} & \mathbf{x}_{2,1} & \cdots & \mathbf{x}_{n,1} \\ \mathbf{x}_{1,2} & \mathbf{x}_{2,2} & \cdots & \mathbf{x}_{n,2} \\ \vdots & \vdots & \vdots & \vdots \\ \mathbf{x}_{1,m} & \mathbf{x}_{2,m} & \cdots & \mathbf{x}_{n,m} \end{pmatrix}$$

$\mathbf{M}(\mathbf{x})$ returns the associated matrix of $\mathbf{x}$. The rank of the matrix associated X of $\mathbf{x}$ gives the rank of $\mathbf{x}$.

Definition 29. *(Rank weight) Let* $\mathbf{x}$ *be a vector in* $\mathbb{F}_{q^m}^n$*. The rank weight* $w_R(\mathbf{x})$ *of* $\mathbf{x}$ *is defined by:*

$$w_R(\mathbf{x}) \triangleq Rank(\mathbf{M}(\mathbf{x})).$$

The rank distance between two vectors in $\mathbb{F}_{q^m}^n$ is defined as follows:

Definition 30. *(Rank distance) Let* $\mathbf{x}$ *and* y *be two vectors in* $\mathbb{F}_{q^m}^n$*. The rank distance* $d_R(\mathbf{x}, \mathbf{y})$ *between* $\mathbf{x}$ *and* $\mathbf{y}$ *is:*

$$d_R(\mathbf{x}, \mathbf{y}) \triangleq w_R(\mathbf{x} - \mathbf{y}).$$

Similarly to the Hamming metric, the rank metric has the notion of support. Nevertheless, in the latter, the support of $\mathbf{x}$ does not contain the coordinates of the non-zero elements $\mathbf{x}$ but the $\mathbb{F}_q$-subspace of $\mathbb{F}_{q^m}$ by the coordinates. It is formally defined as follows:

Definition 31. *(Support) Let* $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ *be a vector in* $\mathbb{F}_{q^m}^n$*. The support* $Supp_R(\mathbf{x})$

of $\mathbf{x}$ *is the* $\mathbb{F}_q$*-subspace of* $\mathbb{F}_{q^m}$ *generated by the coordinates of* $\mathbf{x}$*:*

$$Supp_R(\mathbf{x}) \triangleq \langle \mathbf{x}_1, \ldots, \mathbf{x}_n \rangle_{\mathbb{F}_q}$$

and we have $dim(Supp_R(\mathbf{x})) = w_R(\mathbf{x})$*.*

Example 6. *Let* $\gamma \in \mathbb{F}_{2^3}$ *such that* $\gamma^3 = \gamma + 1$*. Let have a vector* $\mathbf{x}$ *over* $\mathbb{F}_{2^3}^3$ *such that* $\mathbf{x} = (\gamma, \gamma^3, 1)$*. So,* $\beta = (1, \gamma, \gamma^2)$ *is a basis of* $\mathbb{F}_{2^3}$*. The rank weight of* $\mathbf{x}$ *is :*

$$w_R(\mathbf{x}) = Rank\left(\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}\right) = 2.$$

And the support of $\mathbf{x}$ *is:*

$$Supp_R(\mathbf{x}) = \langle 1, \gamma \rangle_{\mathbb{F}_2}.$$

The support can be used to put some constraints during the generation of the codes, especially for the ones used in cryptography.

2.3.2 Rank Syndrome Decoding Problem

In rank metric as well, looking for the closest codeword to a given vector is a complex problem. It exists decoders specific to families of code and some generic algorithms that can be executed in order to retrieve the closest codeword to a vector. But, the problem stays hard enough to ensure cryptographic security. Especially the rank syndrome decoding problem, which has been proven difficult with a probabilistic reduction to the Hamming [38]. The two variants of the syndrome decoding problem in rank metric are the computational and the decisional.

Problem 3. *(Computational Syndrome Decoding (CSD)) Given* $H \in \mathbb{F}_{q^m}^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_{q^m}^{n-k}$*, and an integer* $\omega > 0$*. Compute* $\mathbf{x} \in \mathbb{F}_{q^m}^n$ *such that* $\mathbf{y} \cdot H^T = \mathbf{s}$ *and* $w_R(\mathbf{x}) = \omega$*.*

Problem 4. *(Decision Syndrome Decoding (DSD)) Given* $H \in \mathbb{F}_{q^m}^{(n-k)\times n}$, $\mathbf{s} \in \mathbb{F}_{q^m}^{n-k}$ *and an integer* $\omega > 0$*. Decide with non-negligible advantage whether* $\mathbf{s}$ *is from* $H \cdot \mathbf{x}^T$ *with* $w_R(\mathbf{x}) = \omega$ *or* $\mathbf{s}$ *is from a uniform distribution over* $\mathbb{F}_{q^m}^{n-k}$*.*

The CSD problem is the syndrome decoding problem described for Hamming metric, see subsection 2.2, but in rank metric. In contrast, DSD describes the difficulty of distinguishing a vector generated from another one with a constraint of the weight and a vector uniformly generated in a specific space.

The problem of syndrome decoding is not infallible to attacks or generic algorithms. Indeed, it does exist two types of attacks: the combinatorial ones and the algebraic ones.

Combinatorial attacks Combinatorial attacks are used to find the support of the vector error or the support of the codeword. The best attack known is an adaptation of the Information-Set Decoding in Hamming metric [37, 5]. The principle is simple; the attacker tries to find a subspace that contains the support and then solves a linear system of equations created from the parity-check matrix to check its choice.

Remark 5. *The Information-Set Decoding algorithm cannot be used for rank metric. However, the GRS algorithm is an equivalence of ISD for this metric.*

Algebraic attacks The algebraic attack takes the rank syndrome decoding instance and writes it as a system of equations. If there is a solution to the system, then it is a solution for the RSD instance. The algebraic attacks are severe threats to the security of the schemes. For example, the Gabidulin codes are particularly sensitive to this type of attack due to their strong algebraic structure.

The Ideal-LRPC codes are less sensitive to these attacks. Even though the more recent algebraic attack using Gröbner basis reduces their security if the parameters are not carefully chosen.

2.3.3 Ideal Low-Rank Parity-Check Codes

In the same way as the QC-MDPC codes in the Hamming metric, the Ideal Low-Rank Parity-Check codes combine two families of code, the Ideal codes, and the LRPC codes, to create a new family of code interesting for the cryptography schemes.

Ideal Codes

The Ideal codes have a structure similar to the QC-cyclic codes (explain is Subsection 2.2.2) except that an ideal matrix replaces the circulant matrix. Before explaining the notion of ideal matrix, let us define the notion of ideal of the polynomial ring $\mathbb{F}_q[X]$.

Definition 32. *(Ideal) An ideal is a subset I of the polynomial ring $\mathbb{F}_q[X]$ such that:*

1. *if $f \in \mathbb{F}_q[x]$ and $g \in I$ then $f * g \in I$;*
2. *if $g, h \in I$ then $g + h \in I$.*

Given a subset $F \in \mathbb{F}_q[X]$, we denoted $\langle F \rangle$ the smallest ideal containing F. It is the ideal generated by F.

Let us define $P \in \mathbb{F}_q[X]$ a polynomial of degree n. P generates the ideal $\langle P \rangle$ of $\mathbb{F}_{q^m}[X]$. Thus, there exists Ψ, a transformation from the vector space $\mathbb{F}_{q^m}^n$ to the polynomial ring $\mathbb{F}_{q^m}[X] \Big/ \langle P \rangle$, defines as follows:

$$\Psi: \quad \mathbb{F}_{q^m}^n \simeq \mathbb{F}_{q^m}[X] \Big/ \langle P \rangle$$
$$(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}) \mapsto \sum_{i=0}^{n-1} x_i X^i$$

The product of the two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{F}_{q^m}^n$ is the same as the multiplication in $\mathbb{F}_{q^m}[X] \Big/ \langle P \rangle$, $\Psi(\mathbf{x})\Psi(\mathbf{y})$. To lighten the writing, we will omit the symbol Ψ. Let us define the ideal matrix.

Definition 33. *(Ideal matrix) Let $P \in \mathbb{F}_{q^m}[X]$ a polynomial of degree n and $\mathbf{y} \in \mathbb{F}_{q^m}^n$. The ideal matrix $\mathcal{IM}(\mathbf{y})$ generated by $\mathbf{y}$ is the $n \times n$ square matrix of the form:*

$$\mathcal{IM}(\mathbf{y}) = \begin{pmatrix} y & \mod P \\ y * X & \mod P \\ & \vdots \\ y * X^{n-1} & \mod P \end{pmatrix}$$

Contrary to the generator matrices of QC codes, the generator matrix of an ideal code is defined in its systematic form with blocks of ideal matrices of the right size. The ideal codes are formally defined as:

Definition 34. *(Ideal-codes) Let $P \in \mathbb{F}_q[X]$ be a polynomial of degree r. A (n_0, k_0)-ideal code is an ideal code of length $n = n_0 r$ and dimension $k = k_0 r$ with the following as generator matrix:*

$$G = \begin{pmatrix} & \mathcal{IM}(\mathbf{g}_{1,1}) & \cdots & \mathcal{IM}(\mathbf{g}_{1,n_0-k_0}) \\ I_k & \vdots & \ddots & \vdots \\ & \mathcal{IM}(\mathbf{g}_{k_0,1}) & \cdots & \mathcal{IM}(\mathbf{g}_{k_0,n_0-k_0}) \end{pmatrix}$$

where $\mathbf{g}_{i,j}$, for $i \in [1, \ldots, k_0]$ and $j \in [1, \ldots, n_0 - k_0]$, are vectors $\mathbb{F}_{q^m}^n$. We said that $\mathscr{C}$ is generated by the $(\mathbf{g}_{i,j})$.

In our works, k_0 always equals 1. Thus, to simplify the next explanations, we will only consider the $(n_0, 1)-$Ideal codes. Let define a code $\mathscr{C}$ generated by $(\mathbf{g}_1, \ldots, \mathbf{g}_{n_0-1})$ such that :

$$\mathscr{C} = \{(\mathbf{u}, \mathbf{u}\mathbf{g}_1, \ldots, \mathbf{u}\mathbf{g}_{n_0-1}), \mathbf{u} \in \mathbb{F}_{q^m}^r\}.$$

With, as generator matrix, the matrix G defines as follows:

$$G = \begin{pmatrix} I_k & \mathcal{IM}(\mathbf{g}_1) & \cdots & \mathcal{IM}(\mathbf{g}_{n_0-1}) \end{pmatrix}$$

As G is in systematic form, the parity-check matrix can be constructed as follows:

$$H = \begin{pmatrix} (\mathcal{IM}(\mathbf{g}_1))^T & \\ \vdots & I_{r(n_0-1)} \\ (\mathcal{IM}(\mathbf{g}_{n_0-1}))^T & \end{pmatrix}$$

However, the parity-check matrix can also be in systematic form, which is the matrix form used in ROLLO and RQC, two schemes relying on ideal codes. It is easy to go from H in its previous form to its systematic form with a co-prime propriety of the elements in $\mathbb{F}_{q^m}[X] \Big/ \langle P \rangle$.

Lemma 1. *Let m and r be two different prime numbers. Let $P \in \mathbb{F}_q[X]$ be an irreducible polynomial of degree r and $U \in \mathbb{F}_{q^m}$ a non zero polynomial of degree at most $r-1$. Then P and U are co-prime in $\mathbb{F}_{q^m}[X]$.*

Thus, given any $\mathbf{g}_i \in \mathbb{F}_{q^m}^r$ generator of $\mathscr{C}$, there exists a vector $\mathbf{h}_i \in \mathbb{F}_{q^m}^r$ such that according to the lemma:

$$\begin{aligned} g_i * h_i = 1 \mod P \\ \equiv \mathbf{g}_i \cdot \mathcal{IM}(\mathbf{h}_i) = (1, 0, \ldots, 0) \\ \equiv \mathcal{IM}(\mathbf{g}_i)\mathcal{IM}(\mathbf{h}_i) = I_r \end{aligned}$$

The systematic matrix is, therefore :

$$H = \begin{pmatrix} & & \mathcal{IM}(\mathbf{h}_1) \\ & I_{r(n_0-1)} & \vdots \\ & & \mathcal{IM}(\mathbf{h}_1) \end{pmatrix}$$

Example 7. *Let $P = X^5 + X^3 + 1$ a polynomial of degree $r = 5$ in $\mathbb{F}_2[X]$. Let set $m = 3$ and $n_0 = 2$, $\mathbb{F}_{2^m}^r$ such that $\gamma^3 = \gamma + 1$. Let us define $\mathbf{g}_1 = (\gamma^6, \gamma^2, \gamma, \gamma^4, 1) \in \mathbb{F}_{2^m}^r$ generating the ideal code $\mathscr{C}$ with the following generator matrix:*

$$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \gamma^6 & \gamma^2 & \gamma & \gamma^4 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & \gamma^6 & \gamma^2 & \gamma & \gamma^4 \\ 0 & 0 & 1 & 0 & 0 & \gamma^4 & 1 & \gamma^6 & \gamma^2 & \gamma \\ 0 & 0 & 0 & 1 & 0 & \gamma & \gamma^4 & 1 & \gamma^6 & \gamma^2 \\ 0 & 0 & 0 & 0 & 1 & \gamma^2 & \gamma & \gamma^4 & 1 & \gamma^6 \end{pmatrix}$$

The polynomial g_1 as for transpose vector $\mathbf{g}_1^T = (\gamma^6, 1, \gamma^4, \gamma, \gamma^2)$ such that the parity-check

matrix is :

$$H = \begin{pmatrix} \gamma^6, 1, \gamma^4, \gamma, \gamma^2 & 1 & 0 & 0 & 0 & 0 \\ \gamma^6, 1, \gamma^4, \gamma, \gamma^2 & 0 & 1 & 0 & 0 & 0 \\ \gamma^6, 1, \gamma^4, \gamma, \gamma^2 & 0 & 0 & 1 & 0 & 0 \\ \gamma^6, 1, \gamma^4, \gamma, \gamma^2 & 0 & 0 & 0 & 1 & 0 \\ \gamma^6, 1, \gamma^4, \gamma, \gamma^2 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

There exist a polynomial inverse h_1 of g_1. $\mathbf{h}_1 = (0, \gamma^4, \gamma^2, \gamma^5, 1)$

Its parity-check matrix in systematic form is the following :

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \gamma^4 & \gamma^2 & \gamma^5 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & \gamma^4 & \gamma^2 & \gamma^5 \\ 1 & 0 & 1 & 0 & 0 & \gamma^5 & 1 & 0 & \gamma^4 & \gamma^2 \\ 1 & 0 & 0 & 1 & 0 & \gamma^2 & \gamma^5 & 1 & 0 & \gamma^4 \\ 1 & 0 & 0 & 0 & 1 & \gamma^4 & \gamma^2 & \gamma^5 & 1 & 0 \end{pmatrix}$$

Ideal Rank Syndrome Decoding Problem

The ideal codes are helpful in code-based cryptography as they reduce the size of the keys by their structure. Nevertheless, the problems on which the security is based need to be rewritten according to the specificity of the code.

Problem 5. *(Computational Ideal Syndrome Decoding (C-IRSD)) Let $P \in \mathbb{F}_q[X]$ an irreducible polynomial of degree r. Given positive integers r, ω, n_0 a random parity check matrix H under systematic form of an n_0-ideal code $\mathscr{C}$ and a random vector $\mathbf{s} \in \mathbb{F}_{q^m}^r$. Compute $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_{n_0}) \in \mathbb{F}_{q^m}^{n_0 r}$ such that $w_R(\mathbf{x}) = \omega$ and $\mathbf{s} = \mathbf{x} * H^T$.*

Problem 6. *(Decision Ideal Syndrome Decoding (D-IRSD)) Let $P \in \mathbb{F}_q[X]$ an irreducible polynomial of degree r. Given positive integers r, ω, n_0 a random parity check matrix H under systematic form of an n_0-ideal code $\mathscr{C}$ and a random vector $\mathbf{s} \in \mathbb{F}_{q^m}^r$. Decide with non-negligible advantage whether $\mathbf{s}$ is from $H \cdot \mathbf{x}^T$ with $w_R(\mathbf{x}) = \omega$ or $\mathbf{s}$ is from a uniform distribution over $\mathscr{S}(r, n_0)$, the set of the parity-check matrices H under systematic form of n_0-ideal codes.*

There exists a third problem defined for the ideal codes. However, this time, it is not on the syndrome decoding problem but on the possibility of recovering the rank support.

Problem 7. *(n_0-Ideal Syndrome Decoding (n_0-IRSD)) Let $P \in \mathbb{F}_q[X]$ an irreducible polynomial of degree r. Given a vector $\mathbf{h} = (\mathbf{h}_1, \ldots, \mathbf{h}_{n_0-1}) \in \mathbb{F}_{q^m}^r$, a syndrome s and a weight ω. Recover a support E of dimension lower than ω such that $e_0 + e_1 * h_1 + \ldots + e_{n_0} * h_{n_0-1} = s(\mod P)$ where the vectors $\mathbf{e}_i$ are of support E.*

Ideal Low-Rank Parity-Check Codes

The ideal codes are combined with the LRPC codes to form the ideal LRPC codes. The advantage of the LRPC codes is their weak algebraic structure. Indeed, the LRPC codes are defined by their parity-check matrix H such that the rank of H is small. In other words, the support of H has a small dimension.

Definition 35. *(Low-Rank Parity-Check codes) Let $H \in \mathbb{F}_{q^m}^{(n-k)\times n}$ a full-rank matrix such that its coefficients $h_{i,j}$ for $i \in [1, n-k]$ and $j \in [1, n]$, generate an $\mathbb{F}_q$-subspace F of small dimension d:*

$$F = \langle h_{i,j} \rangle_{\mathbb{F}_q}$$

Let $\mathscr{C}$ be the code with the parity-check matrix H. By definition, $\mathscr{C}$ is an $[n, k]$-LRPC code.

The $(2, 1)-$ideal LRPC codes, used in ROLLO and RQC codes, are formally defined as follows:

Definition 36. *(Ideal LRPC) Let F be a $\mathbb{F}_q$-subspace of dimension d of $\mathbb{F}_{q^m}$, $(\mathbf{h}_0, \mathbf{h}_1)$ two vectors of $\mathbb{F}_{q^m}^r$ of support F and $P \in \mathbb{F}_q[X]$ a polynomial of degree r. Let*

$$H = \begin{pmatrix} \mathcal{IM}(\mathbf{h}_0) & \mathcal{IM}(\mathbf{h}_1) \end{pmatrix}$$

By definition, the code $\mathscr{C}$ with parity check matrix H is an $(2, 1)$-ideal LRPC code.

In cryptographic schemes, only the systematic form of H of an ideal LRPC code is public. The initial structure is hidden, and it is hard to determine whether h was generated uniformly at random or came from h_0 and h_1.

Problem 8. *(Ideal LRPC codes indistinguishability) Let $P \in \mathbb{F}_q[X]$ an irreducible polynomial of degree r. Given a vector $\mathbf{h} \in \mathbb{F}_{q^m}^r$ and a weight d. Distinguish whether the ideal code C with the parity-check matrix generated by $\mathbf{h}$ and P is a random ideal code or an ideal-LRPC code of weight d.*

Example 8. *Let $P = X^5 + X^3 + 1$ a polynomial of degree $r = 5$ in $\mathbb{F}_2[X]$. Let set $m = 3$, $n_0 = 2$, $d = 2$, $\mathbb{F}_{q^m}$ such that $\gamma^3 = \gamma + 1$, and $F = \langle \gamma^2, 1 \rangle$. Let us define $\mathbf{h}_0 = (\gamma^6, \gamma^2, 1, \gamma^2, 1) \in \mathbb{F}_{2^m}^r$ and $\mathbf{h}_1 = (1, \gamma^6, 1, \gamma^6, \gamma^2) \in \mathbb{F}_{2^m}^r$ generating the ideal-LRPC code $\mathscr{C}$ with the following parity-check matrix:*

$$H = \begin{pmatrix} \gamma^6 & \gamma^2 & 1 & \gamma^2 & 1 & 1 & \gamma^6 & 1 & \gamma^6 & \gamma^2 \\ 1 & \gamma^6 & \gamma^2 & 1 & \gamma^2 & \gamma^2 & 1 & \gamma^6 & 1 & \gamma^6 \\ \gamma^2 & 1 & \gamma^6 & \gamma^2 & 1 & \gamma^6 & \gamma^2 & 1 & \gamma^6 & 1 \\ \gamma^2 & 1 & \gamma^6 & \gamma^2 & 1 & 1 & \gamma^6 & \gamma^2 & 1 & \gamma^6 \\ \gamma^2 & 1 & \gamma^2 & 1 & \gamma^6 & \gamma^6 & 1 & \gamma^6 & \gamma^2 & 1 \end{pmatrix}$$

Let $h = h_1 * h_0^{-1} = X^4 + \gamma X^3 + \gamma^5 X^2 + \gamma^3 X + \gamma^4$ *and the parity-check matrix on its systematic form:*

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \gamma^4 & \gamma^3 & \gamma^5 & \gamma & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & \gamma^4 & \gamma^3 & \gamma^5 & \gamma \\ 0 & 0 & 1 & 0 & 0 & \gamma & 1 & \gamma^4 & \gamma^3 & \gamma^5 \\ 0 & 0 & 0 & 1 & 0 & \gamma^5 & \gamma & 1 & \gamma^4 & \gamma^3 \\ 0 & 0 & 0 & 0 & 1 & \gamma^3 & \gamma^5 & \gamma & 1 & \gamma^4 \end{pmatrix}$$

The Ideal-LRPC code has a decoding algorithm called the Rank Support Recovering algorithm, explained later in Subsection 4.1.2.

2.4 Original Schemes

In cryptography, the first schemes were in Hamming metric.

2.4.1 McEliece Encryption

Robert McEliece proposed in 1978 the first PKE based on error-correcting codes. The idea in the scheme is to randomly generate a code $\mathscr{C}$ in a family with an efficient decoding algorithm able to decode up to t errors. The encryption process is simple: Bob encodes a message $\boldsymbol{m}$ with a matrix $\hat{\mathrm{G}}$ and adds a random vector $\mathbf{z}$ of weight t to $\boldsymbol{m}$. With the decryption process, Alice finds the error and recovers the message. It looks like a classical encoding principle. However, $\hat{\mathrm{G}}$ is not the matrix generator matrix of $\mathscr{C}$ but derivation from the matrix generator G such that G cannot be reconstructed from $\hat{\mathrm{G}}$. Let us formally describe the three algorithms.

Key Generation(n,k,t):

1. Randomly generate a code $\mathscr{C}$ of length n, dimension k with a capacity of correction t.
2. Generate the generator matrix G associated to $\mathscr{C}$
3. Randomly generate an invertible $k \times k$ binary matrix $\dot{S}$
4. Randomly generate a permutation $n \times n$ matrix $\dot{P}$
5. Compute $\hat{\mathrm{G}}$ such that $\hat{\mathrm{G}} = \dot{S}G\dot{P}$
6. Return the public key $(\hat{\mathrm{G}}, t)$ and the private key $(\dot{S}, \dot{P}, \mathit{Decoder})$ where $\mathit{Decoder}$ is the decoder of the family of code of $\mathscr{C}$.

Encryption(m,($\hat{\mathrm{G}}$,t)):

1. Compute the vector $\mathbf{c}' = m \cdot \hat{\mathrm{G}}$
2. Generate a random binary vector $\mathbf{z}$ of length n and Hamming weight t
3. Compute the ciphertext $\mathbf{c} = \mathbf{c}' + \mathbf{z}$

Decryption($c, (\dot{S}, \dot{P}, Decoder)$):

1. Compute this inverse of $\dot{P}$
2. Compute $\mathbf{c}^\star = \mathbf{c} \cdot \dot{P}^{-1}$
3. Decode $\mathbf{c}^\star$ with the decoder, $m^\star = Decoder(\mathbf{c}^\star)$.
4. Compute the message $m = m^\star \cdot \dot{S}^{-1}$

Figure 2.5 visually represents the McEliece PKE.

The McEliece scheme was first proposed with binary Goppa codes. Nonetheless, variants with different types of codes were also proposed but proven less secure than the original code. However, the binary Goppa codes have the disadvantage of generating large-size keys, which are difficult to use in practice.

However, the McEliece scheme has a definite advantage in terms of execution times of the encryption and the decryption.

2.4.2 Niederreiter Scheme

In 1986, Harald Niederreiter proposed a variant of the McEliece scheme. The idea is the same but uses the parity check-parity matrix instead of the generator matrix. With the same level of security, the Niederreiter scheme is faster in execution time than McEliece. The three algorithms composing Niederreiter PKE are the following ones:

Key Generation(n, k, t):

1. Randomly generate a code $\mathscr{C}$ of length n, dimension k with a capacity of correction t.
2. Generate the $(n-k) \times n$ parity-check matrix H associated to $\mathscr{C}$
3. Randomly generate an invertible $(n-k) \times (n-k)$ binary matrix $\dot{S}$
4. Randomly generate a permutation $n \times n$ matrix $\dot{P}$
5. Compute $\hat{\mathrm{H}}$ such that $\hat{\mathrm{H}} = \dot{S}H\dot{P}$
6. Return the public key $(\hat{\mathrm{H}}, t)$ and the private key $(\dot{S}, \dot{P}, Decoder)$ where $Decoder$ is the decoder of the family of code of $\mathscr{C}$.

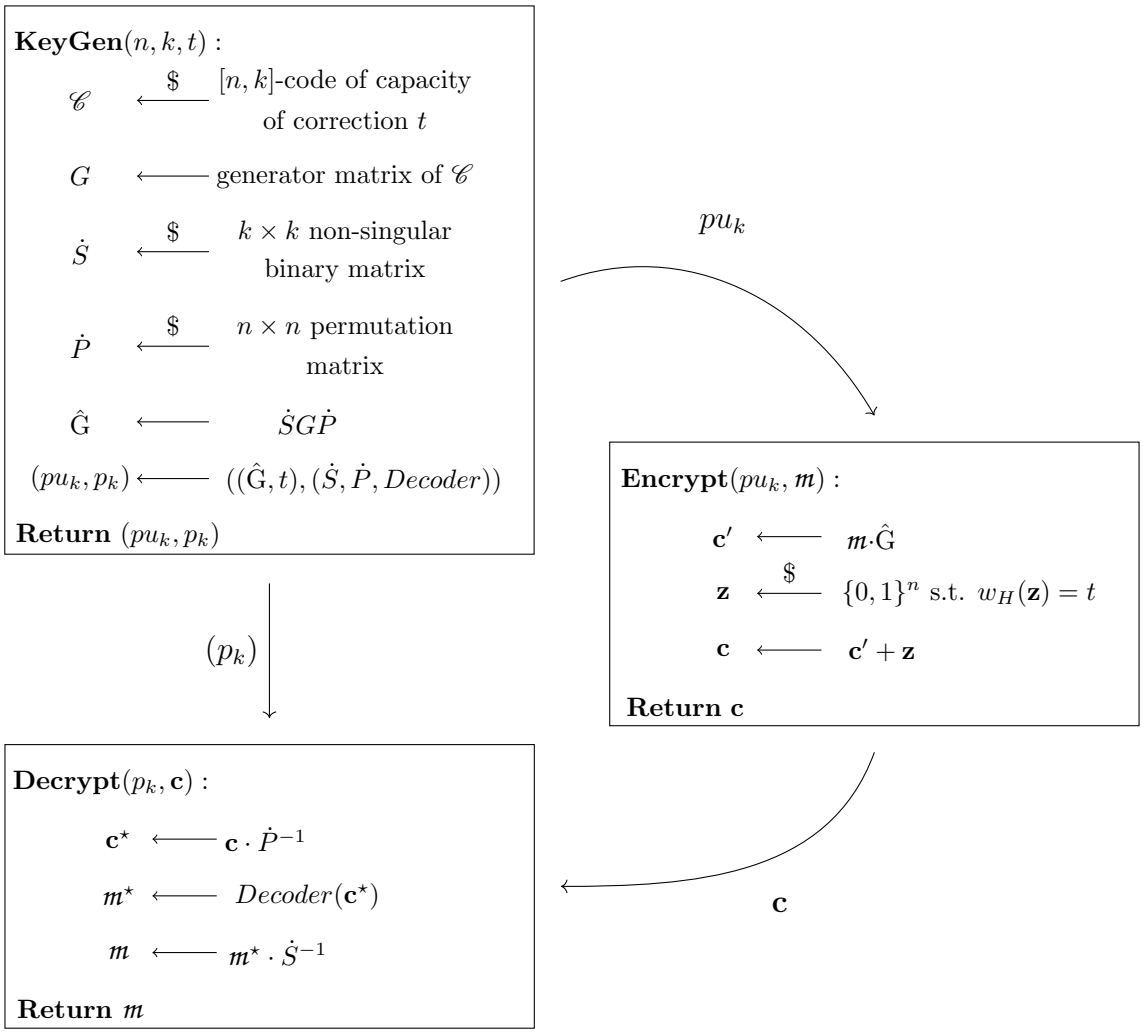


Figure 2.5: McEliece public-key encryption

Encryption(m,($\hat{\mathbf{H}}$,t)):

1. Encode the message m into a binary vector z of length n and Hamming weight at most t.
2. Compute the ciphertext $\mathbf{c} = \hat{\mathrm{H}} \cdot \mathbf{z}^T$

Decryption(c, $(\dot{S}, \dot{P}, Decoder)$):

1. Compute $\mathbf{s} = \dot{S}^{-1} \cdot \mathbf{c}$
2. Decode $\mathbf{s}$ with the syndrome decoder $Decoder$ to recover $\mathbf{z}^\star = \dot{P} \cdot \mathbf{z}^T$.
3. Compute $\mathbf{z} = \dot{P}^{-1} \cdot (\mathbf{z}^\star)^T$
4. Decode $\mathbf{z}$ to recover m

Figure 2.6 visually represents the Niederreiter PKE.

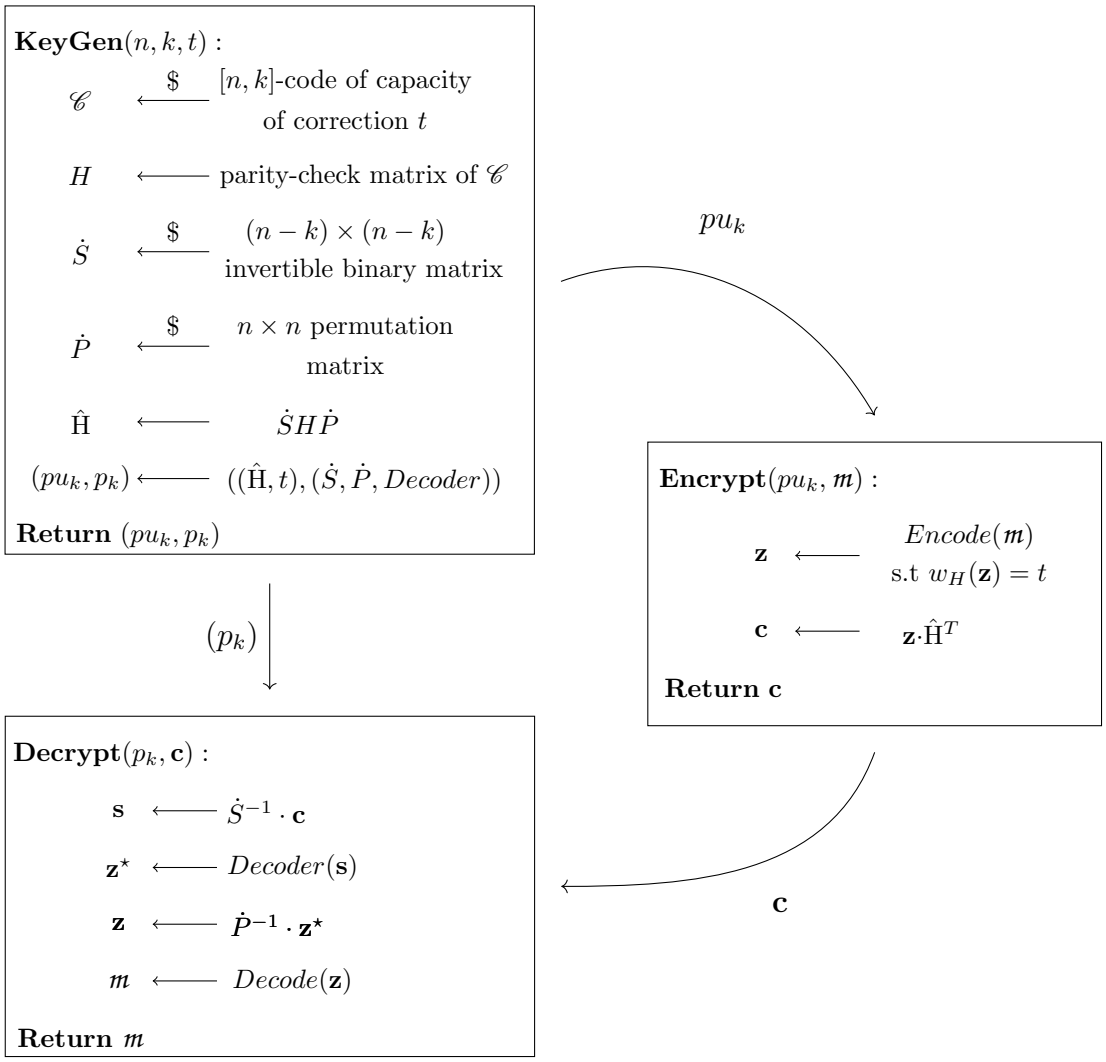


Figure 2.6: Niederreiter public-key encryption

The original Niederreiter scheme proposal was broken [17]. However, the scheme remains secure with some other families of codes, such as the binary Goppa codes.

Even today, McEliece and Niederreiter remain the benchmarks for creating new schemes like the ones proposed at the NIST post-quantum standardization process.

Chapter 3

Side-Channel Attacks

3.1 Definition

Side-channel attacks concern the security of the cryptographic algorithms once there are written in code and executed.The general idea is to extract data threatening the security of the cryptosystem from physical information inherent to the execution of a program, such as its execution time. Different types of threats exist, such as message or key recovery attacks. The side channel alone can be enough to attack, but sometimes, it is just additional information to reduce the difficulty of mathematical attacks. Also, the data can be directly related to the sensitive elements or intermediate values. Nevertheless, the side-channel attacks do not threaten the problems on which the cryptography schemes rely. Side-channel attack is formally defined as follows.

Definition 37. *(Side-channel attack) An attack enabled by leakage of information from a physical cryptosystem. Characteristics that could be exploited in a side-channel attack include timing, power consumption, and electromagnetic and acoustic emissions*

Side-channel attacks require some access to the device, less or more complicated. For example, a timing attack requires monitoring the time of execution, which implies a program to get the information, but it is possible to execute it at a distance. An attack by power consumption requires obtaining the power consumption of the devices with an oscilloscope, which is challenging to manage by distance.

A side-channel attack does not interfere with the cryptosystem during its execution. It is only listening; side-channel attacks are said to be non-invasive. Therefore, by definition, fault injection attacks, even though there are physical attacks, are not side-channel attacks. Nevertheless, the attacker can send forged plaintext or ciphertext to the cryptographic algorithm to observe a specific behavior.

3.2 Side-Channel Attacks Types

The side-channel attacks were first introduced by Paul Kocher in 1996 with the timing attack on RSA [47].

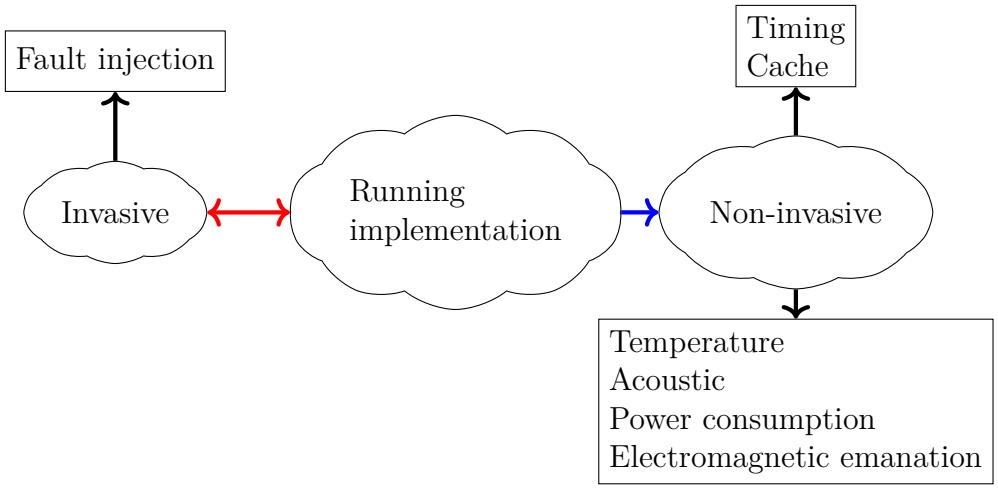


Figure 3.1: Physical attacks.

3.2.1 Timing Attack

The principle behind timing attacks is simple. The attacker records the time to execute a cryptographic algorithm and extract information on a sensitive element. Let us give an example.

Example 9. *Let us run an algorithm to check a four-digit code pin. The algorithm checks the digits one at the ime, moving on to the nex one only the previous one is correct. If a digit is incorrect, the algorithm stop. Therefore, the more correct digits there are, the longer the verification process will take. In this case, an attacker tries the ten possibilities for the first digits and selects the one with the highest running times. And so on until the attacker finds the four digits of the code pin.*

In the example, the early stop of the algorithm allows an attack but can also be more specific, such as the method used to make logical operations.

3.2.2 Cache Attack

In cache attacks, an attacker monitors the cache access made by the victim. The attacker and the victim need to share a physical system. The idea is to determine with the knowledge of the victim's different cache access (or no access) if a specific value (or other) is in the sensitive element. Let us give an example with a flash and reload attack.

Example 10. *The attacker selects one cache and evicts them. Then, the attacker waits for the victim to run the algorithm and reload the data. According to the running times, the attacker knows if the victim accessed or not the cache and deduced some information.*

The cache attacks can vary depending on the type of architecture, as for the two next side-channels attacks, the power analysis and electromagnetic emanation attacks.

3.2.3 Power Analysis Attack

Contrary to cache attacks, the power analysis attack requires direct access to the device, microprocessor, or hardware to monitor the power consumption through an oscilloscope. It makes this attack more difficult to step than the previous one. The idea behind the attack is that any operation require a certain amount of power to be executed, which depends on the parameters of the operation. We distinguish two types of power analysis: the Simple Power Analysis (SPA) and the Differential Power Analysis (DPA).

Simple Power Analysis

SPA is the simplest of the two types. It involves observation of the power measurement (or trace) within time windows. For the observation, the attacker deduces information. For example, (we suppose that the device is not protected):

Example 11. *Let us have an algorithm executing a multiplication. If the value a is even, then the multiplication is $a \times a$. Otherwise, it is $2 * a$. Except for a equals two, it is highly likely that the attacker can distinguish the square multiplication from the other multiplication and then determine if a is even or odd.*

Differential Power Analysis

DPA uses statistics to analyze the power consumption of a cryptographic algorithm. It exploits the variations observed in the power measurement. Even though it requires numerous traces, the DPA attacks allow the use of traces unexploitable with SPA. An example of an attack using the DPA on RSA is presented by Kocher, Jaffe, and Jun [46].

Often, numerous measurements are necessary to attack using power measurement. However, it is sometimes possible to only use a unique trace of power measurement to make a successful attack. It is called a Single Trace Attack (STA).

3.2.4 Electromagnetic Emanation Attack

The electromagnetic emanation (EM) attack uses the emanations emitted by a device during the execution of a cryptographic algorithm. The measurement obtained is analysis, similar to the power analysis attacks analysis. Moreover, we distinguish two main types of analysis: the Differential Electromagnetic Analysis (DEMA) and the Simple Electromagnetic Analysis (SEMA). Those attacks are the same idea as the SPA and the DPA.

Remark 6. *SEMA is more effective for asymmetric cryptography and sometimes requires few traces. DEMA is also used for symmetric cryptography.*

To compare EM attacks can be more difficult as the electromagnetic emanations are obtained with a probe, and the right place can be hard to determine. Sometimes, it requires depacking the chip and collecting the signal closer to the source.

The four side-channel attack types presented previously are the four main ones, but other ones also exist using acoustics or even temperature.

3.3 Countermeasurses

At the same time, as advances have been made in side-channel attacks, protection methods have been developed to reduce the possibilities of attack.

Definition 38. *(Countermeasure) Actions, devices, procedures, techniques, or other measures that reduce the vulnerability of an information system.*

The countermeasures take different forms according to the necessity or the possibilities of actions. For example, on a microprocessor, they can include a constant-time multiplication instruction during the microprocessor design. However, a user cannot change the type of multiplications but can protect through the implementation.

In this section, we introduce three countermeasures applicable at the implementation stage: the constant time (cite a few sentences ago), the shuffling, and the masking.

3.3.1 Constant-time

An implementation is said to be in constant time if it resists timing and cache attacks. In other words, the execution time of an implementation cannot be exploited to recover information about sensitive elements. The name constant-time comes from the fact that a function in constant-time must take the same amount of time to be executed independently to the parameters (of the same size). For instance, to check a code pin of four digits, the code must always check the four digits, even if the first one is false.

To be in constant time, an implementation must, at least, follow the following rules:

- Sensitives values may not be used to decide the next code steps, i.e. not used as a condition in the branch instruction.
- Sensitives values may not be used to decide what memory address to access.
- Sensitives values may not be used as input to variable-time functions or instructions.

The rules described above are the minimum requirements to ensure constant time in an implementation. The timing and cache attacks are seen as a major threat to security due to their ease of setup. Therefore, the constant-time implementations are increasingly being deployed. Nonetheless, the constant-time implementations are vulnerable to power analysis or EM attacks. Thus, both attacks require different countermeasures.

Remark 7. *The constant-time protections can be exploited to attack with the other side-channel attacks.*

3.3.2 Shuffling

The first countermeasure against EM and power analysis is the shuffling. It consists of randomizing the order of the operations or the treatment order of the sensitive elements during the execution. For instance, in a function executing a scalar multiplication of a vector $(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{10})$ by λ, instead of computing $\lambda \otimes \mathbf{x}_0$ first then $\lambda \otimes \mathbf{x}_1$ and so on. The function, once, compute $\lambda \otimes \mathbf{x}_7$ then $\lambda \otimes \mathbf{x}_2$ and so on. And the next time, it will start with another $\mathbf{x}_i$.

The shuffling countermeasure does not necessarily eliminate the leakage. It increases the difficulty enough to make it unusable. The order of execution must be determined randomly; otherwise, there is potential for exploitation to attack.

3.3.3 Masking

The masking is more complex than the shuffling. The idea is to hide the sensitive elements with masks. In other words, a sensitive element is modified with another value of the same type randomly generated before any operation. Once the operations are over, the mask is removed. So, the leakage is eliminated as the access to the sensitive element is removed. We distinguish two types of masking: the boolean and the arithmetic masking.

The boolean masking used boolean operations such as the XOR to modify the sensitive element with a random binary element. The arithmetic masking is the same as boolean masking but with arithmetic operations such as addition or multiplication. From an implementation point of view, arithmetic masking is more challenging to implement than boolean masking.

Example 12. *Let us make an addition $z = x + y$, with x a sensitive element and y known. We generate $mask_1$, a random element. We compute $x_1 = x + mask_1$, then $z = (x_1 + y) \star mask_1$. In boolean masking, the operators $+$ and $\star$ are an XOR, while in arithmetic masking, $+$ is an addition and $\star$ a subtraction.*

However, the shuffling and the masking have a main drawback: the computation cost. Both require randomly generated elements, and the masking requires more operations. For instance, in the previous example, instead of one operation, it required three operations.

Generally, the countermeasures for the power analysis and EM attacks are less common in implementation because of their cost and as these attacks are less immediate threats.

Chapter 4

BIKE and ROLLO two Candidates of the NIST Standardization

Code-based cryptography has been studied since the end of the seventies. Firstly, with McEliece encryption and a few years after the Niederreiter scheme. In the following years, attacks and variants of those schemes were proposed, making the schemes more secure. However, code-based cryptography has a main drawback: the huge sizes of keys, especially in comparison to the ones for RSA, an encryption scheme introduced in the same years. The general interest was focused on the latter.

A revival of interest in code-based cryptography research occured with the demonstration of the existence of an efficient algorithm able to find the prime factors of an integer: Shor's algorithm. Although dependent on the existence of quantum computers, this algorithm breaks the security of RSA but not the problem on which the security of code-based cryptography relies. It was even more prominent with the Post-Quantum cryptography standardization process organized by NIST.

With the opening of the standardization process, news code-based schemes were proposed as candidates, including PKEs, KEMs, and signatures. Between the 21 propositions, Classic McEliece represents the initial schemes. However, there were also adaptions of schemes with new families of codes, especially with the quasi-cyclic structure, to reduce the size of the keys.

NIST made a selection of candidates at the end of the first round. Only ten candidates were selected and reduced to seven due to their similarity (family of codes). The seven candidates contain only KEMs or PKEs. At the beginning of the second round, January 2020, five candidates were based on the Hamming metric.

- Classic McEliece [13],
- NTS-KEM (merge with Classic McEliece later) [3],
- LEDACrypt [8],
- HQC [54],
- BIKE [1],

As well as two in rank metric:

- ROLLO [2],
- RQC [53].

In this work, we focus on two candidates, BIKE and ROLLO. They are both very similar if we disregard the specific features of the metric, but these features also make them interesting to study.

4.1 BIKE: Bit-Flipping Key Encapsulation

Initially, BIKE is a candidate for the standardization process containing three key encapsulation mechanisms. They are an adaption of McEliece, Niederreiter, and Aleknovich cryptosystems to KEM with security based on Quasi-Cyclic Moderate-Density Parity-Check codes. Respectively refer as BIKE-I, BIKE-II, and BIKE-III.

Progressively to the standardization process, the number of schemes for BIKE was reduced to solely one, BIKE-II, now referred to as BIKE.

An essential point about BIKE is that the scheme was modified numerous times to go from an IND-CPA KEM only used with ephemeral keys as allowed by the NIST to an IND-CCA KEM under some assumption. In this section, we only talk about the latter.

4.1.1 BIKE Scheme

As with any key encapsulation mechanism, the three algorithms (KeyGen, Encap, Decap) compose BIKE.

Key Generation

In BIKE, the private key is a (2,1)-QC-MDPC code of length n, and its related public key, the parity-check matrix H in its systematic form. The advantage of utilizing the (2,1)-QC codes is that only the first row needs to be randomly generated as the parity-check matrix is a two-block circulants matrix. Thus, KeyGen creates the private key p_k needed to generate two vectors $\mathbf{h}_0$ and $\mathbf{h}_1$ of length r such that $n = 2r$. From p_k, the public key pu_k is computed by the polynomial multiplication $h = h_1 * h_0^{-1}$. It is possible because there exists a ring isomorphism between binary circulant matrices and the quotient polynomial ring $\mathcal{R} = \mathbb{F}_2[X] / X^r - 1$.

For the more formal description, see Figure 4.1, let us define $\mathscr{H}_\omega = \{(\mathbf{h}_0, \mathbf{h}_1) \in \mathcal{R} | w_H(\mathbf{h}_0) = w_H(\mathbf{h}_1) = \frac{\omega}{2}\}$ is the private key space. It includes the weight constraint on the private key. $\mathcal{M} = \{0,1\}^\ell$ is the message space. For all the security level λ, either 128, 192, or 256, $\ell = 256$ in BIKE parameters.

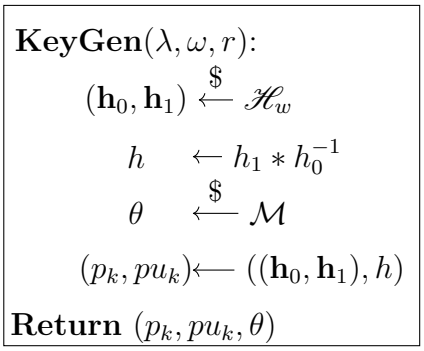


Figure 4.1: Key generation in BIKE

Note that the key generation algorithm of BIKE also returns θ, an element of the message space. It is not in the public key nor on the private one. However, Alice keeps this element hidden in case of a decryption failure.

Encapsulation

Executed by Bob with p_k, the encapsulation algorithm creates the shared secret $\mathcal{K}$ from a message m randomly generated at the beginning of the algorithm. m is used as a seed to generated the errors vectors $\mathbf{e}_0$ and $\mathbf{e}_1$ with the hash function $\mathbf{H}$. $\mathbf{e}_0$ and $\mathbf{e}_1$ belongs to the error vector space $\mathscr{E}_\rho = \{(\mathbf{e}_0, \mathbf{e}_1) \in R^2 | w_H(\mathbf{e}_0) + w_H(\mathbf{e}_1) = \rho\}$ with ρ a parameter given in Table 4.1 and they are used to encrypt m. Here as well a hash function is used, $\mathbf{L}$ takes $(\mathbf{e}_0, \mathbf{e}_1)$ and return a element of $\mathcal{M}$ which is XORed to m to obtain $\mathbf{c}_1$. The error vectors are also encoded with the public key into $\mathbf{c}_0$ to deliver m to Alice. $(\mathbf{c}_0, \mathbf{c}_1)$ is required as the parameter of the hash function $\mathbf{K}$ to the last step of the Encapsulation in the computation of the shared secret. $\mathbf{K}$ is a key derivation function.

Remark 8. *The hash functions* $\mathbf{H}$, $\mathbf{L}$, *and* $\mathbf{K}$ *are random oracles. In concrete implementation [1],* $\mathbf{H}$ *uses SHAKE256 while* $\mathbf{L}$ *and* $\mathbf{K}$ *use SHA3-384.*

Figure 4.2 formally describes the encapsulation algorithm.

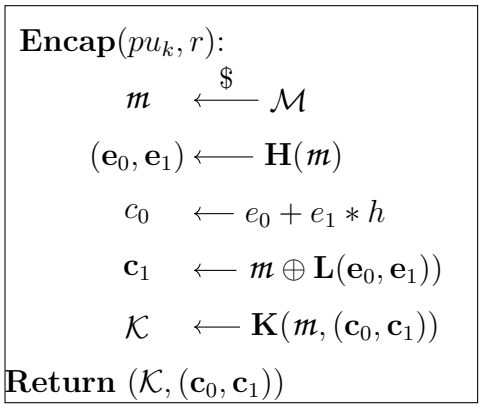


Figure 4.2: Encapsulation algorithm in BIKE

Decapsulation

The BIKE decapsulation algorithm enables Alice to compute the shared secret from the cipher with its private key. To do so, Alice must extract the message m from the cipher, which is processed in three steps. First, recover the error vectors with a QC-MDPC decoder with $\mathbf{c}_0$ and p_k. Second, computed m' the same way as for m. Third, check if the error vectors are equivalent to the result of $\mathbf{H}(m')$. If so, then m' is used to compute the share secret $\mathcal{K}$. Otherwise, it is replaced by θ. In any case, $\mathcal{K}$ is returned with the impossibility of distinguishing whether it is one case or the other.

The Decapsulation is formally described in Figure 4.3.

Decap$(p_k, \theta, (\mathbf{c}_0, \mathbf{c}_1))$:

$e' \leftarrow$ DECODER$(c_0 * h_0, (\mathbf{c}_0, \mathbf{c}_1))$

$m' \longleftarrow c_1 \oplus \mathbf{L}(e')$

IF $e' = \mathbf{H}(m')$ THEN

$\mathcal{K} \leftarrow \mathbf{K}(m', (\mathbf{c}_0, \mathbf{c}_1))$

ELSE

$\mathcal{K} \longleftarrow \mathbf{K}(\theta, (\mathbf{c}_0, \mathbf{c}_1))$

Return $(\mathcal{K})$

Figure 4.3: Decapsulation in BIKE schemes

Full BIKE

Figure 4.4 recaps the BIKE scheme process and Table 4.1 the parameters proposed.

Security	λ	ℓ	r	ω	ρ
Level 1	128	256	12323	142	134
Level 3	192	256	24659	206	199
Level 5	256	256	40973	274	264

Table 4.1: BIKE parameters

Decoder Selection

In BIKE specification, the authors specified that any QC-MDPC codes decoder can be used. However, the choice of DECODER is crucial for the security claim. Indeed, the IND-CCA security depends on the correctness of DECODER. Otherwise, BIKE only has IND-CPA security and must be used with an ephemeral key. In other words, new keys are generated by KeyGen each time BIKE is used.

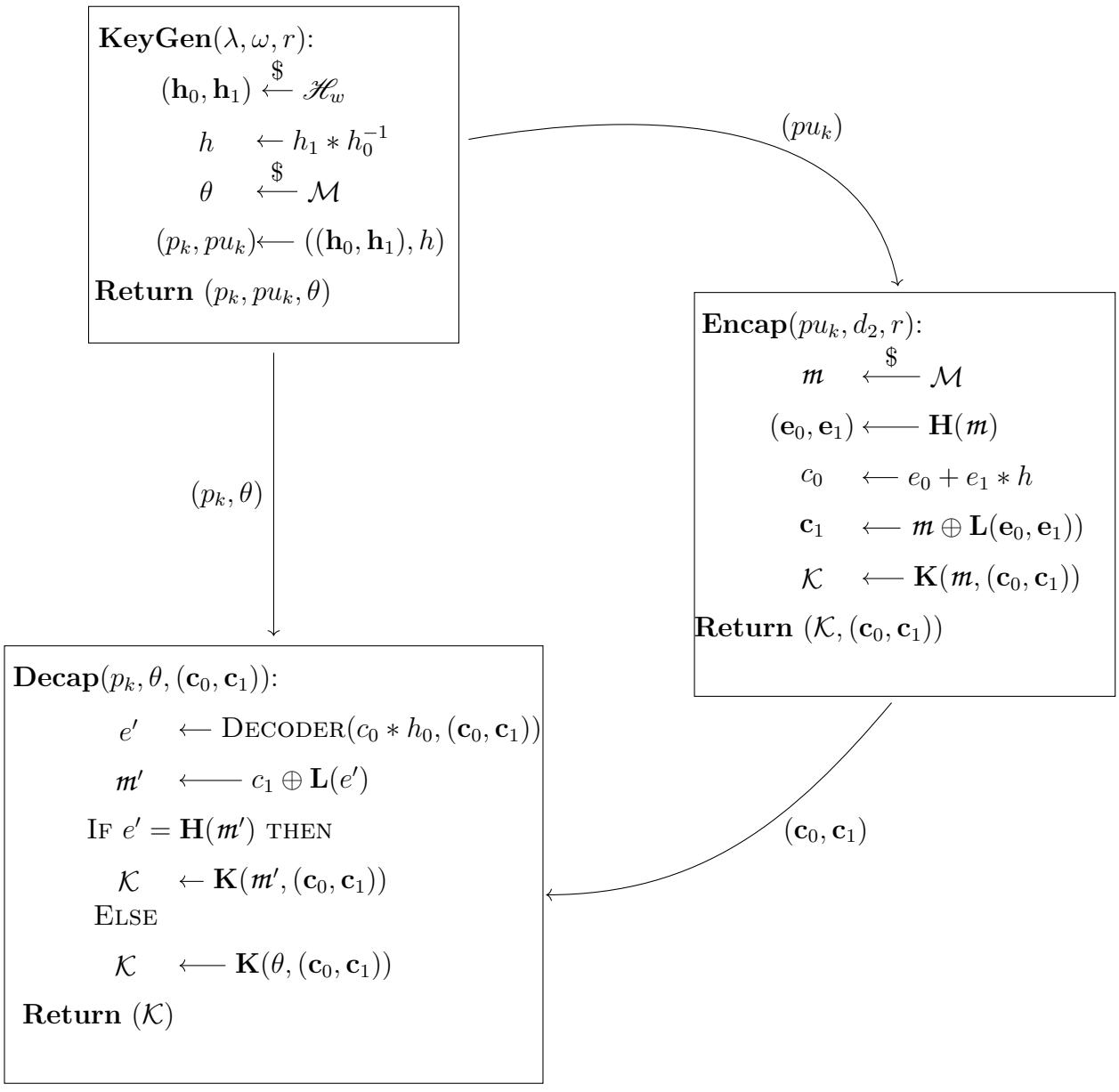


Figure 4.4: BIKE scheme

In the specification, one decoder is proposed: the Black-Grey Flip.

4.1.2 BIKE's Decoder

The Black-Grey Flip decoder is a variant of the iterative bit-flipping algorithm proposed by Robert Gallager [40] for Low-Density Parity-Check codes.

Bit-Flipping Decoder

The Bit-Flipping decoder takes the syndrome $\mathbf{s}$ and the parity-check matrix H and works as follows: first, the error vector $\mathbf{e}$ of length n is created and set at 0. Then, the following step is repeated for each column of H. s is compared to the column $H_{,j}$ of H. The algorithm counts each time that $\mathbf{s}_i$ equals $H_{i,j}$ and equals 1. If the number obtained is equal or greater than a threshold T, then $\mathbf{e}_j$ is XORed to 1. Once all the iterations on the columns are executed, a new syndrome is computed as follows: $\mathbf{s} = \mathbf{s} \oplus \mathbf{e} \cdot H^t$. The process is repeated nb times. In the end, the syndrome is only filled with 0, and the vector error $\mathbf{e}$ is returned.

COUNTER function The count of the ones at the same position in the syndrome and in one column j of H is executed by a function called COUNTER, visually represented in Figure 4.5. COUNTER return an integer call *count*.

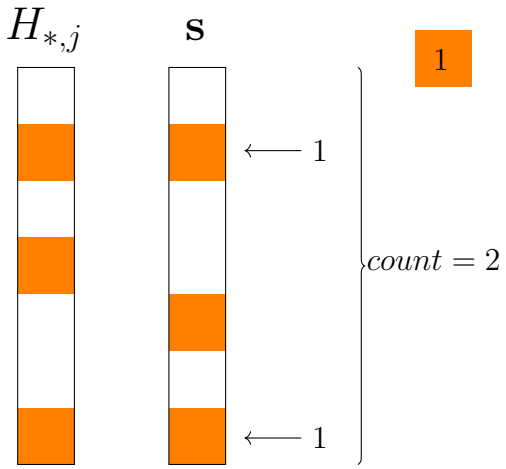


Figure 4.5: Visual representation of the COUNTER function for one column of H.

Threshold The threshold T is computed with a formula according to the Hamming weight of the syndrome and the iteration. In Algorithm 2, the computation is represented by THRESHOLD.

Bit-Flipping algorithm The whole Bit-Flipping algorithm is described in Algorithm 2.

Algorithm 2: Bit-Flipping

Input: $\mathbf{s}$,H
Output: $\mathbf{e}$

$\mathbf{e} \leftarrow 0^n$
for $i \leftarrow 1$ **to** N **do**
  $T \leftarrow \text{THRESHOLD}(w_H(\mathbf{s}), i)$
  for $j \leftarrow 1$ **to** n **do**
    $count \leftarrow \text{COUNTER}(H_{*,j}, \mathbf{s})$
    if $count \geq T$ **then**
      $\mathbf{e}_j \leftarrow \mathbf{e}_j \oplus 1$
  $\mathbf{s} \leftarrow \mathbf{s} \oplus \mathbf{e} \cdot H^T$

Black-Grey Flipping Decoder

One problem with the Bit-Flipping decoder is that some bits of the error vector can be flipped because of a high counter, but if they are wrongly flipped, it is hard to correct them. The different variants of the Bit-Flipping algorithm that lead to the Black-Grey Flipping (BGF) tend to correct that.

The algorithm uses two additional vectors, **black** and **grey**, set with zeros at the beginning. Each time the error vector is modified, the corresponding bit in **black** is flipped to 1. **grey** is modified only if *count* is less than T but greater or equal to $T - \tau$, where τ is fixed. Then, once the initial Bit-Flipping is over and the new syndrome computed, two Bit-Flipping iterations are executed, but this time, the threshold is computed differently, and $\mathbf{e}_j$ is not Xored to 1 but to the value into $\mathbf{black}_j$ for the first one and $\mathbf{grey}_j$ for the second one. These two additional Bit-Flipping iterations with **black** and **grey** are only executed once. It is during the first main iteration, *i. e.* $i = 1$.

In short, the two additional steps will confirm, remove, or add modifications in $\mathbf{e}$.

Algorithm 3 describes the Black-Grey Flipping algorithm.

BGF Algorithm BIKE Parameters

Some variables necessary for the Black-Grey Flipping algorithm in BIKE execution are fixed, such as the number of iterations. Those values and the formula to compute the threshold T are given in Table 4.2.

The last element in Table 4.2 is DFR, an acronym for Decoding Failure Rate. In this case, it is an estimated value for BFG according to the parameters defined just before. The smaller, the better, as the IND-CCA security of BIKE depends on the DFR. Nevertheless, even if, for this algorithm, the DFR tends to be small in simulation, there is no formal proof of an upper bound.

The question of the value of the decoding failure rate is essential as there exists an attack based on it.

Algorithm 3: Black-Grey Flipping

Input: $\mathbf{s}$,H,ω
Output: $\mathbf{e}$

$\mathbf{e} \leftarrow 0^n$
for $i \leftarrow 1$ **to** N **do**
  $T \leftarrow \text{THRESHOLD}(w_H(\mathbf{s}), i)$
  for $j \leftarrow 1$ **to** n **do**
    $count \leftarrow \text{COUNTER}(H_{*,j}, \mathbf{s})$
    if $count \geq T$ **then**
      $\mathbf{e}_j \leftarrow \mathbf{e}_j \oplus 1$
      $\mathbf{black}_j \leftarrow 1$
    else if $count \geq (T - \tau)$ **then**
      $\mathbf{grey}_j \leftarrow 1$
  $\mathbf{s} \leftarrow \mathbf{s} \oplus \mathbf{e} \cdot H^T$
  if $i == 1$ **then**
    $T_1 \leftarrow \frac{\omega}{2}$
    for $j \leftarrow 1$ **to** n **do**
      $count \leftarrow \text{COUNTER}(H_{*,j}, s)$
      if $count \geq T_1$ **then**
        $\mathbf{e}_j \leftarrow \mathbf{e}_j \oplus \mathbf{black}_j$
      if $count \geq T_1$ **then**
        $\mathbf{e}_j \leftarrow \mathbf{e}_j \oplus \mathbf{grey}_j$
    $\mathbf{s} \leftarrow \mathbf{s} \oplus \mathbf{e} \cdot H^T$

Security	N	τ	T	DFR
Level 1	5	3	$max(\lfloor 0.0069722 \cdot w_H(s) + 13.530 \rfloor, 36)$	2^{-128}
Level 3	5	3	$max(\lfloor 0.005265 \cdot w_H(s) + 15.2588 \rfloor, 52)$	2^{-192}
Level 5	5	3	$max(\lfloor 0.00402312 \cdot w_H(s) + 17.8785 \rfloor, 69)$	2^{-256}

Table 4.2: BGF parameters for BIKE

4.1.3 Attacks and Implementations of BIKE

In the state-of-the-art BIKE, we can observe a variety of works, from the study of the mathematical resistance against attack to the side-channel resistance of the different implementations of BIKE.

General Study of the Scheme

Many attacks for Hamming metric code-based cryptography were known during the design of BIKE. So, the parameters of BIKE were chosen such that BIKE is resistant to mathematical attacks. The scheme is beginning to be well studied from this point of view. However, there is still research to try some vulnerabilities. They sometimes use a Quantum ISD [78] or exploit the DFR [63, 6]. Using the side channels to attack with a mathematical

attack. The idea is to recover the private key from partial exposure through side-channel [35]. For instance, it is possible to recover the private key by combining a timing attack exploiting the rejection sampling and the GRJ attack [42].

For side-channel attacks, two other papers on implementations for QC-MDPC cryptosystems are interesting. Indeed, both are using power consumption analysis to recover the private. One executes a DPA to recover h_0 and solves a system of equations to get h_1 [71]. The second proposed another DPA as well as a single trace attack to execute the same final processes to get the private key [74]. Both are using a leakage in the implementation.

Implementation

There are numerous implementations of BIKE schemes with different characteristics. There are the official ones described in the specification of BIKE used as reference ones [1]. An important work was done by Drucker, Gueron, and Kostic on the implementation of QC-MDPC codes-based cryptography [34]. They are the authors of the BIKE portable implementation, available on the PQM4 git (git gathering NIST candidates still in list implementation for Cortex-M4). Other works followed to optimize or guarantee the constant-time security [20, 21] as well as for other devices, for instance, FPGA [66, 67, 39].

Those new implementations were not studied for side-channel attacks, making them interesting study subjects.

Remark 9. *The BIKE implementation in the PQM4 git was modified by Chen et al. with their optimizations and the constant-time improvement.*

4.2 ROLLO: Rank-Ouroboros, LAKE, and LOCKER

Merge of three candidates in the first round of the NIST standardization process; ROLLO was one of the candidates in rank metric in the second round. Even so, eliminated in July 2020, ROLLO was an interesting candidate due to its small keys. However, the need for more security information eliminated the rank metric code-based cryptography from the standardization.

A few months before the end of the second round, ROLLO was reduced to two schemes: one KEM, ROLLO-I, and one PKE ROLLO-II, with new parameters to resist algebraic attacks. As ROLLO-I and ROLLO-II are adapted from the Niederreiter scheme, a few modifications are necessary to obtain the PKE from the KEM.

4.2.1 $\mathbb{F}_{2^m}$ specificities

Reminder: the Ideal-LRPC codes used in ROLLO are formed of elements of $\mathbb{F}_{2^m}$, an extension of $\mathbb{F}_2$, see Subsection 2.3.3. To construct the extension field $\mathbb{F}_{2^m}$, a polynomial

P_m of degree m is used.

The m values and the corresponding polynomials are given in the specification for ROLLO [2]. Table 4.3 gives the polynomial according to the different possibilities of m given in the parameters of ROLLO.

m	P_m
67	$X^{67} + X^5 + X^2 + X + 1$
79	$X^{79} + X^9 + 1$
83	$X^{83} + X^7 + X^4 + X^2 + 1$
97	$X^{97} + X^6 + 1$

Table 4.3: Polynomial P_m according to the m values in ROLLO specification.

The fact is that by working with $\mathbb{F}_{2^m}$, it also means that the vectors in ROLLO are composed of elements of $\mathbb{F}_{2^m}$. The set of the vectors of length n and dimension of support ω is defined as follows:

$$\mathscr{S}_w^r(\mathbb{F}_{2^m}) = \{\mathbf{x} \in \mathbb{F}_{2^m}^r | dim(Supp_R(\mathbf{x})) = \omega\}$$

r and ω are two parameters depending on the scheme and the security level. They are given in Subsection 4.2.2.

$\mathscr{S}_\omega^r$ is the unique set of vectors of $\mathbb{F}_{2^m}$ elements of length r, and it is defined for the private key vector space as well the error vector space of ROLLO.

4.2.2 ROLLO schemes

Key Generation of ROLLO

The key generation algorithm is identical for ROLLO-I and ROLLO-II. The schemes rely on a (2,1)-ideal LRPC codes with the parity-check matrix generated by the vectors $\mathbf{h}_0$ and $\mathbf{h}_1$. $(\mathbf{h}_0, \mathbf{h}_1)$ forms the private key and their support F as a dimension of ρ. The public key h is computed by $h = h_1 * h_0^{-1} \mod P$. P is a polynomial of degree r defined in the parameters of ROLLO, it ensures that h is at most of degree $r - 1$. All the vectors generated by the key generation algorithm are of length r.

Figure 4.6 gives the formal description of KeyGen. The algorithm takes as input ρ and r and returns pu_k the public key, p_k the private key, and F the support of the private key, which is kept secret.

ROLLO-I

As a Key Encapsulation mechanism, ROLLO-I aims to create a shared secret $\mathcal{K}$ and to pass it down to Alice without revealing it to others. In this case, the result of the hash function $\mathbf{H}$ on the support of the error vectors.

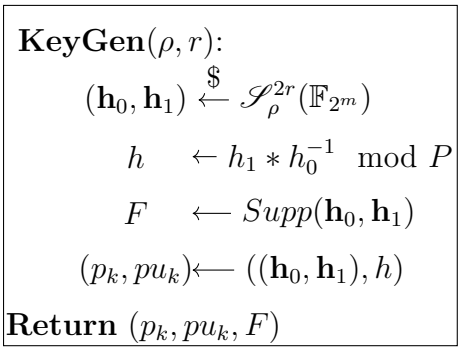


Figure 4.6: Key Generation for ROLLO schemes

Encapsulation In the encapsulation algorithm, the error vectors randomly generated into the space of error vectors ($\mathscr{S}_{\omega}^{n}(\mathbb{F}_{2^m})$) are essential to the computations that follow their generation. Indeed, as said previously, $(\mathbf{e}_0, \mathbf{e}_1)$ is the entry of the hash function $\mathbf{H}$ to compute $\mathcal{K}$. Nonetheless, $(\mathbf{e}_0, \mathbf{e}_1)$ is encoded into c to be send to Alice in secure way.

The encapsulation algorithm is formally described in Figure 4.7.

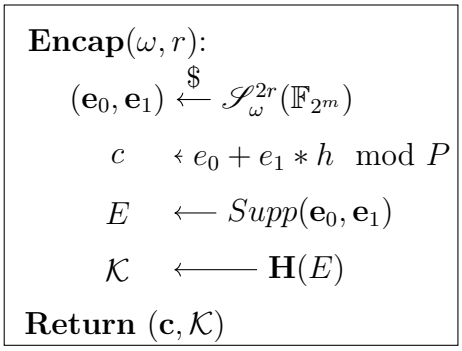


Figure 4.7: Encapsulation in ROLLO-I

Decapsulation With the decapsulation algorithm of ROLLO-I, Alice recovers the support of the error vectors E' using the Rank Support Recover (RSR) algorithm, described in Subsection 4.2.3. The RSR algorithm takes as input F, the dimension supposed of E', and the syndrome vector $s = h_0 * c \mod P$. There is no verification of the support obtained, and the shared secret is computed as the same as for the encapsulation.

Figure 4.8 formally described ROLLO-I decapsulation.

The parameters according to the security level for ROLLO-I are given in Table 4.4, in Subsection 4.2.3 .

ROLLO-II

ROLLO-II is public-key encryption, so a message $\mathfrak{m}$ is encrypted in *cipher* during the encryption process and decrypted in the decryption. *cipher* is simply the XOR between

$$
\begin{array}{l}
\textbf{Decap}(\mathbf{c}, F, (\mathbf{h}_0, \mathbf{h}_1), \omega): \\
\quad \mathbf{s} \longleftarrow h_0 * c \mod P \\
\quad E \longleftarrow RSR(F, \mathbf{s}, r) \\
\quad \mathcal{K} \longleftarrow \mathbf{H}(E) \\
\textbf{Return } (\mathcal{K})
\end{array}
$$

Figure 4.8: Decapsulation in ROLLO-I

the message and the result of $\mathbf{H}$ with $(\mathbf{e}_0, \mathbf{e}_1)$. Therefore, only a few additions are necessary to obtain the ROLLO-II scheme from ROLLO-I.

Encryption ROLLO-II encryption algorithm takes m as additional parameters. Until the end, the algorithm is identical to the ROLLO-I decapsulation algorithm. At the end, the algorithm computes $\mathbf{cipher} = m \oplus \mathbf{H}(E)$ and returns $(\mathbf{c}, \mathbf{cipher})$.

Figure 4.9 formally describes the encryption algorithm of ROLLO-II.

$$
\begin{array}{l}
\textbf{Encrypt}(\omega, r, m): \\
\quad (\mathbf{e}_0, \mathbf{e}_1) \xleftarrow{\$} \mathscr{S}_{\omega}^{2r}(\mathbb{F}_{2^m}) \\
\quad \mathbf{c} \leftarrow e_0 + e_1 * h \mod P \\
\quad E \longleftarrow Supp(\mathbf{e}_0, \mathbf{e}_1) \\
\quad \mathbf{cipher} \longleftarrow m \oplus \mathbf{H}(E) \\
\textbf{Return } (\mathbf{c}, \mathbf{cipher})
\end{array}
$$

Figure 4.9: Encryption in ROLLO-II

Decryption Similarly to the encryption, the decryption is the same as the decapsulation with an additional operation to recover the message m'. This algorithm returns m'. ROLLO-II decryption is formally described in Figure 4.10.

$$
\begin{array}{l}
\textbf{Decryp}((\mathbf{cipher}, \mathbf{c}), F, (\mathbf{h}_0, \mathbf{h}_1), \omega): \\
\quad \mathbf{s} \longleftarrow h_0 * c \mod P \\
\quad E \longleftarrow RSR(F, \mathbf{s}, \omega) \\
\quad \mathbf{m}' \leftarrow cipher \oplus \mathbf{H}(E) \\
\textbf{Return } (m')
\end{array}
$$

Figure 4.10: Decryption in ROLLO-II.

Summary ROLLO schemes and parameters

ROLLO-I and ROLLO-II full schemes are described in Figure 4.11.

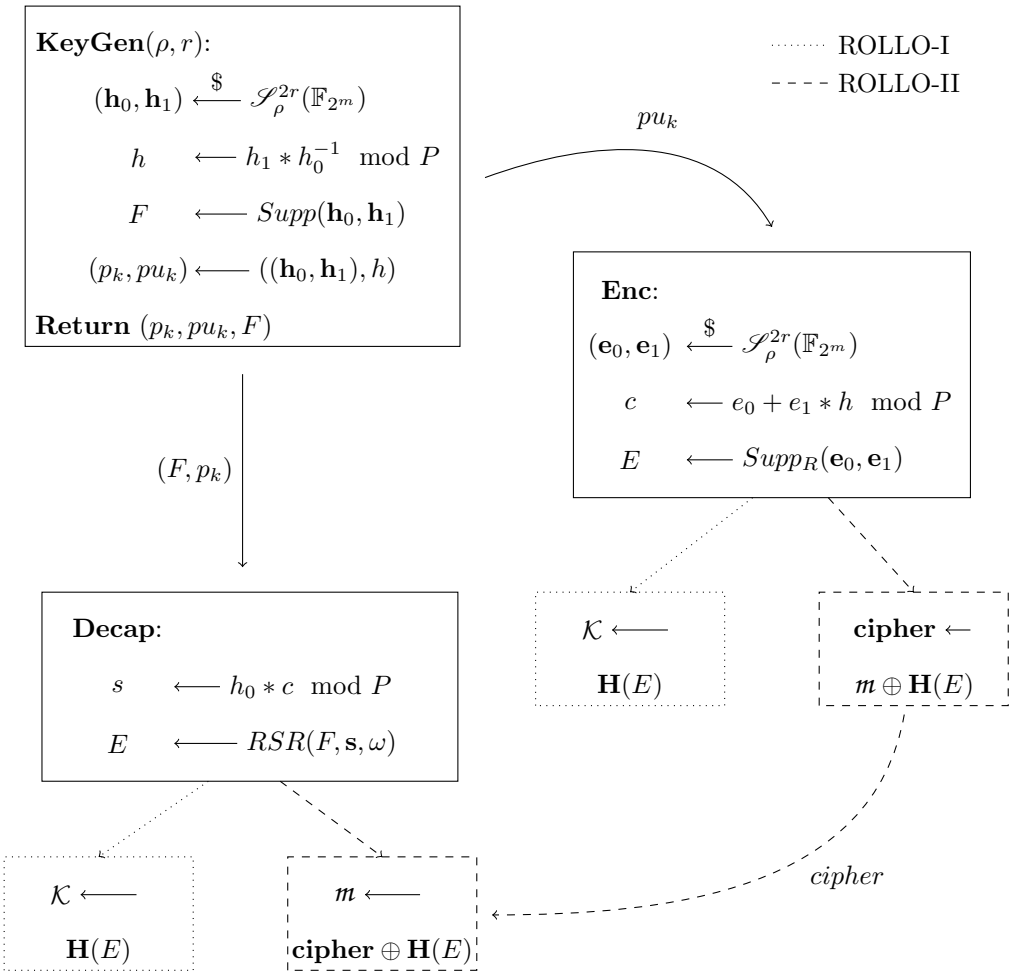


Figure 4.11: ROLLO-I and ROLLO-II full schemes.

Table 4.4 gives the parameters of ROLLO-I and ROLLO-II according to the targeted security level. The parameters for ROLLO-I are smaller than the ROLLO-II ones, especially for r, because the security requirement is different. Indeed, as a KEM, ROLLO-I only claims an IND-CPA security, while ROLLO-II needs an IND-CCA security. Thus, the decoding failure rate of the Rank Support Recover algorithm must be smaller in the latter than in the former. DFR depends on the parameters; see Subsection 4.2.3.

4.2.3 Rank Support Recovery Algorithm

The Rank Support Recover (RSR) algorithm is a decoder algorithm for the LRPC codes [36]. It consists of two main steps. First, recover the support of the error vectors E. Second, it recovers the errors coordinates by solving a linear system of equations. In ROLLO decap-

Name	Security	λ	r	m	ρ	ω	P
ROLLO-I-128	Level-1	128	83	67	8	7	$X^{83} + X^7 + X^4 + X^2 + 1$
ROLLO-I-192	Level-2	192	97	79	8	8	$X^{97} + X^6 + 1$
ROLLO-I-256	Level-3	256	113	97	9	9	$X^{113} + X^9 + 1$
ROLLO-II-128	Level-1	128	189	83	8	7	$X^{189} + X^6 + X^5 + X^2 + 1$
ROLLO-II-192	Level-2	192	193	97	8	8	$X^{192} + X^{15} + 1$
ROLLO-II-256	Level-3	256	211	97	9	8	$X^{211} + X^{11} + X^{10} + X^8 + 1$

Table 4.4: ROLLO-I and ROLLO-II parameters.

sulation and decryption, the second step is unnecessary as only the support of the error is used to generate the shared secret.

The process to recover E with RSR is in two steps. The support S of the syndrome vector is recovered. It can be made by a Gaussian elimination on the binary matrix representation of the syndrome. Once the S is obtained, the error vectors can be recovered by the intersection of S and the inverse of elements in the support F of the private key.

Algorithm 4 formally describes the RSR algorithm.

Algorithm 4: Rank Support Recover

Input: F,s,ω
Output: E
1 $S \leftarrow Supp(s)$
2 $E \leftarrow \bigcap_{i=1}^{\omega} f_i^{-1} S$

Algorithm 4 has a possibility of failure. The Decoding Failure Rate for the RSR algorithm in the case of ROLLO is computed as follows :

$$2^{-(\rho-1)(m-\rho\omega-\omega)} + 2^{-(r-\rho\omega+1)}$$

Table 4.5 gives DFR according to the ROLLO schemes and the security level.

Name	DFR
ROLLO-I-128	2^{-35}
ROLLO-I-192	2^{-35}
ROLLO-I-256	2^{-35}
ROLLO-II-128	2^{-35}
ROLLO-II-192	2^{-35}
ROLLO-II-256	2^{-35}

Table 4.5: ROLLO-I and ROLLO-II Decoding Failure Rate.

4.2.4 Previous works on ROLLO schemes

As a more recent field, there is less research on the security of the rank metric code-based cryptography than the Hamming metric ones, especially for the LRPC codes.

Algebraic attacks

Nonetheless, during the second round of the PQC standardization process, new algebraic attacks against the RSD problem were proposed [9, 10]. Hopefully, this attack did not break ROLLO. However, the parameters have had to be revised. The parameters present in Subsection 4.2.2 are the last ones proposed and chosen to resist the attacks.

Rank-based cryptography library

The rank-based cryptography library [4] is a library for implementing rank-based cryptography in C. It is not only for ROLLO schemes but also for RQC and rank-based signatures. This library contains many functions essential in rank metric implementation. The library is also proposing functions in constant time. The library has not been attacked through side-channel in any previous works.

Other implementations

There are other implementations. There are two other implementations for micro-processor [48, 56]. The first one proposes an optimized implementation of ROLLO. The second one only proposes an implementation for ROLLO-I. However, this work also contains a part on the security of the implementation, specifically, the authors propose a side-channel attack on the implementation. This attack exploits power measurement in order to recover the private key. Furthermore, they also propose a countermeasure.

The most recent implementation is a constant-time one for AVX2 [27]. However, it is for a variant of ROLLO.

BIKE and ROLLO are the two schemes we have selected to study the security of their implementations using a methodology explained in the next chapter.

Chapter 5

Methodology

In this thesis, we sought to test the resistance of cryptosystem implementations based on error-correcting codes when attacked with side-channels.

This chapter introduces the methodology we have followed to uncover the vulnerabilities of an implementation by following three distinct steps. The first one is to select a scheme and its implementation. The second is its analysis of vulnerabilities and finding a way to exploit them with a side-channel attack. The final step is experimentation, enabling us to check the feasibility of our attack using the tools at our disposal, but also the limitations of the attack.

5.1 Selection of the Scheme

There are a variety of cryptographic algorithms based on codes. In order to simplify the selection, we have chosen to consider only the candidates for NIST post-quantum standardization. In particular, those that have reached the second round. It eliminates cryptosystems that have shown construction weaknesses that cannot be solved by changing parameters.

Subsequently, the choice of scheme studied is based on the selection of its implementation. We focus on analyzing implementations proposed by people specialized in it and in open access. Two reasons for this choice. First, we wanted a real implementation study case. Second, we ensure the reproducibility of our attack.

This criterion does not lead to eliminating any schemes on the list because NIST required two implementations, one reference, and one optimized to submit to the standardization process. Furthermore, other implementations with better performance or special features, such as constant time, were proposed.

What influenced our implementation choice was its adaptability to the system used for our experiments. In all our experiments, we use microcontrollers.

Microcontroller

A microcontroller is a small processor unit on a single integrated circuit designed for embedded systems. It is able to execute a program, store data and to communicate. They are widely used in industry, such as in cars, smartphones, and clocks. The choice

of the microcontroller is not inconsequential. Indeed, the memory size and the processor determine the capacity of a microcontroller and consequently impact the implementation of the program. For example, in a program in C, a variable *var* of type *uint16_t* cannot be directly managed by a 8-bit processor as the register size is 8-bit. So *var* is spread among two registers what significantly impacts the corresponding leakage and thus the attack.

For our experiments, we used two 32-bit microcontrollers namely the ARM SecurCore SC300 and the STM32F4.

ARM SecurCore SC300 Only used in the attack presented in Chapter 6, the ARM SecurCore SC300 is a microcontroller based on the ARM Cortex-M3 processor with additional security features to help designers to produce secure code advanced forms of attack. To lighten the reading, this device will be referred to as Cortex-M3 in the remainder.

STFM32F4 The microcontroller STFM32F4 is based on ARM Cortex-M4. Although belonging to the same family of processors as the Cortex-M3, it is designed for high performances. Thus, the main characteristics remain, but it also provides optimized arithmetic operations and specific assembly instructions that are not available in Cortex-M3. It changes the appearance of the power measurement curves. To lighten the reading, this device will be referred to as Cortex-M4 in the remainder.

Remark 10. *NIST selected the Cortex-M4 for the post-quantum standardization process.*

During the standardization process, many candidate schemes were implemented for execution by microcontrollers. Some of these implementations are gathered in three git files called MUPQC, PQM4, and PQClean [57, 29, 28]. The final selection criterion was our research directions, like the fact that we wanted to study implementations in constant time.

By following this methodology, we selected first the ROLLO scheme, then the BIKE scheme.

5.2 Analyze of Vulnerability

Once the scheme and its implementation selected, we seek potential vulnerabilities in the latter. We will proceed in several steps to identify them, starting with a detailed understanding of the scheme.

5.2.1 Knowledge of the Scheme

Knowing the details of the scheme is essential, as it enables us to identify elements that may be particularly vulnerable, such as the private key. However, that is not all. Many

features of the schemes have an impact on implementation. For example, a multiplication of elements in $\mathbb{F}_q$, as found in Hamming metric schemes, is not implemented in the same way as a multiplication in $\mathbb{F}_{q^m}$. These differences can have an impact on the security of an implementation.

The non-exhaustive list of features we have taken into account:

- The metric: Hamming or Rank.
- Type of scheme: KEM, PKE, or signature.
- Structure of the scheme.
- Ephemeral key or not.
- Family of code.
- Mathematical attack.

Thanks to this information, we can get an initial idea of what could be vulnerable to attack. We can also eliminate possible attacks. For example, forging ciphertexts with particular shapes using the Encapsulation is not feasible when the scheme is a KEM. However, it can be beneficial in an attack on a PKE.

One of the last pieces of information we can extract from the study of the scheme's specification is what we call sensitive elements. A sensitive element is the message/shared secret or the private key, but not only. It can also be elements related to the final target, for instance, the syndrome in code-based schemes. The syndrome is often manipulated to decode during the decapsulation/decryption process to recover the initial message/shared secret. The syndrome is directly related to the private key and also from the ciphertext or a derivative of it, which is often known. It is, therefore, a starting point to execute a complete key recovery attack.

Once we have the specifics of the scheme in mind, we can move on to the implementation analysis.

5.2.2 Implementation Study

Due to the often complex code structure, analyzing an implementation can be a fastidious task. We proceed in steps to carry out this analysis, starting with identifying the programming language. The language influences how the elements of the scheme, such as the public key, are represented and typed. It leads us to our second step, understanding the format of cryptosystem elements such as the keys, the message, or the syndrome.

For example, an element a of $\mathbb{F}_{2^m}$, which is widely used in ROLLO, is represented in the authors' C implementation by an array of typed variables *uint32_t*. The representation

of elements is essential information for further analysis. We can obtain information from it that could assist in an attack.

To illustrate using with the ROLLO example, when $m = 67$, the representation of a requires a three-variable array *uint32_t*. However, the last variable will be made up of zeros except for the three low-order bits, which are unknown to us. If we know the Hamming weight $w_H(a_2)$ of this last variable, then we have $\binom{3}{w_H(a_2)}$ possible combinations instead of $\binom{32}{w_H(a_2)}$ for it.

The third step is to identify when the sensitive elements identified during the specification study are used in the implementation. During this step, we need to determine whether they are manipulated directly, which operations are used (addition, multiplication, ...), and with which other variables.

By collecting all of this information, we are then able to identify the vulnerabilities of the implementation and propose an attack.

5.2.3 Proposing an Attack

By studying the scheme and analyzing the selected implementation, we have gathered enough data to propose an attack. It is common to detect more than one vulnerability when analyzing the implementation. However, we have chosen to focus on targeting the one that poses the greatest security threat to the scheme.

In order to collect data about the vulnerability while the implementation is running, we must decide on a type of side-channel attack. Our options were power analysis or electromagnetic attacks because the constant-time countermeasure protects against timing and cache attacks.

We can influence some of the parameters transmitted as input for the execution of the scheme, such as the ciphertext for the Decapsulation/Decryption. It enables specific behavior from the implementation to be obtained during its execution but this was not needed for the vulnerabilities we found.

However, it was necessary to find a way to trace back to the targeted element. Vulnerabilities can provide information on sensitive elements linked to the target but not directly on it. Also, side channels may only provide partial information about the targeted sensitive element. We then need to use our scheme knowledge to finalize the attack proposal.

After establishing the attack procedure, we proceed to the experimental phase.

5.3 Experimentation

Our work focused on power analysis attacks, which are simple to set up and reproduce. In this section, we discuss the tools we employed to measure power consumption and the

methods used to interpret the power measurements in the attacks presented in Chapter 6 and Chapter 7.

5.3.1 Setting-Up the Experimentation

To record power consumption, we connected a device to an oscilloscope. The experiments in this manuscript were conducted in two different labs, each using its own oscilloscope.

- A Lecroy SDA-725 Zi-A oscilloscope took the Cortex-M3 power measurement with a bandwidth of 2.5GHz.
- The Cortex-M4 was connected to an RTO2000 oscilloscope with a bandwidth of 3GHz.

Connecting the oscilloscope to the target (microcontroler) is simplified by the existence of motherboard designed for side-channel and fault attacks. On which, a dedicated connector is available for measuring the power consumption of the target. For instance, in our experiment on the Cortex-M4, the microcontroller is set on a CW308 UFO [58].

Figure 5.1 is a photo of the setup for a power consumption recording on the Cortex-M4.

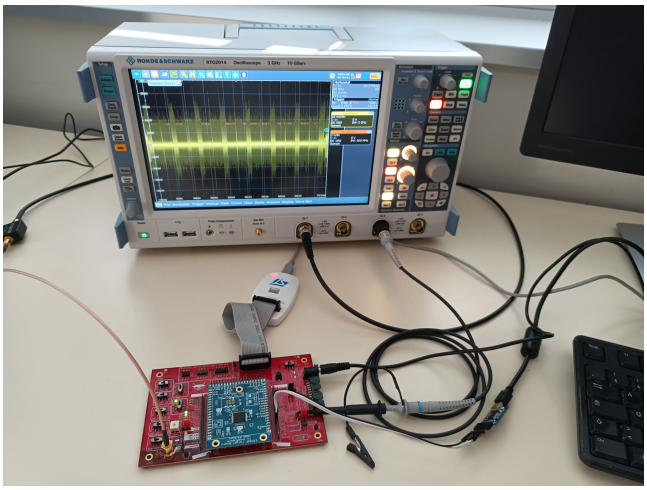


Figure 5.1: Setup for power analysis attacks on the Cortex-M4.

The tricky part of measuring power consumption is setting the parameters. They have a considerable influence on the quality of the measurements taken. Which impacts the amount of information available to the attacker. For asymmetric cryptography, its memory depth is of great importance since the computations are significantly longer than for symmetric cryptography. The better the quality of the measurements, the easier it is to carry out the attack.

The power measurements recorded by the oscilloscope are saved as arrays of points, which form a trace once plotted. It is great that we can measure power consumption, but this is only the start of the process. Indeed, the measurements obtained contain a wide

range of information, including the ones we are targeting but also some others unrelated such as the noise. It is the power consumption added by some perturbation to the initial power consumption.

The power measurements are referred to as traces, and if they are cuts of the initial trace, they are called substraces. The next step in the experiment is to identify the moments when the vulnerable operation or function is executed on the traces.

5.3.2 Detection of the Localization

Sometimes, observing the traces is enough to locate the function executions on the traces. Indeed, we can make the deduction with the information gathered during the study of the specification and the implementation, such as the number of calls and their position in the implementation.

This deduction is straightforward to verify by setting a trigger up before the function and a trigger down after the function. It allows us to know with more precision the beginning and the end of the function on the traces. It also uses one the localization of the function execution on the traces is not straightforward.

After completing this step, we have determined the location of the possible leaks. We, then, must check the leakages' existence and exploitability.

5.3.3 Verification of the Leakage Existence

We use two different methods to determine if there was indeed leakage in the identified vulnerability. We need to determine if power consumption differs based on targeted data values to do so.

Comparison of the Mean

The first method compares the average power consumption of all the possible values and is easy to implement. We separate the subtraces corresponding to precisely the function execution into groups T_i according to their values. In other words, if we are looking for the bit value, *i. e.* 0 or 1, then we create one group T_0 containing the subtraces for one execution with the bit equals 0, and the other T_1 the same way but for the bit equals 1. Then, we compute the means of each group $E(T_i)$ and plot them to observe potential differences. If there is a difference, then there is a leakage.

Computing the average power consumption has the advantage of eliminating some noise.

If there are only two groups, this comparison of averages is similar to the T-test.

SOST T-Test

T-tests are statistical tools to test the differences between two groups using their means and often also their standard deviation $\sigma(T_i)$. The Sum Of Squared pairwise t-difference T-test (SOST) is one of them, and it is defined as follows :

$$\left(\frac{E(T_0) - E(T_1)}{\sqrt{\frac{\sigma(T_0)^2}{\#T_0} - \frac{\sigma(T_1)^2}{\#T_1}}} \right)^2$$

where $\#T_i$ is the number of elements in T_i.

The result of SOST is an array of positive values that can be plotted and observable to deduce useful information. However, the maximum value of the SOST alone gives the information if there is a leakage. Indeed, a bound of 4.5 has been set for the t-test. It means that if the values of the t-test are greater than 4.5, then we consider that it is highly probable there is a leakage. If it is not, then the probability drops. Here, for SOST, with the square, the limit is $(4.5)^2 = 20.25$.

If we have proven the existence of a leakage, we now need to extract the information.

5.3.4 Extract the Data

To extract the wanted information from the subtraces, we use different techniques according to the characteristics of the leakages.

Specific Patterns and Correlation

Sometimes, the targeting function shows different power consumption patterns according to the values; for instance, double power peaks for a bit at 1 instead of one for a bit at 0. The advantage of really different patterns is that they are easy to spot once detected.

However, detecting patterns with the naked eye is limited by the number of values that need to be recovered. Indeed, it is hard for an attacker to recover over a thousand values without confusion. Nevertheless, it is possible to determine the value of a subtrace by comparing it with the patterns using the correlation. The highest correlation returned from both comparisons is, therefore, the value. Many correlation computations include the Pearson coefficient that we detail in Section 6.4.

We have been talking about leakages with an important difference in the patterns. However, there are cases where the difference is on the consumed power, rather than a difference between patterns. It requires other methods to extract the data.

Machine Learning

Some machine learning algorithms are particularly interesting for side-channel attacks. In our case, the clustering type algorithms are ideal as they separate data (here subtraces) into groups called clusters according to some characteristics that make them different. Furthermore, the clustering algorithms have the advantage of being unsupervised machine learning methods. In other words, it does not require data for training beforehand.

Remark 11. *Deep learning is also used in side-channel attacks. It is efficient in some instances but requires a large amount of data for the training.*

The limit of extracting information from power analysis attacks is that some data can be wrongly determined. Errors on the determination of data could jeopardize the success of the attack The errors are managed case by case in the application of the attack.

After selecting two schemes, BIKE in Hamming and ROLLO in rank metric, with constant-time implementations, we applied our methodology to highlight vulnerabilities and proposed and tested adapted attacks. We began with ROLLO because the rank metric implementation is a more recent concept and has not been studied as extensively than Hamming ones.

Chapter 6

ROLLO: A Single Trace Attack on a Constant-Time Gaussian Elimination

This chapter is based on a joint work with Lina Mortajine and Tania Richmond which was published in WCC 2022 [23] with an extended version in the WCC special edition of Design Codes and Cryptography [24].

We selected ROLLO [2] because there is little work on the security of implementations and that an implementation had been made available by the authors using a library for rank based cryptography [4]. We give the details of the scheme in Chapter 4. Following the methodology explained in Chapter 5, we detected a vulnerability in a regularly used function: Gauss elimination in constant-time. This function is used in all three scheme phases, in KeyGen, Enc and Dec. However, when studying the elements that could lead to an attack, we focused on the application of Gauss elimination on the syndrome when executing the RSR algorithm during decapsulation, see Algorithm 4 Subsection 4.2.3. In fact, obtaining the syndrome through side-channel attack allows us to find the private key. This following chapter described the setting up of the attack and the experimentation results. Starting from the explanation of the Gaussian elimination function, before highlighting why it is a threat with the syndrome. Finishing by the experimentation.

6.1 Gaussian Elimination

Computing the rank or the inverse of the matrix are two examples of operations executable using the Gaussian elimination method. Moreover, these two operations are commonly used in coding theory to manipulate the generator or the parity-check matrix.

Gaussian elimination, also named row reduction, is a well-known process to reduce a matrix to its row echelon form. At the end of the process, the matrix has the following characteristics:

- All rows containing only zeros are at the bottom

- The leftmost non-zero entry (pivot) of a row is to the right of the pivot of the row above it.

Sometimes, for a matrix to be in row echelon form, the pivot coefficients, also called leading coefficients, must be 1.

A sequence of elementary row operations creates this specific matrix form. Specifically, the following three:

- swap the position of two rows,
- multiply a row by a non-zero scalar,
- add a scalar multiple of another to a row.

Therefore, for any columns *col*, the Gaussian elimination can be processed by the following step:

1. Swap rows to get a non-zero pivot coefficient.
2. Eliminate the non-zero coefficient under pivot by adding the pivot row to the rows under.

Gauss-Jordan elimination is an adaptation of the Gaussian elimination process to obtain the reduced row echelon form of the matrix. The latter is a row echelon form matrix with one as the pivot coefficient, and the pivot is the unique non-zero coefficient in the column. This process uses the same row operations. It just performs more.

Gaussian Elimination in ROLLO Scheme

As a rank metric code based cryptosystem, ROLLO, Figure 4.11, make goods use of the Gaussian elimination. Indeed, the private keys $(\mathbf{h}_0, \mathbf{h}_1)$ and the errors vectors $(\mathbf{e}_0, \mathbf{e}_1)$ have a constraint on their rank weight. Thus, during the keys generation, resp. the encapsulation, the rank of the support F, resp. E, of $(\mathbf{h}_0, \mathbf{e}_1)$, resp. $(\mathbf{e}_0, \mathbf{h}_1)$, is verified by a Gaussian elimination. A less obvious use of the Gaussian elimination is to obtain the vector space EF from the syndrome $\mathbf{s}$ in the RSR algorithm during the decapulation process. Actually, the vector space is computed with the Gauss-Jordan elimination to derive a reduce row echelon form of the syndrome under its binary matrix representation form, Subsection 3.3.1.

Adaptation to a vector in $\mathbb{F}_{2^m}^n$

In the rank metric over $\mathbb{F}_{2^m}^n$, the rank of the elements in $\mathbb{F}_{2^m}^n$ is obtained through Gaussian elimination on their matrix representation. We recall that any vector $\mathbf{x}_0 = (\mathbf{x}_{0,0}, \mathbf{x}_{0,1}, \ldots, \mathbf{x}_{0,n-1}) \in \mathbb{F}_{2^m}^n$ has a matrix representation $X \in \mathcal{M}^{m \times n}$ into $\mathbb{F}_2$. However, for this

matrix, the rank is computed on the columns. Thus, Gaussian elimination is applied to the transpose matrix of X.

The Gaussian elimination process on a matrix in $\mathbb{F}_2$ uses slightly modified operations except for the position swap. Multiplication and addition are operations with different properties in $\mathbb{F}_2$ than in the general case. Indeed, as the coefficients are either 0 or 1, multiplication is never used, and the addition between two rows is actually an exclusive OR (XOR). Such differences affect the implementation.

6.1.1 Algorithmic

Implementing the Gaussian elimination algorithm is quite a simple process, especially for binary matrices. It consists mainly of writing a general algorithm, *i. e.* having a Gaussian elimination algorithm that works for any matrix, and translating it into the chosen language.

Algorithm 5 is a naive Gaussian elimination for any binary matrix M of size $(n \times m)$. We denote by $M_{i,*}$ the i-th row of M and by $M_{i,j}$ the element at the i-th row and j-th column of M. The symbol $\oplus$ represents the XOR between two elements or two rows of M.

Algorithm 5: Naive Gaussian elimination

Input: $M \in \mathcal{M}_{n \times m}$
Output: M in reduced row echelon form

```
1  dim ← 0
2  for col ← 0 to m − 1 do
3      pivot ← minimum(dim, n − 1)
4      while M_{pivot,col}! = 1 and pivot! = n do
5          pivot ← pivot + 1
6      if pivot < n then
7          pivot_row ← M_{pivot,*}
8          M_{pivot,*} ← M_{dim,*}
9          M_{dim,*} ← pivot_row
10         for j ← 0 to n − 1 do
11             if j! = dim then
12                 if M_{j,col} == 1 then
13                     M_{j,*} ← M_{j,*} ⊕ M_{dim,*}
14     dim ← dim + M_{dim,col}
```

Algorithm 5 consists of two main steps executed for each column of M. It starts by searching the pivot in the columns. If the pivot exits, *i. e.* $pivot \neq n$, it swaps the rows. Directly follow by the elimination of the ones in the column for the other rows. The variable dim keeps the row position where the next pivot is supposed to be. In other

words, *dim* is set at 0 at the beginning and only increased if the column treat has a pivot. If the matrix is full rank *dim* is equal to *col* or $n-1$ if $m > n$.

Because of its simplicity, Algorithm 5 can be translated from algorithmic to any language by seeing the matrix M as a two dimensional array. However, using the naive version of Gaussian elimination is a bargain to attackers. Implementing this algorithm without security precautions make it vulnerable to timing attacks. For example, an attacker might determine the number of ones in a column by computing the time of execution. Nonetheless, it can be adapted to a constant-time version by fulfilling the following two conditions:

- Finding the pivot must have a running time independent of its position
- The elimination operation must be executed independently of $M_{j,col}$

There are many possibilities to fulfill these two conditions. However, in this work, we only introduce the constant-time version proposed by the authors of ROLLO. To simplicity purpose, we refer to the latter by constant-time algorithm.

Getting a pivot row with pivot coefficient A particularity of constant-time algorithm is that it does not swap the initial pivot row, $M_{dim,*}$ in Algorithm 5, and the row $M_{j,*}$ containing the pivot. Instead, once $M_{j,*}$ is detected, the pivot row takes the result of the pivot row XOR to $M_{j,*}$. But $M_{j,*}$ is not modified in this step. Thus, in constant-time algorithm, setting the pivot row with the pivot is a very different procedure than in naive Gaussian elimination.

Algorithm 6 presents a simplified version of the constant-time pivot row setting. The implementation details are explained later in this section. Algorithm 6 introduces a new notation, $\otimes$, which denotes the multiplication between a scalar in $\mathbb{F}_2$ and a vector in $\mathbb{F}_2^m$.

Algorithm 6: *Pivot_setting*

Input: $M \in \mathcal{M}_{n\times m}$, *col* a column position, and *dim*
Output: M with the pivot coefficient set

```
1 pivot ← minimum(dim, n − 1)
2 for row ← 0 to n − 1 do
3 |   mask ← M_{pivot,col} ⊕ M_{row,col}
4 |   if row < pivot then
5 |   |   M_{pivot,*} ← M_{pivot,*} ⊕ (mask ⊗ M_{row,*})
6 |   else
7 |   |   dummy ← M_{pivot,*} ⊕ (mask ⊗ M_{row,*})
```

The method in Algorithm 6 is to go through the whole column and perform the same operation. The effect on the pivot row depends on a condition and a variable. The first ensures that no unwanted ones are added to the columns already in reduced row echelon form. In other words, for a row above the pivot row in the matrix, the result of the

operation is allocated to a dummy variable. That way, the row is not affected by the operation. The variable $mask$, line 6, determines whether $M_{row,*}$ is added to the pivot row. The scalar multiplications with $mask$, lines 5 and 7, return $M_{row,*}$ only if $mask$ is 1. This operation makes it possible to set up the pivot to the pivot row if it does not already have the pivot. Indeed, if $M_{pivot,col}$ is 0, and $M_{row,col}$ is 1, then $mask$ is one. Consequently, $M_{pivot,*}$ is xored with $M_{row,*}$, and $M_{pivot,col}$ is now 1. Furthermore, the three other combinations to allocate $mask$ do not affect $M_{pivot,col}$ see Figure 6.1.

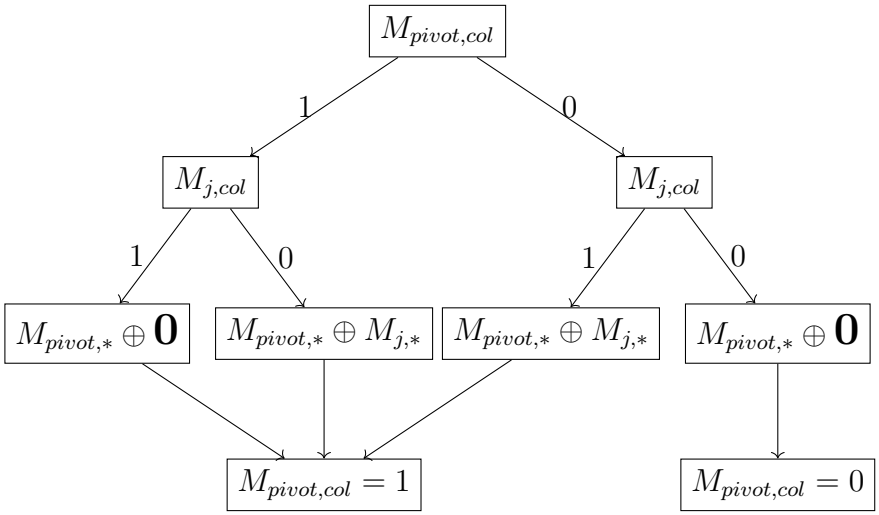


Figure 6.1: Affect on the pivot row according to $M_{row,col}$ and $M_{pivot,*}[xcol]$ values

At the end of the execution of Algorithm 6 the pivot is on the pivot row if it does exist. In any case, the non-zero coefficient elimination in the column except for the pivot row follows Algorithm 6.

Removing non zeros coefficient The elimination step in the naive Gaussian elimination needs fewer modifications to be in constant-time. Indeed, the constant-time is partially insured as the algorithm got through the whole column. It is only to get ride of the $M_{row,col}$ dependence, *i. e.* executing the same operation for each row $M_{row,*}$ for $row < n$.

Similarly to Algorithm 6, Algorithm 7 gives the idea of the operations uses without the implementation details. The latter modifies the row treat, if necessary, by XORing itself with the pivot row. In Algorithm 7, there are two conditional branches. The first one to not impact the pivot row in the process, otherwise the pivot row is XORed with itself. While the second is similar to Algorithm 6, the condition determines the destination of the XOR result. The condition is always fulfill if $n > m$. The variable $mask$ is allocated to $M_{row,col}$ *i. e.* either 0 or 1. In other words, if the element at the row and for the column treat is a nonzero coefficient, the pivot row is xored to it to set the element at zero.

Successively executing Algorithm 6 then Algorithm 7 for all the columns contained in the matrix M makes the Gaussian elimination in constant-time. Algorithm 8 shows the

Algorithm 7: *Elimination* Nonzeros elements elimination

Output: $M \in \mathcal{M}_{n \times m}$; col a column position, dim, $pivot$
Input: M in reduced row echelon form

```
for row ← 0 to n − 1 do
    if row! = pivot then
        mask ← M_{row,col}
        if dim < size then
            M_{pivot,*} ← M_{row,*} ⊕ (mask ⊗ M_{pivot,*})
        else
            dummy ← M_{row,*} ⊕ (mask ⊗ M_{pivot,*})
```

latter.

Algorithm 8: Gaussian Elimination in constant-time

Input: $M \in \mathcal{M}_{n \times m}$
Output: M in reduced row echelon form

```
dim ← 0
for col ← 0 to m − 1 do
    pivot ← minimum(dim, n − 1)
    Pivot_setting(M, col, dim)
    Elimination(M, col, dim, pivot)
    dim ← dim + M_{pivot,col}
```

6.1.2 Implementation

In the official ROLLO implementation, Algorithm 8 is slightly modified to be adapted efficiently to the C language. The matrix M is considered to be a two-dimensional array. More specifically, a row is an array of $\lceil \frac{m}{32} \rceil$ 32-bit elements, here $uint32\_t$, referred as *rbc_m_elt*. Thus M is a array of n *rbc_m_elt*.

Algorithm 8 calls three specific functions to manipulate the *rbc_m_elt*, namely:

- *rbc_m_elt_add*
- *rbc_m_elt_set_mask1*
- *rbc_m_elt_get_coefficient*

rbc_m_elt_get_coefficient and *rbc_m_elt_set_mask1* are exclusively used to manipulate the variable *mask*. Respectively to allocate the values and to compute $M_{pivot,*} \times mask$ or $M_{row,*} \times mask$. The former function, Listing 6.1, returns the *col*-th (index) value in $M_{row,*}$ (e) by using shifts and logical AND.

```
/**
 * \fn uint8_t rbc_m_elt_get_coefficient(const rbc_m_elt e, uint32_t index)
 * \brief This function returns the coefficient of the polynomial <b>e</b>
   at a given index.
 *
 * \param[in] e rbc_m_elt
 * \param[in] index Index of the coefficient
 * \return Coefficient of <b>e</b> at the given index
 */
uint8_t rbc_m_elt_get_coefficient(const rbc_m_elt e, uint32_t index) {
  uint32_t w = 0;

  for(uint8_t i = 0 ; i < RBC_M_ELT_DATA_SIZE ; i++) {
    w |= -((i ^ (index >> 5)) == 0) & e[i];
  }

  return (w >> (index & 31)) & 1;
}
```

Listing 6.1: rbc_m_elt_get_coefficient

The scalar multiplication between $mask$ and $M_{row,*}$ or $M_{pivot,*}$ (e1), Listing 6.2, is executed in $\lceil \frac{m}{32} \rceil$ steps. One for each element in *rbc_m_elt*. If $mask = 1$ then $o[i]$, a *rbc_m_elt*, takes the value $e1[i]$. When $mask = 0$ then $o[i]$ is set to $e2[i]$. In the Gaussian elimination $e2$ is always 0 and o is called *tmp*.

```
/**
* \fn void rbc_m_elt_set_mask1(rbc_m_elt o, const rbc_m_elt e1, const
   rbc_m_elt e2, uint32_t mask)
* \brief This function copies either e1 or e2 into o depending on the mask
   value
*
* \param[out] o rbc_m_elt
* \param[in] e1 rbc_m_elt
* \param[in] e2 rbc_m_elt_n* \param[in] mask 1 to copy e1 and 0 to copy e2
*/
void rbc_m_elt_set_mask1(rbc_m_elt o, const rbc_m_elt e1, const rbc_m_elt
   e2, uint32_t mask) {
  for(uint8_t i = 0 ; i < RBC_M_ELT_SIZE ; i++) {
    o[i] = mask * e1[i] + (1 - mask) * e2[i];
  }
}
```

Listing 6.2: rbc_m_elt_set_mask1

Once *tmp* (e1) is set by the previous function, then it is XORed to either $M_{pivot,*}$ or $M_{row,*}$ (e2) with the *rbc_m_elt_add* function, Listing 6.3. The function executes a XOR between e1 and e2 for every $\lceil \frac{m}{32} \rceil$ elements contains in the *rbc_m_elt*.

```
/**
 * \fn rbc_m_elt_add(rbc_m_elt o, const rbc_m_elt e1, const rbc_m_elt e2)
 * \brief This function adds two finite field elements.
 *
 * \param[out] o Sum of <b>e1</b> and <b>e2</b>
 * \param[in] e1 rbc_m_elt
 * \param[in] e2 rbc_m_elt
 */
void rbc_m_elt_add(rbc_m_elt o, const rbc_m_elt e1, const rbc_m_elt e2) {
  for(uint8_t i = 0 ; i < RBC_M_ELT_SIZE ; i++) {
    o[i] = e1[i] ^ e2[i];
  }
}
```

Listing 6.3: rbc_m_elt_add

Those three functions are all called multiple times during the Gaussian elimination and are the primary possible source of leakage.

6.2 Theoretical Attack

Among the various sensitive elements manipulated by the Gaussian elimination function, see Section 6.1, the syndrome $\mathbf{s}$ is the most relevant to attack as it is directly related to the private key. Indeed, the latter is computed as follows:

$$s = h_0 \times c \mod P.$$

with h_0 part of the private key, see Section 4.2. Furthermore, P is fixed and given in the specification of ROLLO [2]. The last element is the ciphertext c that can be intercepted, so it is considered to be known. Thus, if we obtain the syndrome by inversing the computation we recover h_0 In other words, we recover half of the private key. The last part of the private key, *i. e.* h_1, is also recovered with the public key h and the relation $h = h_0^{-1} * h_1 \mod P$.Thus, attacking the syndrome leads to a full key recovery attack. Therefore, the remainder of the chapter will only consider the case of the Gaussian elimination on the syndrome on its binary matrix form.

As a reminder, an $m \times n$ matrix in $\mathbb{F}_2$ can be constructed from a vector in $\mathbb{F}_{2^m}^n$ using a basis of $\mathbb{F}_q^m$, see Subsection 2.3.1. Thus, we have a binary matrix S constructed from the vector syndrome $\mathbf{s}$. However, as previously explain in Subsubsection 6.1 the Gaussian elimination is apply on the transpose matrix, *i. e.* here S^T. To simplify the notation, S refers to the transpose matrix representation of the syndrome $\mathbf{s}$. Therefore, the syndrome matrix in ROLLO is defined as follows:

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,m-2} & s_{0,m-1} \\ s_{1,0} & & \cdots & & s_{1,m-1} \\ \vdots & & \ddots & & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-2} & s_{n-1,m-1} \end{pmatrix}.$$

6.2.1 Side-Channel Information

Studying the C implementation gives many possibilities for leakages. For example, one XOR between two rows of S could give a Hamming distance leakage type. Which could be interesting to exploit as directly related to the syndrome. However, another operation attracts our attention:

$$tmp = (e0 \otimes mask) \oplus (e1 \otimes (1 - mask)) \tag{6.1}$$

As a matter of fact, in Operation 6.1, the potential leakage depends on the given parameters. Indeed, we notice that for the Gaussian elimination, $e1$ in Operation 6.1 is always set to zero, while $e0$ is a non-zero element most of the time. Thus, as $mask$ is either 0 or 1, Operation has two behaviors :

$$tmp = \begin{cases} (e0 \otimes 1) \oplus (0 \otimes 0) \text{ if } mask = 1 \\ (e0 \otimes 0) \oplus (0 \otimes 1) \text{ if } mask = 0 \end{cases} \tag{6.2}$$

In the first case, $mask = 1$, there is one multiplication with non-zero elements and one with two zero elements. In the second case, $mask = 0$, both are executed with a zero element. Since multiplying with a zero requires less power than multiplying with two non-zero values, they behave differently.

Detecting the difference between the two cases allows us to determine the $mask$ values. It gives us intermediate information about the syndrome for the columns and the rows. Nonetheless, we need to transform this information to recover S, the syndrome matrix.

6.2.2 Impact of $mask$ on S

To exploit the information on $mask$ values to recover $\mathbf{s}$, we first need to understand, in detail, the relation between the two. Indeed, we know that if $mask$ is 1, then S is modified, but we need to be more precise to reconstruct the syndrome. To do so, we mathematically represent the relation between $mask$ values and the syndrome matrix S.

For the explanation purpose, we introduce the notation S_{col} which refers to the syndrome matrix after the execution of the Gaussian elimination for the $col - th$ column. In other words, after the execution of the col-th iteration in Algorithm 8. We also introduce S'_{col-1}, the syndrome matrix right after the execution of Algorithm 6 and before Algo-

rithm 7 during the *col*-th iteration of Algorithm 8. During the two inner algorithm, the variable *mask* is used but the information given and the impact are different.

Pivot Detection Function

In Algorithm 6, in the line 3 the variable *mask* is set to the result of $M_{pivot,col}$ XOR to $M_{row,col}$. A little further on, in the line 5, the pivot row $M_{pivot,*}$ becomes the result of $M_{pivot,*}$ XOR to $mask \times M_{row,*}$. Thus, we have two pieces of information in addition to *mask* values. First, only the pivot row is modified. Second, *mask* depends of the xor of two values. So if *mask* is 1,then both values are identical. Otherwise, they are different. The former is the most helpful information with *mask* for the mathematical representation of the Gaussian elimination process.

We define $\sigma_{col} = (\sigma_{col,0}, \sigma_{col,1}, \ldots, \sigma_{col,n-1})$ all *mask* values obtained during the execution of Algorithm 6 for the *col*-th column. Let us write the Algorithm 6 execution for the *col*-th as a multiplication. To complete that we introduce a $n \times n$ binary matrix J_{col} corresponding to the impact of execution of Algorithm 6 for the *col*-th column such that $S'_{col-1} = J_{col} \times S_{col-1}$. We begin $J_{col} = I_n$ and modify it by replacing *col*-th row by σ_{col} as follows:

$$J_{col} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & & & \vdots \\ \sigma_{col,0} & \cdots & \sigma_{col,col} & \cdots & \sigma_{col,n-1} \\ \vdots & & & \ddots & \vdots \\ 0 & & \cdots & & 1 \end{pmatrix}$$

Thanks to this construction J_{col}, the multiplication $J_{col} \times S_{col-1}$ only impacts the *col*-th row, *i. e.* the pivot row, as in Algorithm 6. However, the current multiplication $J_0 \times S_{col-1}$ also XOR the rows of S_{col-1} with label less than *col* to the pivot row which differs from Algorithm 6.

To exactly reproduce the impact of Algorithm 6 on S_{col-1}, $\sigma_{col,col}$ needs to be replaced by a 1. If not, the initial value of the pivot row is erased as $\sigma_{col,col}$ is always equal to 0. Furthermore, all the $\sigma_{col,i}$ for $i < col$ are replaced by 0. Which is equivalent to the conditional branching in Algorithm 6. Therefore

$$J_{col} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ \vdots & \ddots & & & \vdots \\ 0 & \cdots & 1 & \cdots & \sigma_{col,n-1} \\ \vdots & & & \ddots & \vdots \\ 0 & & \cdots & & 1 \end{pmatrix},$$

and the syndrome matrix after Algorithm 6, S'_{col-1}, is computed as follows:

$$S'_{col-1} = J_{col} \times S_{col-1}.$$

Elimination in the matrix As for Algorithm 7, $mask = M_{row,col}$ and $M_{row,*}$ takes itself XOR to $mask \times M_{pivot,*}$. Similarly to Algorithm 6, the elimination function, Algorithm 7, can be mathematically represented by the multiplication of a matrix J'_{col} by S'_{col-1}. J'_{col} is constructed as J_k but in its transposed form. In other word, the $\sigma'_{col} = (\sigma'_{col,0}, \sigma'_{col,1}, \ldots, \sigma'_{col,n-1})$ are put on the column corresponding to the pivot row as follow:

$$J_{col} = \begin{pmatrix} 1 & & \sigma_{col,0} & & 0 \\ \vdots & \ddots & \vdots & & \\ 0 & \cdots & \sigma_{col,col} & \cdots & \vdots \\ \vdots & & \vdots & \ddots & \\ 0 & & \sigma_{col,n-1}s & & 1 \end{pmatrix}$$

In J'_k, $\sigma'_{col,col}$ is also set at 1, as Algorithm 7 never computes it. Nonetheless, for J'_k all the values in σ' are kept. Thus,

$$S_{col} = J'_{col} \times S'_{col-1}$$

is the mathematical equivalence of Algorithm 7 execution.

Full Gaussian elimination is, therefore, the following product:

$$S_{m-1} = \prod_{col}^{n-1} J'_{col} \times J_{col} \times S$$

From this representation, we can mount an attack to recover the syndrome matrix S with the information from the different σ'_{col} and σ_{col}.

6.2.3 Recovering the Matrix S

Reconstructing S with the knowledge of σ_{col}, σ'_{col} and the mathematical relation $S_{col} = J'_{col} \times J_{col} \times S_{col-1}$ is straightforward. For explanation purposes, we will start with the specific case $col = 0$. In other words, the first execution of the Gaussian elimination, Algorithm 8.

We suppose that we have σ_0 and σ'_0. And we know by construction that $\sigma'_{0,i} = S'_{-1,0,0}$ for $i \geq 0$ and $S'_{-1} = J_0 \times S$. However, J_0 does only impact $S_{0,*}$ so $\sigma'_{0,i} = S_{0,i} = s_{0,i}$ for $i > 0$. Therefore, we already obtain all the values in the 0-th of the syndrome except one $S_{0,0} = s_{0,0}$.

To get the last value, we start from the fact that $S'_{-1,0,0} = 1$. By definition $S'_{-1,0,0} \stackrel{\circ}{=} s_{0,0} \oplus s_{1,0} \times \sigma_{0,1} \oplus \cdots \oplus s_{n-1,0} \times \sigma_{0,n-1}$. Thus, by solving the following linear system of

equation

$$\begin{cases} s_{0,0} \oplus s_{1,0} \times \sigma_{0,1} \oplus \cdots \oplus s_{n-1,0} & = 1 \\ s_{1,0} & = \sigma'_{0,1} \\ \vdots & = \vdots \\ s_{n-1,0} & = \sigma'_{0,n-1} \end{cases} \tag{6.3}$$

we obtain the missing value as all the $\sigma_{0,i}$ are known. Actually, this system is generated by the product of J_0 by the 0-th column of S, denoted $S_{*,0}$ (generalize to $S_{*,col}$ for any column *col* of S). To simplify, the system of equation is $J_0 \times S_{*,0} == (\sigma'_0)^T$. Therefore, to generalise to any column *col* we have $J_{col} \times S_{col-1,*,col} == \sigma'_{col}$ as system of equation. It is S_{col-1} instead of S, however $S_{col-1} = (\prod_i^{col} J'_i \times J_i) \times S$ so the system to solve can be expend to recover $S_{*,col}$ as follows:

$$(J_{col} \times \prod_i^{col} J'_i \times J_i) \times S_{*,col} == \sigma'_{col}$$

By solving this system of equations for each column $S_{*,col}$, we are in capacity of reconstructing the syndrome matrix S. As remainder, knowing the syndrome matrix is equivalent to knowing the syndrome vector and by consequence recover the private key.

Remark 12. *In this section, we suppose that after the execution of Algorithm 6, the pivot coefficient is set at* 1*. However, it is possible not to have a pivot coefficient, i. e. the value is* 0*. We are able to detect these cases by the simple fact that all the mask obtained from Algorithm 6 for the rows under the pivot row are* 0*.*

6.2.4 Toy Example

Let us take a small example, with $m = 5$ and $n = 7$, to illustrate the recovery of the matrix syndrome from *mask*.

Assume we want to recover the following matrix

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

corresponding to the syndrome $\mathbf{s} \in \mathbb{F}_{2^5}^7$.

$\longrightarrow$ The searched matrix is defined as

$$S = \begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} & s_{0,4} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} & s_{1,4} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} & s_{2,4} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} & s_{3,4} \\ s_{4,0} & s_{4,1} & s_{4,2} & s_{4,3} & s_{4,4} \\ s_{5,0} & s_{5,1} & s_{5,2} & s_{5,3} & s_{5,4} \\ s_{6,0} & s_{6,1} & s_{6,2} & s_{6,3} & s_{6,4} \end{pmatrix}$$

After the execution of the Gaussian elimination process, we guess from the power consumption analysis the masks in Algorithm 6 and Algorithm 7:

1. masks in the first loop for each column:

$$(*,1,1,1,0,0,0),(1,*,1,0,1,1,0),(1,0,*,0,1,0,1),(1,1,1,*,0,1,1),$$
$$(1,1,1,0,*,1,0)$$

2. masks of the second loop for each column:

$$(*,0,0,0,1,1,1),(1,*,1,1,0,0,1),(0,1,*,1,0,1,0),(1,1,1,*,0,1,0),$$
$$(1,1,1,0,*,1,1),$$

   with $*$ the pivot. As explained in Subsection 6.2.2, the $*$ correspond to the $\sigma_{col,col}$ which are replaced by one.

Let us focus on recovering the two first columns of the syndrome matrix. The recovered masks vector of Algorithm 6 $(1,1,1,1,0,0,0)$ provides the additions on the pivot row 0:

$$J_0 \times S = \left(\begin{array}{c|cccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline \mathbf{0} & \multicolumn{6}{c}{I_6} \end{array}\right) \times S_{0,*} = \begin{pmatrix} s_{0,0}+s_{1,0}+s_{2,0}+s_{3,0} \\ s_{1,0} \\ s_{2,0} \\ s_{3,0} \\ s_{4,0} \\ s_{5,0} \\ s_{6,0} \end{pmatrix}.$$

The masks vector of the Algorithm 7 $\sigma'_0 = (1,0,0,0,1,1,1)$ is the solution vector of the system of linear equations where $s_{i,j}$ are unknowns. Thus, by applying a linear solver on the system

$$J_0 \times S_0 = (1\ 0\ 0\ 0\ 1\ 1\ 1)^T,$$

we find the solution $(1,0,0,0,1,1,1)$, which corresponds to the first column of the syndrome matrix. At the end of the process of the first column, we have the matrix

$$S_0 = \left(\begin{array}{c|c} (\sigma'_0)^t & \mathbf{0} \\ \hline & I_6 \end{array}\right) \times J_0 \times S.$$

For the second column, the recovered masks vector of the first loop is $(1,0,1,0,\ 1,1,0)$. However, as explained in Section 6.1.1, only the rows for which the index row is greater than the index pivot row are added to the pivot row. Thus, in the recovered masks vector,

σ_1, we replace one by zero for $i < 1$. This gives us the vector $\sigma_1 = (0, 0, 1, 0, 1, 1, 0)$. In addition, the masks vector of the second loop is $\sigma'_1 = (1, 1, 1, 1, 0, 0, 1)$. We can then apply a linear solver on the system

$$\underbrace{\left(\begin{array}{c|cccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline \mathbf{0} & \multicolumn{6}{c}{I_5} \end{array}\right)}_{J_1} \times S_{0,1,*} = (1\ 1\ 1\ 1\ 0\ 0\ 1)^t,$$

with $S_{0,1,*}$ the column 1 of the matrix S_0.
The result of this system corresponds to the vector $(1, 0, 1, 1, 1, 1, 0)$.
At the end, we have the matrix

$$S_1 = \left(\begin{array}{cc|c} 1 & (\sigma'_1)^t & \mathbf{0} \\ \mathbf{0} & & I_5 \end{array}\right) \times J_1 \times S_0.$$

We perform the same for the three remaining columns.

From the theoretical part the attack proposed is efficient. However, the success of the attack in practice depends of the information actually obtained during the side-channel attack.

6.3 Side-Channel Attack

To perform our side-channel attack, we choose not to interact with the scheme execution at any moment. In other words, we are just "listening" to the implementation as its runs. And we do not modifiy the ciphertext given to the decapsulation process.Indeed, we target the syndrome which depends on the ciphertext $\mathbf{c}$. However, the ciphertext changes at each execution, so the syndrome is also different at each execution. Therefore, we have a unique execution to obtain enough information to recover the private key. As we use power analysis to attack, see Chapter 3, it will necessarily be a Single Trace Attack (STA).

We tried our attack, firstly on the Cortex-M3, and as the results were conclusive, we also tried on the Cortex-M4. Both have an ARMv7 architecture, but the way operations, such as addition or multiplication, are executed are different, see Section 5.1. So, even if we attack the M3, it is possible not to be able to execute it on the M4. Furthermore, NIST chose the Cortex-M4 as a reference microcontroller.

From experimentation, we conclude that the Cortex-M4 is also sensitive to the attack through power analysis. However, the difference in power consumption does exist between

both microcontrollers. Let us start with the Cortex-M3.

6.3.1 Cortex-M3

To test the practicability of our attack on the Cortex-M3, we sightly modified the reference implementation to adapt it to the device.

During experimentation, we measured the power consumption of the entire decapsulation process. To detect the Gaussian elimination execution, we set a trigger right before the beginning. The measurement obtained shows a succession of Algorithm 6 execution pattern, Figure 6.2, and the Algorithm 7 execution pattern, Figure 6.3.

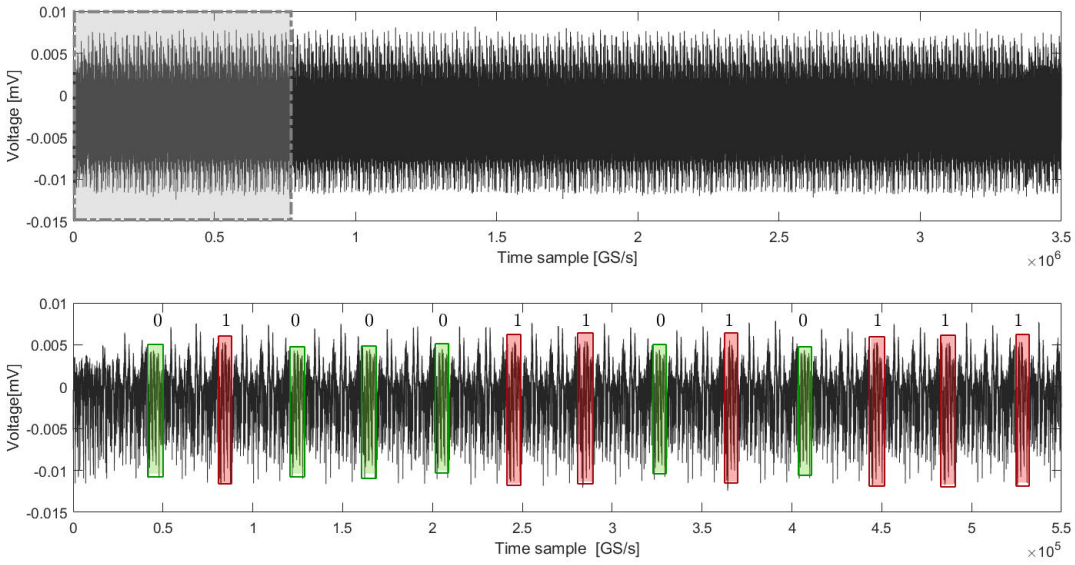


Figure 6.2: Pivot detection algorithm power consumption with *mask* values

In Figure 6.2, resp. Figure 6.3, we obverse the beginning of Algorithm 6, resp. Algorithm 7, execution. Both have a regular and repetitive pattern corresponding to the sequence of operations of each algorithm. We highlight the multiplication with *mask*. Green when $mask = 0$ and red when $mask = 1$. Looking closely at the power consumption for both cases, we observe *mask* at 1 requires more power consumption than when *mask* is 0.

As a difference does exist we are able to determine σ_i and σ'_i for $0 < i < m$. Thus, our attack works when the ROLLO-I-128 runs on a Cortex-M3.

However, we did not stop there. We also tested the practicability of our attack on a Cortex because the Cortex-M4 uses an optimized multiplication that does not exist for the M3.

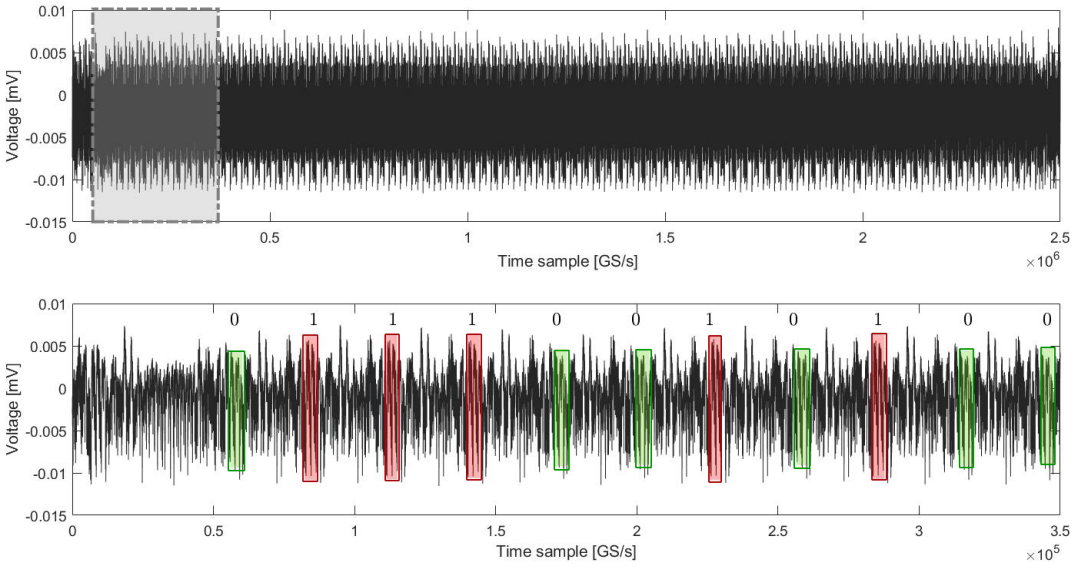


Figure 6.3: Elimination algorithm power consumption with *mask* values

6.3.2 Cortex-M4

For all our experiments on the Cortex-M4, we used the implementation provided on the PQM4 git without any modification [29]. It corresponds to the official implementation. Same as the one attack on the Cortex-M3 but without any adaptation.

Figure 6.4 and Figure 6.5 show the power consumption of one execution of Algorithm 6 and Algorithm 7 with a zoom at the beginning. The targeted operation is highlighted in red when *mask* is equal to 1 and green when it is equal to 0.

Apart from the patterns, we observe a difference in the trace length for the Algorithm 6 between Figure 6.2 (Cortex-M3) and Figure 6.3. In the former, the full trace has a length of 3.5×10^6, and the zoom has a length of 5.5×10^5 with 13 patterns. While for the Cortex-M4, the full trace measures around 5×10^4 with 19 pattern on the zoom trace of length 1.2^4. It is the same for the second loop. Consequently, the power consumption corresponding to the multiplication has a tiny length making it more harder than for the Cortex-M3 to detect the leakage and to determine the *mask* value. Nonetheless, the leakage exists for both algorithm and allows us to obtain the wanted information.

In the Cortex-M4, the multiplication power consumption decreases when $mask = 0$ while it stays high or increases when $mask = 1$. It is the most visible on the trace for Algorithm 7, Figure 6.5. The multiplication of the power consumption in Figure 6.4 shows the same behavior but on a smaller scale. To be sure that *mask* values are related to the decrease of the power consumption, we computed the average power consumption for both possible value of the execution of Algorithm 6, resp. Algorithm 7, and compared them, see Figure 6.6. Each mean were computed by randomly selecting 10 algorithms execution. The average power consumption of the multiplication when $mask = 0$ is clearly decreasing in

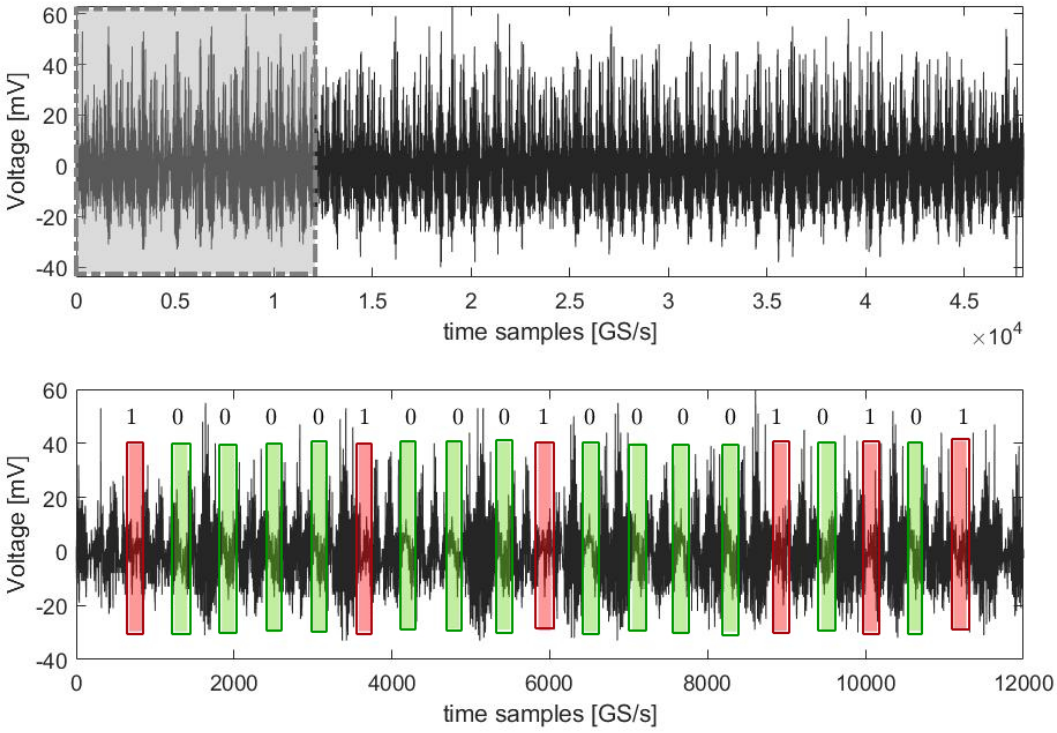


Figure 6.4: Pivot detection algorithm power consumption with *mask* values

both loops. Between 380 and 480 in Figure 6.6a and between 240 and 300 in Figure 6.6b. However, in the latter, the power consumption for $mask = 1$ stays high or even increases while in the former it is decreasing a little, *i. e.* not as much as, and remains after.

Both algorithms leak the *mask* value during their execution on a Cortex-M4 with greater ease for the Algorithm 7. Thus our attack works on the ROLLO-I-128 implementation running on a Cortex-M4. However, we notice two things. Firstly, it will be inefficient for an attacker to look for all the *mask* values by hand or, more likely, by its eyes. Secondly, we need to be sure of the values, but a human is more likely to make mistakes, especially with Algorithm 6. It is why we have sought to automate the *mask* value detection.

6.4 Automation

Automating the detection of *mask* values for any Gaussian elimination power consumption traces must be based on power consumption for the two possible values, 0 and 1. Specifically, we need to determine for Algorithm 6 and Algorithm 7 executions if its power consumption corresponds to the behavior when *mask* is 1 or 0. One method is to compute the correlation between the pattern of each type with the execution we want to determine. We set *mask* to the value with the highest correlation between both. To do

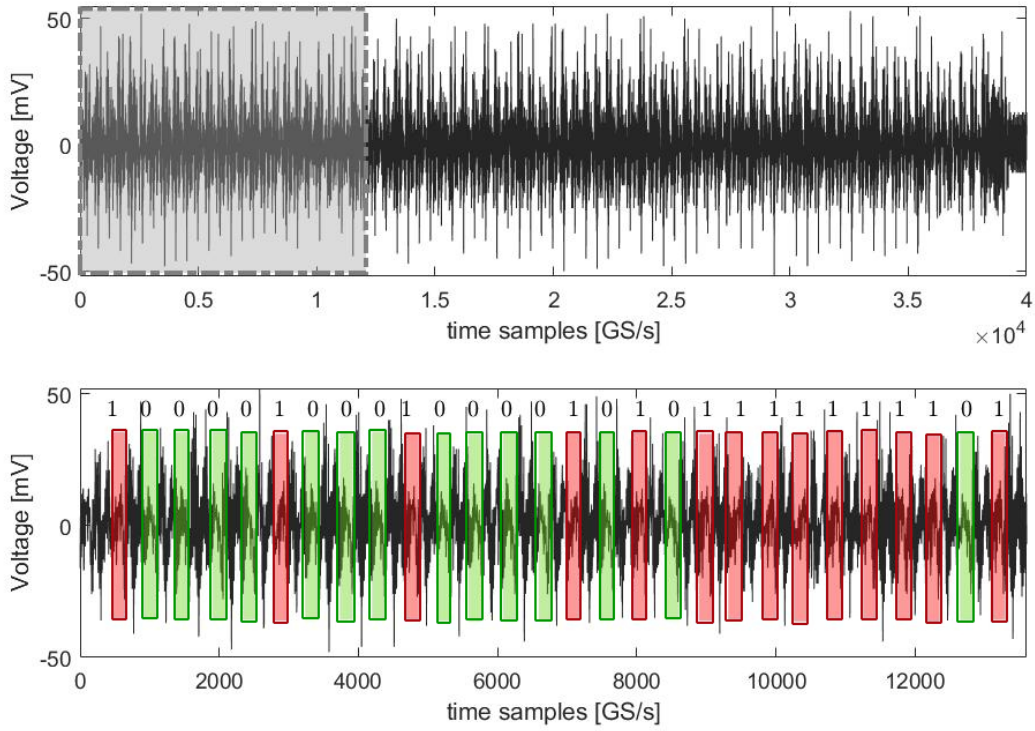


Figure 6.5: Elimination algorithm power consumption with *mask* values

so, we chose to use the Pearson correlation coefficient.

Pearson correlation Let us define w and o two samples, here the pattern of length len and the power consumption trace to determine, also of length len. The Pearson correlation coefficient (pcc) is computed as follow:

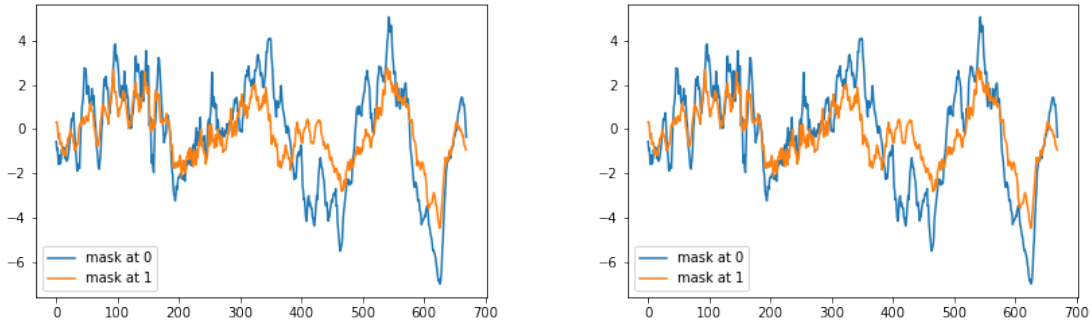
$$pcc(w,o) = \frac{\sum_{i=1,\cdots,len}(w_i - \bar{w})(o_i - \bar{o})}{\sqrt{\sum_{i=1,\cdots,len}(w_i - \bar{w})^2}\sqrt{\sum_{i=1,\cdots,len}(o_i - \bar{o})^2}}$$

with $\bar{w}$, resp. $\bar{o}$, is the sample w, resp. o, mean.

In our attack, we compute $pcc(w,o)$ twice. One with w the pattern for $mask$ at 0 ($pcc(w_0,o)$) and the other with w the pattern for $mask$ at 1 ($pcc(w_1,o)$). Both have the same o. To compare $pcc(w_0,o)$ and $pcc(w_1,o)$, we use their absolute value. However, this technique requires that o corresponds exactly to one execution of the algorithms, while we aim to generalise the detection of the patterns.

Generalisation of the detection We note that the patterns for the Gaussian elimination execution never appear elsewhere in the decapsulation process of ROLLO. So, we use this to our advantage by computing pcc for any block of length len in the full power consumption of the decapsulation. We illustrate the idea in Figure 6.7.

Figure 6.8 shows the totality of pcc computed for a small part of the decapsulation



(a) Means of the full pivot detection algorithm execution for *mask* equal to 0 (blue) and to 1 (orange)

(b) Means of the full elimination algorithm execution for *mask* equal to 0 (blue) and to 1 (orange)

Figure 6.6: Comparison of the average power consumption of the First and the Second Inner For Loop executions

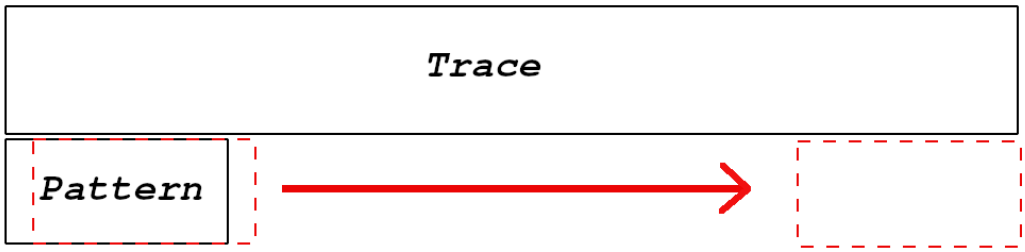


Figure 6.7: Computing the Pearson correlation coefficients for the full decapsulation

process. We notice that the highest n coefficients have a regular separation. They correspond to the Gaussian elimination. Thus, we only keep them by eliminating those under 0.8, see Figure 6.9.

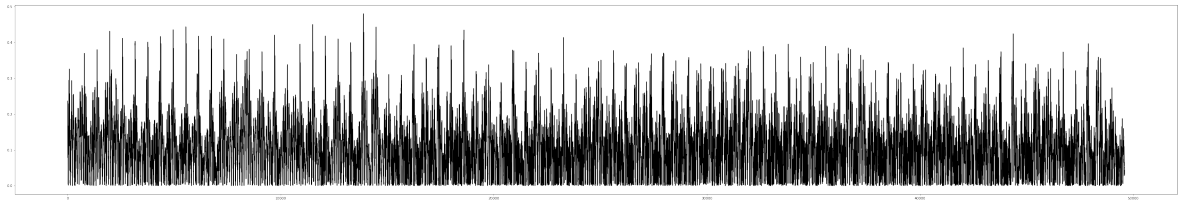


Figure 6.8: Pearson correlation coefficient for Algorithm 6 with mask at 1 pattern

The process is repeated four times, one for each pattern of the two algorithms. A function running through the coefficients and selecting the n highest values determines the *mask* values for Algorithm 6. Then do the same for the $n-1$ *mask* in Algorithm 7. The function executes the process m times.

Result This method works well up to $\frac{3m}{4}$ iterations of the Gaussian elimination in constant time, Algorithm 8. The *mask* are correctly detected. However, at the end of the

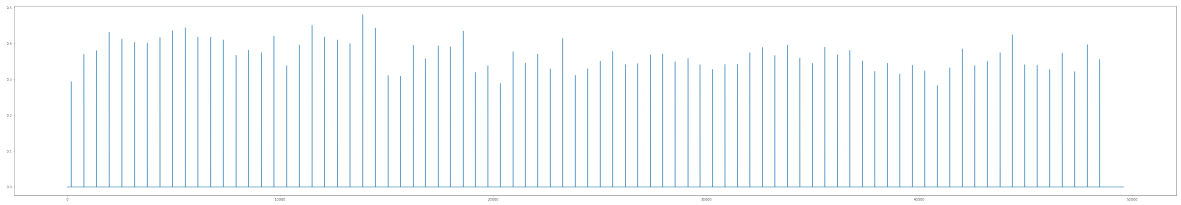


Figure 6.9: n highest Pearson correlation coefficient for Algorithm 6 with mask at 1 pattern

execution, many detection errors appear. Nonetheless, it is always on the way that a 0 is detected while *mask* is a 1. Actually, it can be easily explained by the fact that the leakage depends on the number of 1 in the element multiply by *mask*, and the further we go through Gaussian elimination, the less they have of 1. Therefore, the pattern adapts for the first executions of Algorithm 8 is not anymore.

However, if we adapt the pattern to the last iterations, the same effect occurs for the first iterations. This time, some *mask* at 0 are detected as 1.

One solution could be to have two patterns, one for the beginning and one for the end of Gaussian elimination. But it requires determining the moment to change the pattern. We chose to stop our work with the Gaussian elimination automation here.

Resume and Conclusion

Resume

The important information on the attack of ROLLO schemes (ROLLO-I, ROLLO-II) are gathered here:

- Targeting the decapsulation, precisely the RSR Algorithm with the computation of the support of the syndrome.
- The function execute a Gaussian Elimination on the syndrome in its binary matrix representation.
- Exploiting a multiplication that ensure the constant time, we obtain the syndrome matrix.
- From the syndrome, we recover the private key.
- Our attack requires a single trace of power consumption and solving a system of binary equation.
- We have worked to automate the attack using the Pearson Correlation, but it was unsuccessful.

Through the ROLLO attack, we have shown that the decoder of Ideal-LRPC constant-

time decoder is particularly sensitive to power analysis attacks. As a result, there are several possibilities for further study. First, there is the question of finding a countermeasure to this attack. This subject is covered later in Chapter 8. Another approach is to extend the work on the Gaussian elimination implementation to other schemes, such as the Classic McEliece. A third possibility is to study the impact of ideal-code structure in the Hamming metric on the security of the implementation. We chose to continue with this third option. Precisely, we question the security of the decoder for QC-MDPC codes, a specific case of Ideal codes often used in Hamming metric schemes through the case of the BIKE scheme.

Chapter 7

BIKE: Combining Machine Learning and Information-Set Decoding

This chapter is based on a joint work with Nicolas Aragon, Tania Richmond and Benoît Gérard which was published in ACNS 2023 [22].

We studied the optimized, constant-time implementation for Cortex-M4 of the BIKE scheme [1]. It was interesting because it contains two versions, one totally C and one where certain parts are replaced by assembly. The study of the code revealed the existence of a weakness in the decapsulation and, more precisely, in the execution of the COUNTER function in Black-Grey Flip decoder, see Subsection 4.1.2. Combining clustering with the Information-Set Decoding we are able to recover the private key from this leakage. This chapter is divided into four sections: the first one set the background in counter function optimization, the second one develop the attack structure and the last two described the experiments of the attack on the C and the assembly implementation.

7.1 Sparse-Dense Multiplication

The COUNTER function basically compares two column vectors and returns the number of 1 in common. In BIKE, the function is executed for all the columns of the parity-check matrix H and the vector syndrome $\mathbf{s}$. However, due to the specific structure of H and $\mathbf{s}$, COUNTER is replaced by sparse-dense polynomial multiplications in the optimized implementation for Cortex-M4. In this section, we start from explaining the sparse-dense multiplication definition to introduce progressively the targeting function.

Sparse-Dense Multiplication

Multiplying two polynomials by a naive method consists of summing each coefficient of the first polynomial multiplied by all the coefficients of the second polynomial.

$$a \times b = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} a_i b_j X^{i+j}$$

To compute the product of two polynomials of degree $n-1$, n^2 integer multiplications and $n^2 - 1$ additions are required for a time complexity of $\mathcal{O}(n^2)$. With the degree of

polynomials used in BIKE, the cost is high for each polynomials multiplications. One way to reduce complexity is to take advantage of the properties of certain polynomials, in particular, the so-called sparse polynomials. A polynomial is sparse if it has a small number of non-zero coordinates relative to its degree. Consequently, a majority of the product between the integers $a_i b_j$ returns 0. This is exploited to optimize the complexity of the multiplication. Let us defines $\mathscr{L}$ as a set of non-zero coordinates in a sparse polynomial a. Therefore we compute the multiplication of polynomials with only the elements contained in $\mathscr{L}$ as follows:

$$a \times b = \sum_{l \in \mathscr{L}} \sum_{j=0}^{j<n} a_l b_j X^{l+j}$$

In this case, the number of integer multiplications is reduced to $dim(\mathscr{L}) \times n$, and the number of additions is reduced to $dim(\mathscr{L}) \times (n-1)$.

Adaptation to QC-MDPC Codes

By working with QC-MDPC codes, we have sparse polynomials in the parity-check matrix. In BIKE, see Section 4.1, the two vectors $(\mathbf{h}_0, \mathbf{h}_1)$ constitute the parity-check matrix. By mathematical construction, a vector $\mathbf{h}_i$ is isomorphic to a polynomial in $\mathbb{F}_2[X] \big/ (X^r - 1)$. $\mathbf{h}_i$ has a moderate density, so its polynomial representation is sparse. During decapsulation, the syndrome computation uses sparse-dense multiplication.

However, it is not the only potential application in the decapsulation.

In the Bit-Flipping algorithm, see Subsection 4.1.2, the parity check to obtain the error vector can be performed by multiplying the syndrome $\mathbf{s}$ with each non-zero element in $(\mathbf{h}_0, \mathbf{h}_1)$. The relation between the latter and the syndrome comparison with the columns of the parity-check matrix $H = (H_0|H_1)$ is not straightforward.

Intuitively, we can see that the element at the row-th position in the syndrome, denoted $\mathbf{s}_{row}$, is only compared to $H_{row,*}$, the row-th row of H. In other words, when $\mathbf{s}_{row}$ is 1, the counter for column col increase by one if $H_{row,col}$ is also 1. Let us represent all the counters in a single vector, **counter**, of integers. Then the counters can be incremented at once for $\mathbf{s}_{row}$ by adding $H_{row,*}$ to **counter**. Since $H_{row,*}$ is composed of $\mathbf{h}_0$, and $\mathbf{h}_1$, rotated by row positions. Then, adding $H_{row,*}$ to **counter** is equivalent to adding the vector $(\mathbf{h}_0|\mathbf{h}_1)$ with a rotation of row positions to **counter**. And these rotations can be represented as polynomial multiplications.

Let us define $\mathbb{F}_{p_1}$ as a finite field, with p_1 the first prime number such that $p_1 \geq \frac{\omega}{2}$, and $\mathbb{F}_{p_1}[Z] \big/ (Z^r - 1)$ a polynomial ring. $\{0, 1\}$ belongs to $\mathbb{F}_{p_1}$, so we rewrite the syndrome, resp. the private key, into the polynomial ring $\mathbb{F}_{p_1}[Z] \big/ (Z^r - 1)$ as follows:

$$s = \sum_{col=0}^{r-1} \mathbf{s}_{col} Z^{col}$$

resp.

$$\mathbf{H}_i = \sum_{col=0}^{r-1} h_{i,col} Z^{col}$$

.

A polynomial $count_i$ for $i \in 0, 1$ is computed as follows:

$$count_i = \sum_{row=0}^{r-1} \mathbf{counter}_{row} Z^{row} = \sum_{row=0}^{r-1} (\mathbf{H}_i \times s_{row}) Z^{row} = \sum_{row=0}^{r-1} (\mathbf{H}_{i,row} \times s) Z^{row}$$

where $i \in \{0, 1\}$ and $\mathscr{L}_i$ the set of non-zero positions in to $\mathbf{h}_i$. Finally, since H_i is a sparse polynomial with coefficients in $\mathbb{F}_2$, it takes the following form:

$$count_i = \sum_{l \in \mathscr{L}_i} \mathbf{H}_{i,l} \times s \times Z^l = \sum_{l \in \mathscr{L}_i} s \times Z^l$$

Since the polynomial s is assumed to be dense, $count_i$ is the sparse-dense multiplication result between the syndrome and the private key. Then, the parity number checked in each column of H can be obtained by multiplication. This operation is used in the implementation we have studied.

Implementation

Sparse-dense multiplication saves space memory because only the non-zero element position of the sparse polynomial needs to be kept. However, most programming languages do not have a polynomial structure, and manipulating them is hard. Hence, developers used vectors instead of polynomials. A binary vector written in an array is easier to handle. Therefore all the processes using polynomials need to be adapted to vector manipulation.

The link between vectors and polynomials is evident, but not the one between sparse-dense multiplication and syndrome rotation. To explain, let us take the $count_i$ polynomial, see Subsubsection 7.1. Remainder, $count_i = \sum_{l \in L_i} s \times Z^l$. Let us denote $s^l = s \times Z^l$ the multiplication of the syndrome by the monomial Z^l for $l \in \mathscr{L}_i$. It can be developed as follow:

$$s^l = s_0 Z^{l \mod r} + s_1 Z^{l+1 \mod r} + \ldots + s_{r-1} Z^{r-1+l \mod r}$$

.

Let $\mathbf{s}^l = (\mathbf{s}^l_0, \ldots, \mathbf{s}^l_{r-1})$ be the vector representation of s^l. Any element of $\mathbf{s}^l$ is determined from $\mathbf{s}$ as follows:

$$\mathbf{s}^l_k = \mathbf{s}_{k-l \mod r},$$

which corresponds to a rotation of the syndrome vector to the right of l positions. Therefore, each execution of the sparse-dense multiplication with the syndrome is replaced by a rotation of the syndrome in the implementation.

Algorithm 9 contains the most intuitive version of the rotation. Even if efficiency

could be improved, Algorithm 9 works perfectly. Nevertheless, it is not secure against side-channel attacks.

Algorithm 9: Syndrome rotation

Input: $\mathbf{s}$,r,l
Output: $\mathbf{s}^l$
1 **for** $k \leftarrow 0$ **to** r **do**
2 $\quad \mathbf{s}^l \leftarrow \{0\}^r$
3 $\quad \mathbf{s}^l_k = \mathbf{s}_{k-l \mod r}$

One potential weakness of Algorithm 9 is memory access. To allocate the syndrome values to $\mathbf{s}^l$, multiple memory accesses are performed to the memory allocated to $\mathbf{s}$. As a result, an attacker ($\mathcal{A}$) could achieve a cache attack. For instance, if $\mathcal{A}$ gets to know when the algorithm allocates $\mathbf{s}_k$ to $\mathbf{s}^l$ then $\mathcal{A}$ recovers l, a position in the private key $(\mathbf{h}_0, \mathbf{h}_1)$. Hence the interest in constant-time security.

An optimized constant-time implementation was proposed by Chen *et al.* [20]. The general idea is to prevent an attack by replacing the rotation of l positions. A succession of 2^j positions rotations, j determined by l, leads to the same result.

A sum of the power of two can express any integer, so $l = \sum_{k=0}^{k<j} b_k \times 2^k$ where $b_k \in \{0, 1\}$. $(b_{j-1}, \ldots, b_0)_2$ is called the binary representation of l. Thus, to obtain a l positions rotation, the process executes a $b_k \times 2^k$ positions rotation for $k \in \{j-1, \ldots, 0\}$.

Executing a succession of rotations instead of a unique one implies manipulating only one vector. In other words, the values are taken and allocated into the same vector syndrome $\mathbf{s}$. But, allocating $\mathbf{s}_{i+2^k}$ to $\mathbf{s}_i$ erases the initial value in it, which makes impossible the allocation of $\mathbf{s}_i$ later in the process. Duplicating the syndrome vector solves this problem. Let $\mathbf{ds} = (\mathbf{s}|\mathbf{s}) = (\mathbf{s}_0, \ldots, \mathbf{s}_{r-1}, \mathbf{s}_0, \ldots, \mathbf{s}_{r-1})$ be the duplicate syndrome vector of length $2r$. Thus, $\mathbf{ds}_i$ always takes the value in $\mathbf{ds}_{i+2^k}$ for increasing i. However, to keep the benefits of duplication, allocates $\mathbf{s}_i$ has to be done for $i \in \{0, min(r + 2^k, 2r - 2^k)\}$. At the end, the r first elements in $\mathbf{ds}$ correspond to the syndrome rotated of l positions.

Algorithm 10: Syndrome rotation with binary representation

Input: $\mathbf{s}$,r, $l = (b_{j-1}, \cdots, b_0)$
Output: $\mathbf{ds}_{0:r-1}$
1 $\mathbf{ds} \leftarrow (\mathbf{s}|\mathbf{s})$
2 **for** $k \leftarrow j-1$ **to** 0 **do**
3 $\quad$ **if** $b_k == 1$ **then**
4 $\quad\quad$ **for** $i \leftarrow 0$ **to** $r + 2^k - 1$ **do**
5 $\quad\quad\quad \mathbf{ds} \leftarrow \mathbf{ds}_{k+2^k}$

Algorithm 10 shows an algorithmic version of the syndrome rotation with the binary decomposition of l. In this process, it is more accurate to talk about a shift of $\mathbf{ds}$ instead of

rotation. Nonetheless, Algorithm 10 is not in constant-time due to b_k value dependencies.

Transforming Algorithm 10 into a constant-time version implies removing the dependence at b_k. In other words, executing the same operation on **ds** independently of the value contained in b_k without changing the result. For the sparse-dense multiplication in BIKE, Drucker *et al.* proposed to use a variable called $mask$, related to b_j, to construct this operation [34]. $mask$ is defined as follows:

$$mask = -b_{elt} = \begin{cases} -1 & \text{if } b_{elt} == 1 \\ 0 & \text{if } b_{elt} == 0 \end{cases}$$

where $mask$ is an unsigned element thus if $mask = -1$ then $mask$ is actually a binary full of 1. The operation is the following one:

$$\mathbf{ds}_i = (\mathbf{ds}_i \wedge \neg mask) \vee (\mathbf{ds}_{i+2^k} \wedge mask) \tag{7.1}$$

Therefore, $\mathbf{s}_i$ is allocated either with $\mathbf{ds}_{i+2^k}$ if b_k is equal to 1 or itself if b_k is equal to 0.

Remark 13. *Even if the variable is called mask, it is not a masking countermeasure as mask is not randomly generated.*

Chen *et al.* kept this method for their version of BIKE but adapted it for a Cortex-M4.

Adaptation to Cortex-M4 On the Cortex-M4 implementation, an array of 32-bit words, denoted ds, represents the duplicate syndrome, such that:

$$\mathsf{ds}_i = (\mathbf{ds}_{32*i}, \mathbf{ds}_{32*i+1}, \ldots, \mathbf{ds}_{32*i+31})$$

for $i \leq \lceil \frac{2r}{32} \rceil$ and where $\mathbf{ds}_{32*i+31}$ is the most significant bit.

Remark 14. *If* 32 *does not divide* $2r$*, then zeros complete the last 32-bit word into* ds_i*.*

So, a shift in ds is automatically a shift of at least 2^5 positions in **ds**. That implies using another method for the shift under 2^5 positions, *i. e.* for bits from b_4 to b_0, and slightly adapting Algorithm 10. Algorithm 11 displays the modifications.

Algorithm 11: Syndrome rotation for 32-words until bit b_5

Input: $\mathbf{s}$,$r_2 = \lceil \frac{2r}{32} \rceil$, $l = (b_{j-1}, \cdots, b_0)$
Output: ds

```
1 ds ← (s|s|0)
2 for k ← j − 1 to 5 do
3     for i ← 0 to r_2 do
4         ds_i ← (ds_i ∧ ¬mask) ∨ (ds_{i+2^{k−5}} ∧ mask)
```

Chen *et al.* wrote Algorithm 11 with a succession of **FOR** loops specific to each b_k to execute several shifts at once. For instance, for b_{13} it is possible to shift ds_{i+2^8}, ds_{i+2^8+1}, ds_{i+2^8+2}, and ds_{i+2^8+3} and after ds_{i+2^9}, ds_{i+2^9+1}, ds_{i+2^9+2}, and ds_{i+2^9+3} in the same loop without impacting the result. This method reduces the execution time. Algorithm 12 gives the idea of how it is done.

Algorithm 12: Shift execution for b_k

Input: ds,k, $mask$
Output: ds

for $i \leftarrow 0$**to** 2^{k-5} **by** nb_k **do**
  for $j \leftarrow 0$ **to** nb_{2k} **by** 2^{k-4} **do**
    $2nb_k$ shift
  additional shift depending of b_k
for $i \leftarrow 0$**to** nb_{3k} **by** 1 **do**
  $\mathsf{ds}_{i+2^{k-4}} \leftarrow (\mathsf{ds}_{i+2^{k-4}} \wedge \neg mask) \vee (\mathsf{ds}_{i+2^{k-4}+2^{k-5}} \wedge mask)$

Nonetheless, the bit treated determines the number of **FOR** loops executed and the number of shifts operated in one. Table 7.1 gives the parameter according to k.

k	nb_k	nb_{2k}	nb_{3k}	number of shift	number additional shift
13	4	512	3	8	0
12	4	512	2	8	0
11	4	384	2	8	4
10	4	384	2	8	4
9	4	384	2	8	4
8	4	384	2	8	4
7	4	384	2	8	4
6	2	388	0	4	0
5	1	386	0	2	1

Table 7.1: **FOR** loop parameter

To execute a full syndrome rotation for any $l \in L_i$, Algorithm 11 needs to be combined with Algorithm 13. The latter executes the shift for the last bits.

Algorithm 13: Syndrome rotation for the last 5 bits in 32-words

Input: ds,$r_2 = \lceil \frac{2r}{32} \rceil$, l
Output: ds

$tmp \leftarrow (l \& 31)$
$sh \leftarrow 32 - tmp$
for $i \leftarrow 0$ **to** $r_2 - 1$ **do**
  $\mathsf{ds}_i \leftarrow (\mathsf{ds}_i >> tmp) \vee ((\mathsf{ds}_{i+1} << sh) \wedge 0xFFFFFFFF)$

Therefore by successively executing Algorithm 11 and Algorithm 13, we obtain the full rotation for a Cortex-M4 in constant-time.

Chen *et al.* proposed two implementations of the syndrome rotation for the Cortex-M4. One is in C, and the other one is in Assembly language, referred to, respectively, as C code and Assembly code.

Both implementation are based on the structure explained in Algorithm 12 adapted to the language. However, the authors do not use Operation 7.1 to execute the shift but equivalent operation and instruction.

C code In C code, Operation 7.1 is written as follows

```
1 Rx0 ^= (Ry0^Rx0)&mask;
```

where $Rx0$ and $Ry0$ are previously set up with ds values.

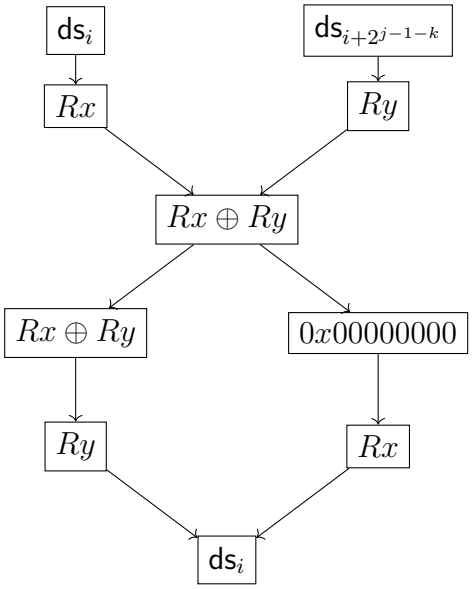


Figure 7.1: Shift representation for C code

Figure 7.1 described the process in detail with the impact of each step.

Assembly code The Assembly code replaces Operation 7.1 by an Assembly instruction SEL. This instruction works by byte movement. Let takes Rd, Rn, and Rm three registers. According to a flag GE[i], Rd[i+7:i], a byte of of the register Rd, will be either Rn[i+7:i] if the flag is set. Otherwise, it will be Rm[i+7:i]. If b_k is 1, therefore all the flags GE[i] from 0 to 3 are set. Otherwise, it is not.

7.2 Theoretical Attack

We constructed our attack with the information obtained by studying the implementations.

We learned that the implementations manipulate the coordinates l in their binary form. Except for the last 5 bits, they are manipulated independently. b_k gives it value to

mask, either *mask* is full of 1s or full 0. Both *mask* values impact power consumption in their own way. For a side-channel attacker, it is a gateway to get b_k. So, if by power analysis we obtain a maximum of b_k values, we restrict the number of possibilities for each l. Nevertheless, by attacking *mask*, an attacker cannot obtain the bit b_k for $k \in \{0, \ldots, 4\}$. The attacker still have $(2^5)^{142}$ possible combinations for $(\mathbf{h}_0, \mathbf{h}_1)$.

Therefore, with only up b_5 recover, the attacker cannot finish its attack. However, with the information gathered, a mathematical attack could allow the attacker to obtain the private key.

A few questions arise while setting up the attack.

1. How to extract the values of b_k for the trace?
2. Which mathematical attack?
3. How many b_k need to be recover?

For the power analysis attack part, we choose to use a clustering algorithm to determine the value of the bits.

7.2.1 Clustering

Clustering is an unsupervised machine-learning method that interprets the input data and finds natural groups called clusters. In our attack, the input data is the trace of the decapsulation process. More precisely, it is a set of subtraces corresponding to the syndrome rotation execution for all l. However, we cannot use a clustering algorithm to find all the bits at once. Indeed, this kind of algorithm does not give the value. It just separates the data into the number of groups asked. Therefore, we do not have any interest in executing a clustering for all the bits at once, as we already know that the majority of them are in different sets of 2^5 possible positions. However, if the data set is cut to only a bit b_k shift, then only 2 clusters are asked. One for each possible value of b_k, a.k.a 0 and 1. It means that if we are able to determine which group corresponds to 1 or 0, we obtain b_k for all l. There are many clustering algorithms, and the classic k-mean algorithm [51] is well suited for our case study. The k-means algorithm partitions a set of points (resp. vectors) into k groups with the objective of minimizing the distance between the points (resp. vectors) in each group and the different means of the groups. The algorithm is repeated until it converges or if a maximum number of iterations, fixed in advance, is reached.

A leakage on our subtraces means that at some points, there exists a distance between the power consumption of both value of b_k. k-mean should be able to detect those points and distribute the set into two clusters. To help this process, we normalize the data set. In other words, the subtraces are rescaled by their standard deviation to improve their

quality. After the execution of k-mean, each cluster receives a label, either 0 or 1. k-mean also returns a centroïd for each cluster, *i. e.* their mean. Nevertheless, the labels are not related to b_k value, *i. e.* b_k can be a 0 and its label be 1. To determine which group is for b_k equal to 1 (respectively b_k equal to 0), we use the centroids. We notice that at some points of comparison the centroid, of the subtraces for b_k at 1 have a higher power consumption than the centroid for b_k at 0. Thus, the bit value is decided according to the maximum of each centroid.

Algorithm 14: Bits b_{elt} Recovery

Input: *traces* for b_{elt}, *iter*
Output: Values of b_{elt}

$traces_r \leftarrow rescale(traces)$
$labels, centroid \leftarrow kmeans(traces_r, 2, iter)$
if $max(centroid[0]) > max(centroid[1])$ **then**
  $labels \leftarrow Permute(labels)$
return labels

Algorithm 14 presents the theoretical full-bit detection process with k-mean for b_k. $Permute(labels)$ changes the labels at 1 to 0 and those at 0 to 1. Executing Algorithm 14 allows us to get the bits from $j-5$ to 5 for each l. But, as said previously, we cannot recover the private key with only those information. However, it can be used as a hint to decrease the security of $BIKE$ against mathematical attacks.

7.2.2 Information-Set Decoding

BIKE parameters were designed to make known mathematical attacks inefficient. However, additional information can decrease the difficulty of applying a mathematical attack. If all the attacks use linear algebra, they are not identical in probability of success.

Solve by Linear Equation with $(\mathbf{h}_0)$

Mélissa Rossi *et al.* proposed an attack to recover the private key of QcBits, another scheme based on QC-MDPC. In their attack, they partially know where the position of the non-zero elements of $(\mathbf{h}_0)$ are. The idea is to exploit the relation $h^{-1} * h_0 == h_1$ to fully recover h_0 by creating a system of linear equations that is easy to solve. The right part of the system is generated by the multiplication of the quasi-cyclic matrix H^{-1} obtained from $\mathbf{h}^{-1}$ and the vector $\mathbf{h}_0^T$. The right side of the equations has r coefficients. However, the number of coefficients can be reduced as many positions of zeros in $\mathbf{h}_0$ are known. When $\mathbf{h}_{0,col}$ is 0 then automatically $H_{*,col} \otimes h_{0,col}$ gives 0. So, the column *col* of H^{-1} can be eliminated of H^{-1}. For k, the less significant bit in each index recovers from the side-channel attack, H^{-1} can be reduced to size $r \times (\frac{\omega}{2} \times 2^k)$. It is also possible to reduce the number of equations in the system. This idea is to select $\frac{\omega}{2} \times 2^k$ equations such that

the left part in the system only contains zeros. This step can be executed even without information on $\mathbf{h}_1$ as it is composed mostly of zeros. Table 7.2 gives the probability of generating a zero vector from $\mathbf{h}_1$ according to 2^k. The closer to zero, the better. The full private key recovery attack is finished by computing $h^{-1} * h_0$ to recover h_1.

ω \ 2^k	32	64	128
142	20.21	46.36	135.57
206	20.65	45.49	112.71
274	20.66	46.86	109

Table 7.2: -Log in base 2 of the probability of randomly selecting only 0 in h_1.

However, we also have information about $\mathbf{h}_1$ from our side-channel attack. We know that the non-zero elements are in a set of $\frac{\omega}{2} \times (2^k)$, so we are sure that for $r - \frac{\omega}{2} \times (2^k)$ element in $\mathbf{h}_1$ we have a zero. Therefore, the number of guess is reduced to $\frac{\omega}{2} \times (2^k - 1) - (r - \frac{\omega}{2} \times (2^k)$ in the set of $\frac{\omega}{2} \times (2^k)$ rows. Table 7.3 presents the probabilities with the additional information.

ω \ 2^k	32	64	128
142	0	0	104
206	0	0	19
274	0	0	0

Table 7.3: -Log in base 2 of the probability of randomly selecting only 0 in h_1 with additional information.

The probabilities in Table 7.3 are better than in Table 7.2, and the majority of the cases have a high probability of success. However, to know the total complexity of such an attack, the complexity of solving a system of linear equations needs to be added.

Thus, some cases become out of computational capacities, such as $\omega = 142$ and 2^k length sets. Furthermore, the attack fails if there are any errors, such as having one non-zero coordinates in the set of supposed zero coordinates in $\mathbf{h}_0$ or $\mathbf{h}_1$. Therefore, we look at other attacks, such as the Information-Set Decoding one.

Information-Set Decoding

Prange's ISD We select Prange's ISD for its simplicity for a first study of ISD algorithm efficiency. Prange's ISD, Subsection 2.2.4, requires as parameters the parity-check matrix H, a syndrome $\mathbf{s}$ as well as a Hamming weight ω to recover a vector $\mathbf{e}$ *s.t.* $H \cdot \mathbf{e}^t = \mathbf{s}$ with $wt(\mathbf{e}) = \omega$. In other words, solve the SD problem; Subsection 2.2.3.

Although we do not use the syndrome defined for BIKE in Section 4.2, $s = c_0 * h_0$, as we are not targeting the vector error $(\mathbf{e}_0, \mathbf{e}_1)$, instead, we use a property in linear coding

theory. By definition, if a vector $\mathbf{c}$ is a codeword of the code $\mathscr{C}$ with H as the parity-check matrix, then

$$H \cdot \mathbf{c}^T = 0_r.$$

Now $(\mathbf{h}_1|\mathbf{h}_0)$ is by construction a codeword of the QC-MDPC code used in BIKE. Therefore the ISD is executed with H,ω, and $\mathbf{s} = 0_r$ to recover the private key.

Of course, the security of the BIKE scheme against ISD attacks was studied. Nevertheless, as we bring new information on $(\mathbf{h}_1, \mathbf{h}_0)$, the security level changes. Indeed, thanks to side-channel, we are able to determine a set $\mathcal{W}_i$ of cardinality η_i of possible positions for each l. So $\mathcal{W} = [\mathcal{W}_0, \ldots, \mathcal{W}_f]$ with $0 < f \leq \omega$ is the set of all the possible columns with cardinality $\eta = \sum_f \eta_i$. Thus instead of picking r columns into the $2r$ columns of H, if $\eta < 2r$, ISD picks in $\mathcal{W}$.

Theoretically, we obtain up to b_5 with our side-channel attack for each l. So, η_i is set at 2^5 for any $\mathcal{W}_i$ to get the probability of success. Nonetheless, we also compute the probability of success while fewer bits are recovered. Table 7.4 shows the probability of success in $-log_2$ format for the worst case, *i. e.* the ω positions are all in different sets.

ω \ 2^k	32	64	128	256
142	0	0	79	222
206	0	0	19	226
274	0	0	0	213

Table 7.4: -Log in base 2 of the probability of success with Prange's ISD for different levels of BIKE and size of set $\mathcal{W}$

The cost of the Gaussian elimination needs to be added to the results in Table 7.4 to obtain the cost of the ISD execution. However, we can conclude that Prange's is, on average, more efficient than solving linear equations. Especially when $\omega = 142$ and $\eta_i = 128$, the level-1 parameters with only up to b_7 recover.

Prange's ISD with hints Nevertheless, reducing the set of columns is not the only type of information we obtain by partially recovering l. Indeed, we have the Hamming weight of any $\mathcal{W}_i$ *i. e.* the number of positions l in the set $\mathcal{W}_i$. A method to exploit this additional information was proposed by Horlemann *et al.* in [45]. Keeping Prange's ISD, the additional information, called a hint, changes the method to select the r columns in $\mathcal{W}$.

Prange's ISD with hint picks a fixed number x_i of columns in each set $\mathcal{W}$ in each execution *s.t.* $\sum_f x_i = r$. Then, we compute the probability of success as follows:

$$P_{hints} = \sum_f \binom{x_i}{t_i} \binom{\eta_i}{t_i}^{-1}.$$

At first x_i is allocated to t_i where t_i is the Hamming weight of $\mathcal{W}_i$. If $\sum_f x_i > r$, the sum is reduced in a way to maximize the value of P_{hints}. For $j \in 0, f$, we compute $\sum_{i \in f \setminus j} \binom{x_i}{t_i}\binom{\eta_i}{t_i}^{-1} + \binom{x_j-1}{t_j}\binom{\eta_j}{t_j}^{-1}$. The x_j that optimizes the sum is decreased by one. Repeat the step until $\sum_f x_i == r$.

Table 7.5 shows the probability of success in $-log_2$ of the Prange's ISD with hints. We compute the probability for the parameter in BIKE's specification for the three levels.

ω	32	64	128	256
142	0	0	79	221
206	0	0	19	225
274	0	0	0	212

Table 7.5: Prange's ISD probability of success in $-log_2$ with hints

Table 7.5 displays probabilities equivalent to Prange's algorithm without hints, see Table 7.4, for sets of length 128. When the sets are of size 256, the probabilities are slightly better. So, the hints improve the success of Prange's ISD.

Nevertheless, the probabilities computed for ISD do not take into account that t_i can be greater than 1, neither the real distribution of the non-zeros positions in $(\mathbf{h}_0, \mathbf{h}_1)$.

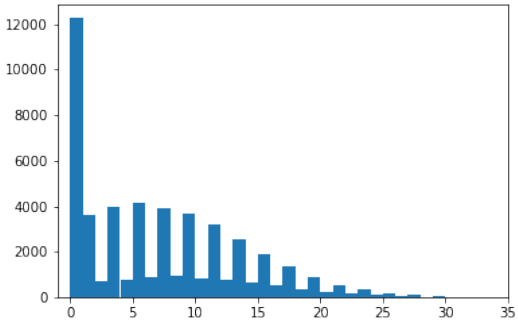
Attack on Private Keys Generated by BIKE Implementation

To have a more realistic estimation of the success of our attack, we compute the ISD probability for 50000 private keys generated by the Key Generation function of BIKE level-1 implementation.

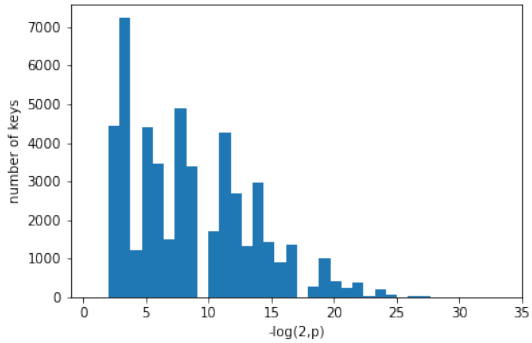
We first look at the case where we recover up to b_7 for each position. Figure 7.2 shows that the higher probability of success in -log base two is 35 for the keys generated by the implementation independently of the use of hints. In other words, there are at most 2^{35} combinations for practical keys while it is 2^{79} for theoretical keys; see Table 7.4 and Table 7.5. Nonetheless, Prange's ISD without hints is more efficient than with hints. Indeed, in Figure 7.2a shows more than 12000 keys have a probability of success at 0 into -log base two while none of the keys are in this case for the ISD with hints; see Figure 7.2b.

We also look at the case if we only recover up to b_8 for each position. In practice, it is clear that the maximum of combination possible for the Prange ISD algorithm is smaller with only 2^{120} combination while it is 2^{221} in Table 7.5. However, in this case, the hints are essentials to improve Prange ISD's success. Indeed we can see in Figure 7.3b that the majority of the keys are under 80 in -log base two while they are above 80 in Figure 7.3a.

From Figure 7.3, we can say that even with only the six most significant bits recover. The attack reduces the security of the practical keys by at least a bit. However, the real complexity of the ISD algorithm by adding the 2^{40} additional operations to solve the system after the selection. Herefore, all the private keys are either unbreakable or difficult to attack if only up to b_8 is recovered. Nonetheless, we can recover most of the keys by

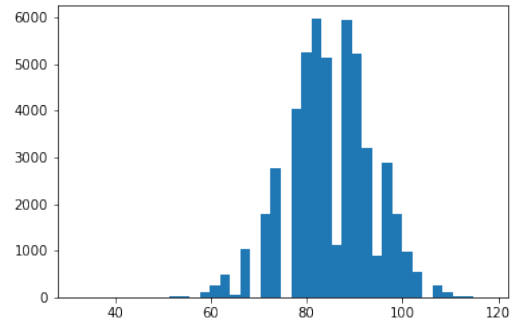


(a) Prange's ISD

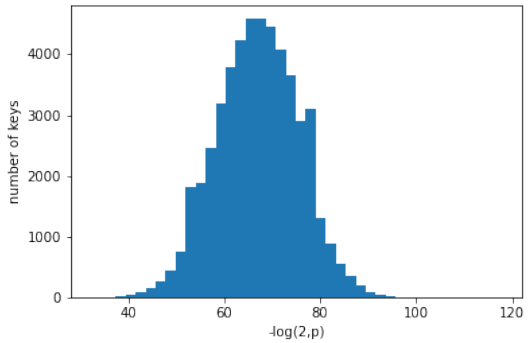


(b) Prange's ISD with hints

Figure 7.2: Number of private keys depending of the $-log_2$ Prange ISD probability of success when b_7 recover.



(a) Prange's ISD



(b) Prange's ISD with hints

Figure 7.3: Number of private keys depending of the $-log_2$ Prange ISD probability of success when b_8 recover.

obtaining another bit, *i. e.* b_7. Some of the practical private keys are straightforward to obtain.

7.3 C Implementation Experimentation

To test our attack in practice, we start with the C code.

7.3.1 Power Measurement Trace

Figure 7.4 shows the full decapsulation process power measurement. By the outlook of the trace, we have some precious information. Seven patterns, highlight in red in Figure 7.4, compose the trace. These patterns show a small block with a wide range of power consumption followed by two identical blocks with a lower range of power consumption, separated by a peak. The first block corresponds to the syndrome computation, while the next two blocks are the syndrome rotations for every $\{l \in \mathscr{L}_i, i \in \{0, 1\}\}$.

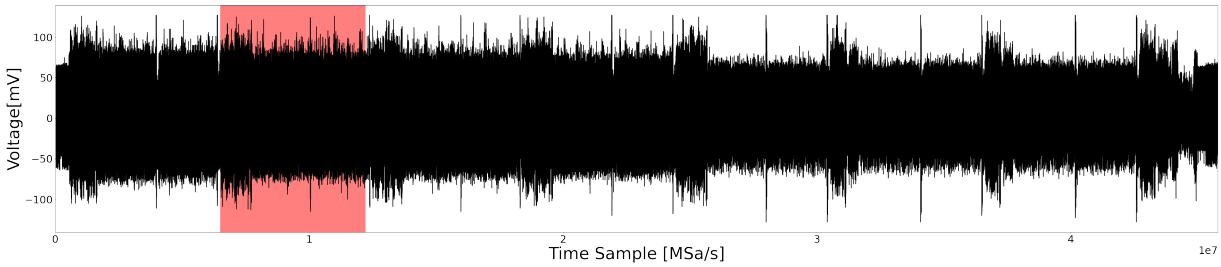


Figure 7.4: Full decapsulation process

We focus our interest on these two identical blocks. Figure 7.5 displays the first one. We distinguish 70 small white spaces on the trace, which are more or less visible. As we know that there are 71 positions in L_i, we conclude these spaces separate two executions of the syndrome rotation. Figure 7.5 is the power consumption of the Bit-Flipping Algorithm execution for h_0.

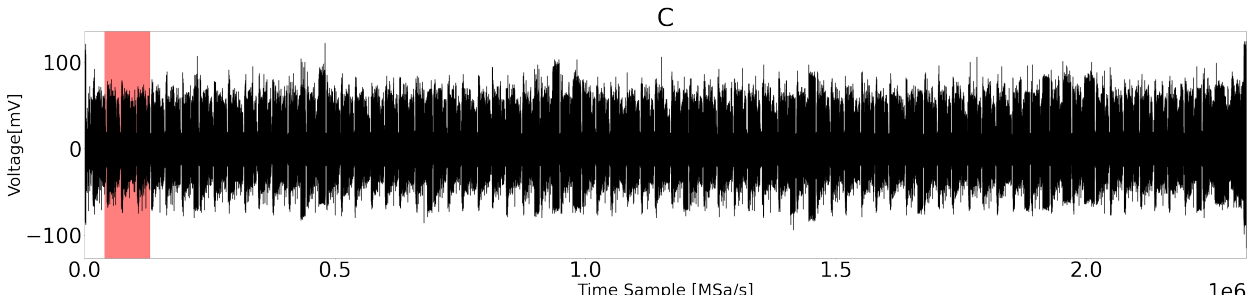


Figure 7.5: Syndrome rotations for h_0

Figure 7.6 shows three syndrome rotations randomly selected on Figure 7.5. The separations between the blocks of execution are clear. Using the code, we determine that a block comprises the syndrome rotation followed by a function to add the result to the parity-check counter. It also explains why the blocks have different lengths. Indeed, the latter function takes different parameters for each execution. Figure 7.7 detail the different section of the block.

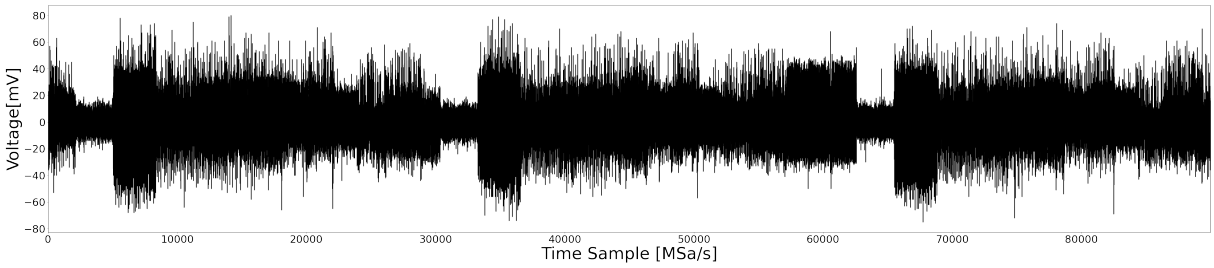


Figure 7.6: 3 syndrome rotations for $l \in L_0$

We focus only on the syndrome rotation and, more precisely, on the first part of it. To attack, we must distinguish between the different moments corresponding to a specific b_k.

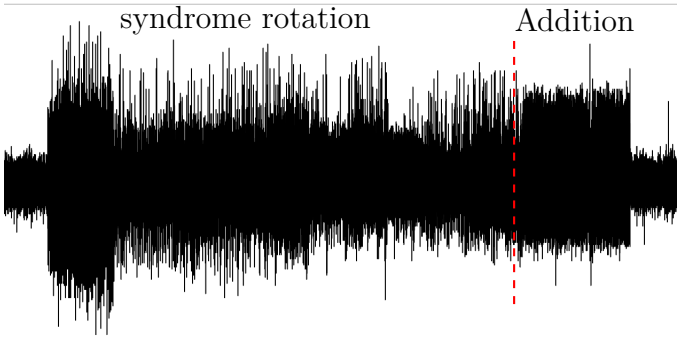


Figure 7.7: Syndrome rotation and add counter block

Figure 7.8 shows the mean of power consumption of an execution of the syndrome rotation function. On this mean, we see enough difference between the b_k to separate them for the clustering part of the attack.

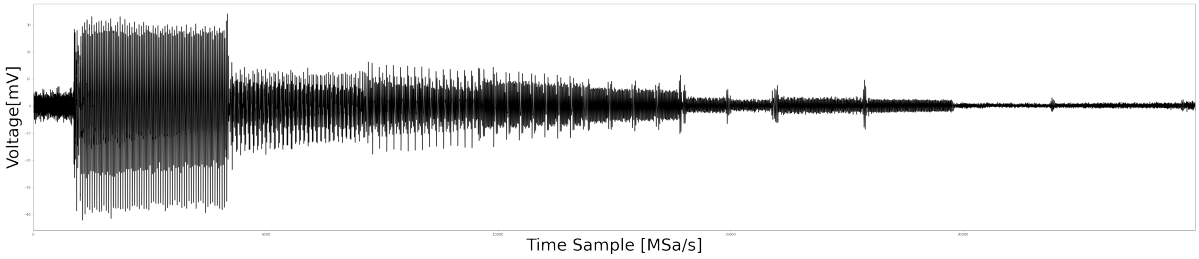


Figure 7.8: Mean syndrome rotation execution

Remark 15. *The power consumption of the trace in Figure 7.8 decreases over time due to some glitches, making the rotation unsynchronised.*

7.3.2 Exploitation of the Trace

We can only exploit the trace if a vulnerability related to our target information exists. Further, we need to determine whether the leakage is enough to be exploited by a clustering algorithm. This section describes our method to exploit the trace to get the bits necessary to reconstruct the indexes. We consider the different parameters related to the success of the K-mean algorithm and the possibility of clustering errors.

Although the patterns differ for each b_k, the operation to shift stays the same. Thus, for any $b_k, 5 \leq k \leq 13$, we execute the same process to extract the values of the bits. So, we will only explain it for the most significant bit, *i. e.* b_{13}.

Find the Leakage

To determine if there is a leakage for b_{13}, we compute the SOST t-test (Subsection 5.3.4) with 2840 patterns. It highlights moments where the subtraces differ depending on the

b_{13} value through points of interest (PoIs). The points of interest are the moment on the subtraces for which the SOST return a values higher than $(4.5)^2 = 20.25$. To compute the SOST, we know the values of b_{13} beforehand.

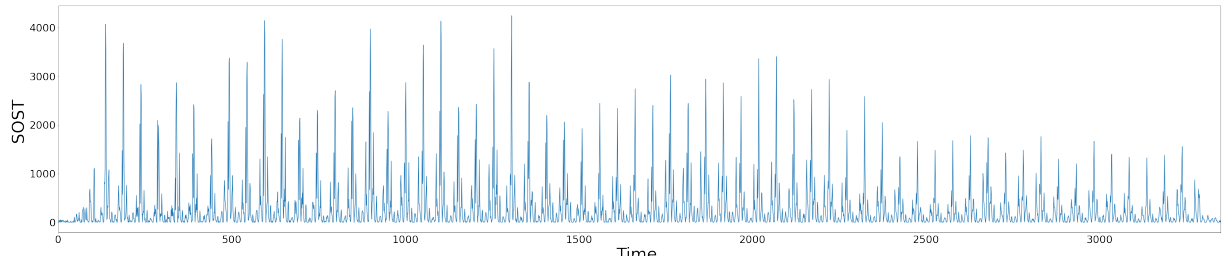


Figure 7.9: SOST for bit b_{13}

Figure 7.9 shows many PoIs with regular intervals between. The number of PoIs combined with their regularity shows the existence of a leakage in the targeted operation. The noise and some glitches during the execution explain the irregularity range. is explained

The high number of PoIs potentially increases the success of the k-mean algorithm. Indeed, the more there are differences, the easier for the clustering algorithm to detect them. On the contrary, the noise and the glitch can reduce it by creating differences unrelated to the target. Although there is a simple technique to overcome the desynchronization problem, we need to understand further the leakage to reduce the noise. Therefore, we search for an understanding of the leakage origin.

Find the leakage The leak must come from some part of the following operation:

$$Rx \ = Rx \oplus ((Rx \ \oplus \ Ry) \ \wedge \ mask) \tag{7.2}$$

where $Rx = \mathsf{ds}_i$ and $Ry = \mathsf{ds}_{i+2^k}$.

Three logical operators are successively used in Equation (7.2). Only one is directly related to the variable $mask$. Unfolding Equation (7.2), there is an exclusive OR (XOR) between and the result of $(Rx \ \oplus \ Ry) \ \wedge \ mask$. So, Rx is either XOR to 0 or $(Rx \ \oplus \ Ry)$ depending on the mask value. It is the first possible leakage origin. The second is the logical AND occurring right before the XOR, for which $mask$ is one of the two elements. Both operators possibly leak, and we cannot conclude on which one is at the origin. However, we are confident that $Rx \ \oplus \ Ry$ does not impact the leakage and creates noise.

Syndrome at zero Hopefully, a few iterations on the BIKE decapsulation process where the impact of $(Rx \ \oplus \ Ry)$ is minor. Indeed,the syndrome is at zero for at least the last two iterations. In other words, Rx is always equal to 0, as is Ry. Thus, the XOR does not generate noise due to the bit values difference in Rx and Ry. Figure 7.10 shows the t-test result with zero syndrome rotation subtraces.

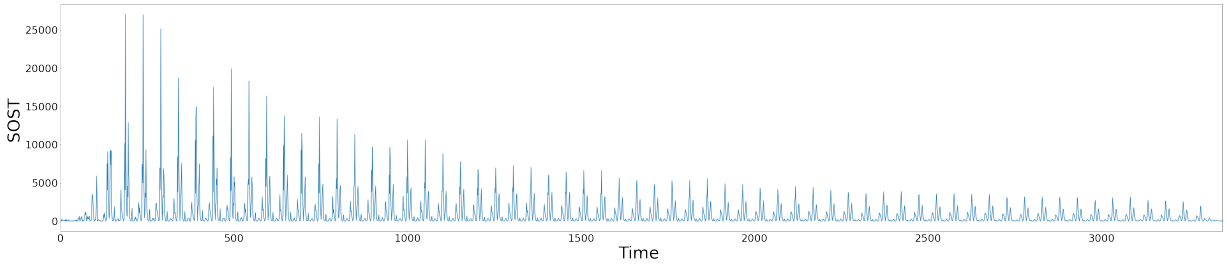
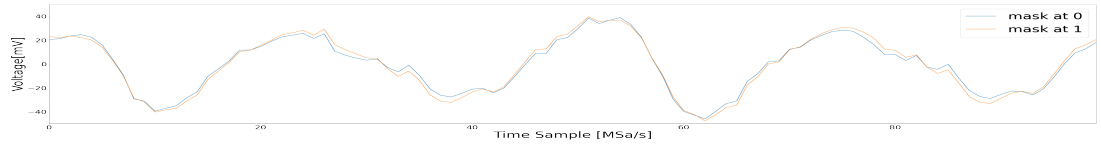
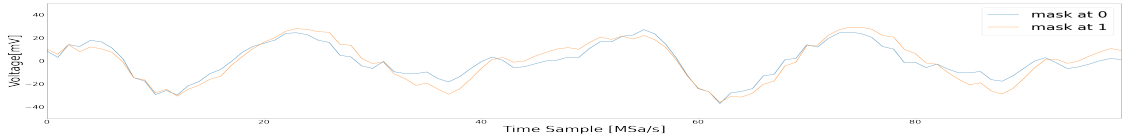


Figure 7.10: SOST for bit b_{13} with a zero syndrome

Figure 7.10 shows some PoIs with a range five times greater than those in Figure 7.9. The reduction of the noise generated by the XOR is also observed in Figure 7.11. If the points of difference exist for the non-zero syndrome rotations, Figure 7.11a, they are more significant for the zero syndrome rotations, Figure 7.11b.



(a) Non-zero syndrome traces



(b) Zeros syndrome traces

Figure 7.11: Extract from comparing the trace means for the 1 and 0 masks, for a non-zero syndrome first, and a zero syndrome second.

Logical AND with a zero syndrome The zero syndrome allows us to ascertain that the logical AND leak as, independently of $mask$ value, $(Rx \oplus Ry) \wedge mask$ always return 0. Thus, Rx is XORed to 0 when the syndrome is zero. There is no possibility of $mask$ value detection through this operator. Which leaves only the logical operator AND as the origin of the leakage.

Nonetheless, it does not mean the XOR does not leak when used with a non-zero syndrome.

For these reasons, we use the subtraces of the last two iterations in our attack.

7.3.3 K-Mean Clustering

The selected subtraces solve the noise problem but not the desynchronization one. A simple solution is to reduce the number of points compared to the k-mean algorithm. In

other words, cut to a low desynchronization moment on the subtraces. For example, at the beginning of each b_k rotation execution. However, the clustering is less precise if the number of points is too small. The right balance for our attack is to take twenty points around the highest PoI. For instance, the first one is higher than 25000 in Figure 7.10. Taking twenty points increases the k-mean precision as they include fewer PoIs than for more points.

K-mean accuracy Table 7.6 gives k-mean algorithm accuracy for each bit b_k. For bits 13 to 7, the accuracies are greater than 0.96. It means that k-mean is efficient in clustering the subtraces. However, the last two bits' accuracies, 6 and 5, drop to 0.5. In other words, k-mean is unable to detect differences in the subtraces between both values.

Bit	13	12	11	10	9	8	7	6	5
Clustering accuracy	0.974	0.987	0.972	0.985	0.962	0.983	0.985	0.504	0.536

Table 7.6: Average k-means accuracy

To remind the reader, the k-mean algorithm does not return the bit value, only a label for each subtraces. We obtain the information through the execution of Algorithm 14, Section 7.2.1.

7.3.4 Specificity of b_6 and b_5 bits

The number of shifts in the FOR loop structure in Algorithm 12 explains such differences between the accuracies in Table 7.6. Indeed, as shown in Table 7.1, b_6 and b_5 respectively require 4 and 2 shifts. In other words, the number of shifts set for bits from b_{13} to b_7 is divided by two for b_6 and by four for b_5. Fewer shifts imply a less important leakage, less visible on the power measurement.

We could stop recovering the positions l to the bit b_7. However, if we obtain b_6, the ISD probability of success is 1. Otherwise, it is 2^{-80} for the worst case. Additionally, b_6 SOST shows some PoIs. Thus, So we try to understand what happens during clustering.

First, on the 20 points selected around the highest PoI, only a few, 2 or 3, are relevant for the cluster. Second, the distance between the two groups at these points is smaller. So there are fewer differences on which to base the clustering. As a result, k-mean fails to create the wanted groups.

Reducing the data to a unique point, the highest PoI, instead of 20, gives an accuracy of 0.92 for b_6. This result is considerably better than the previous one. Although it is slightly lower than the accuracies obtained for the bits from b_{13} to b_7, it is enough to be exploited.

Remark 16. *There is no need to recover b_5 as it does not impact the attack probability of success.*

None of the accuracies are at 1, which implies that detection errors can still occur. Therefore, verifying the gathered information is necessary before any attempt of ISD solving.

7.3.5 Check Our Result

Despite a carefully chosen data set and high accuracies of the k-mean algorithm, clustering errors still appear. The term labeling errors refer to these errors. If there are unnoticed errors in the detection of 1's coordinates in $(\mathbf{h}_0, \mathbf{h}_1)$, the attack may fail. Indeed, ISD needs the correct information to avoid putting aside the sets with the right position.

We identify two reasons for the labeling errors:

- The power measurement in the subtrace is more similar to the other group than the one it belongs
- The k-mean base its clusterisation on noise

Both are related to how k-mean works and cannot be eliminated. The first case happened because the k-mean based its clustering on the Euclidean distance. So, logically when the subtrace is more similar to the centroid of the wrong group, it will be sent into this cluster. The second case results from that k-mean is an unsupervised learning algorithm, specifically the method to create the first centroids. The k-mean algorithm uses a group of randomly selected subtraces to compute those centroids. If a substantial majority of the subtraces have the same bit values, it creates a bias on clustering. Thus, the risk that the clustering is based on the noise is higher in this case than when the selected subtraces are more balanced between both values.

We chose not to correct the clustering for mislabeled subtraces due to smaller differences. Indeed, it is costly to detect these subtraces and remove them from the data set, whereas it only occurs rarely. It is an average of 2 subtraces on groups of 1988 subtraces for bits b_{13} to b_7. Due to smaller differences between the two clusters, b_6 has, on average, 16 subtraces mislabeled on 284. If they are detected, managing those errors is simple, see Subsection 7.3.6.

On the contrary, when the second case occurs, the errors cannot be managed for an ISD application. There are too many mislabelled (half of the subtraces), but it is easily detectable. For p execution of Algorihtm 14, the majority of the returns are identical, while the other possibilities are less frequent. The latter is wrong k-mean execution, while the former is what we aim for.

We choose $p = 50$ as a balance between enough returns to bring out a majority without having unnecessary executions.

We resolve the problem of fully wrong clustering, but we have one last step before successfully applying the ISD algorithm.

7.3.6 Errors Management

At this attack stage, the clustering process returns an average of 18 mislabels for 2272 given labels. In other words, less than 1 percent of the labels are incorrect. It is a small number but, it makes the success of ISD drop. However, knowing which l and bit b_k are concerned can prevent this. To do so, we exploit that we have two values returned by k-means for each b_k of l.

To recall, the syndrome rotations for each l in the last two iterations of the decapsulation process compose the set of subtraces given to the clustering algorithm. In other words, any bit in l is guessed twice with two different subtraces. Thus, the values are confirmed if both label returns are identical. Otherwise, we say that there is an error.

Remark 17. *While k-mean can return wrong labels for both subtraces for a specific b_k in l, the probability is negligible.*

We know where the errors occur so that we can use it to our advantage. We create the set of possible positions for each error with the two possible values for b_k the bit with an error. In other word, we will add the set of possible positions if b_k is one and the set of possible positions when b_k is 0. Therefore we have a set of 128 positions if l has an error in its binary decomposition. For every additional error, the set doubles.

Up to 50, sets of size 2^6 can be added without impacting the ISD probability of success. Indeed, the latter stays at 1 if the sum of the sets of possible positions is below r, here $r = 12323$. In the worst case, *i. e.* all the 142 positions l are in different set of size 2^6, the total is $142 \times 2^6 = 9088$ positions which leaves $r - (142 \times 2^6) = 3235 = 2^6 \times 50 + 35$ positions usually randomly selected. If all the errors are on different l, then 50 errors are manageable. When many errors occur in the same l, it reduces the capacity of error management.

In most cases, more errors are manageable as some of the l are in the same set of 64 possible positions. Furthermore, the errors can also create a set already used for another position l.

7.4 Assembly Implementation Experimentation

The Assembly language allows direct hardware manipulation or specific instructions utilization. The main interest of Assembly language is to speed up the program and use instruction specific to the device. However, while we know the result of such instruction, we need to learn how precisely it works.

In the syndrome rotation, the instruction SEL executes the shifts. Although the result of the syndrome rotations is the same as the ones in C implementation, its impact on power consumption is different. Consequently, the attack setup is impacted and needs to be adapted.

7.4.1 Power Consumption Traces

On its general aspect, the power consumption trace stays similar to the C traces, although shorter in length. Figure 7.12 shows the seven iterations with the same patterns combining the syndrome computation and the Bit-Flipping part with the syndrome rotations.

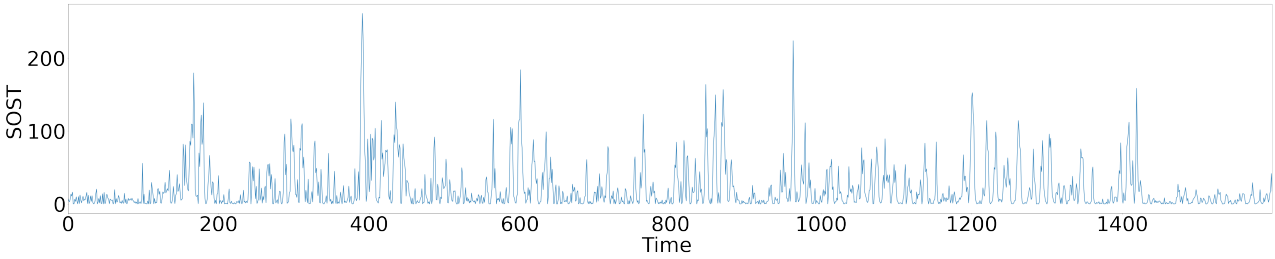
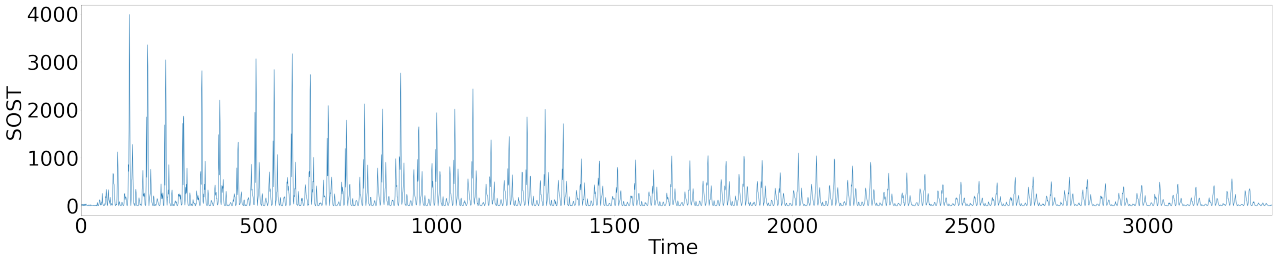


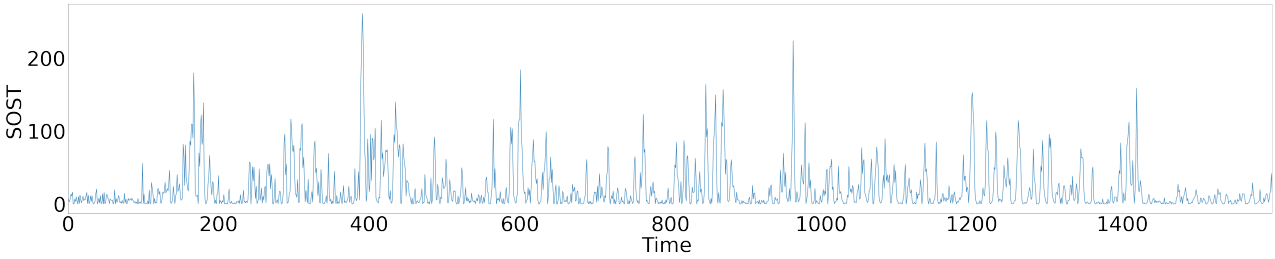
Figure 7.12: BIKE decapsulation in assembly

7.4.2 Syndrome Rotation

However, looking closer at the syndrome rotations, differences in respective bit b_k patterns are noticeable. Consequently, the SOST (Subsection 5.3.4) computed with the assembly traces is quite different. For instance, Figure 7.13 shows the bit b_{13} SOST for both implementations. Compared to Figure 7.13a, Figure 7.13b does not have regulars PoIs. Furthermore, the highest PoI is at least twenty times smaller.



(a) SOST for C implementation



(b) SOST for Assembly implementations

Figure 7.13: Comparison patterns for bit b_{13} C and assembly with non zero syndrome

Syndrome at zero We test if zero syndromes have an impact on the leakage. From the computed SOST, Figure 7.14, we conclude that such syndrome drastically reduces or eliminates the leakage.

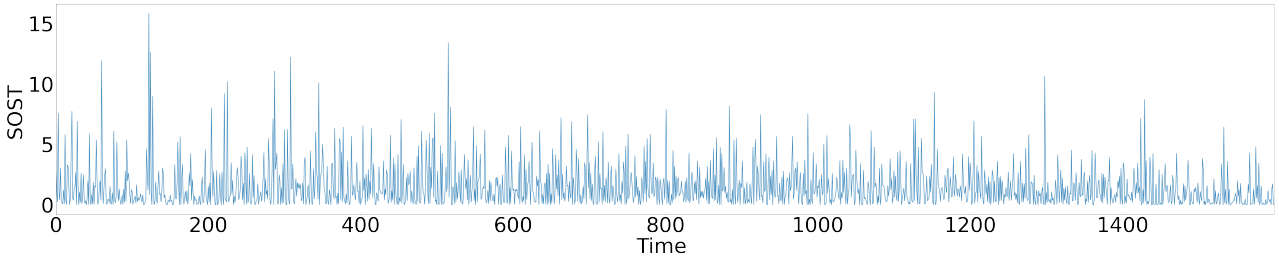


Figure 7.14: SOST for b_{13} with a zero syndrome.

Impact of the syndrome Following our tests with zero syndromes, we notice that the syndrome tremendously impacts on the SOSTs. Indeed, for two SOSTs computed with syndrome rotations subtraces for two different syndromes, the PoIs will not have the same positions, as in Figure 7.13b and Figure 7.15.

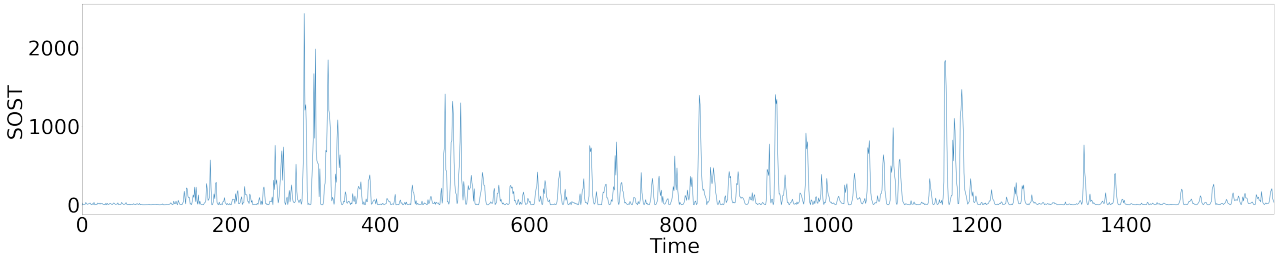


Figure 7.15: SOST for b_{13} with a different syndrome.

The leakage origin can explain this phenomenon.

Leakage Origin In the assembly code, the flipping of bits in the registers leak. The number of bits flipping determined the importance of leakage. The higher it goes, the more important it becomes. Thus the leakages are dependent of ds_i and $\mathsf{ds}_{i+2^{k-5}}$ and therefore on the syndrome.

Those differences in the leakages make it necessary to modify our process for k-mean execution, especially on the data set given as input.

7.4.3 K-Mean

Our modifications of clustering process are on the input. Precisely on the set of subtraces to cluster. We do not modify the number of k-mean execution neither the way we selected the labels from all the returns. Both are unaffected by the leak position change if the data set is well selected.

Select the Subtraces Set

Selecting the subtraces is a major problem due to the syndrome-dependent leakage. In the full decapsulation process, there are seven iterations with potentially a different syndrome for each. The last two iterations are executed with a zero syndrome, as previously

explained in Section 7.3. In most cases, the fifth iteration is also executed with a syndrome at zero. Nevertheless, there are unusable because of the lack of bit flipping.

Four iterations are left. By experimentation, we notice that the syndromes in the second and the third iteration are similar, if not identical. Moreover, they are close enough to the first iteration syndrome to have a majority of leakages in common. On the other hand, the syndrome in the fourth one is quite different from the other syndrome and does not share many leakages with them. Thus we only used the first three iterations. Nonetheless, even if the leakages are located at the same position, their power consumption has a different range. k-mean automatically fails to cluster if the $3 \times 142 = 426$ subtraces are given without preprocessing first.

Executing the k-mean algorithm with only 142 traces is not efficient due to desynchronization, which widely occurs in the power consumption measurement of the Assembly code. Neither combining the subtraces of the second and third iterations.

The solution was to combine the three subtraces corresponding to each l to a unique subtrace, *i. e.* their mean. To do so, we first check if all the subtraces are not too desynchronized with Pearson correlation computation (Subsection 6.4. Then their mean is computed. With the 142 means, k-mean returns the exact labels without errors.

We cannot reduce the subtraces to 20 points around the higher PoI as in the C attack because we do not know where it is. So, for any b_k treat, the related full execution is required. Like this, we can treat every trace. Even without knowing the leakage locations.

With the data set created from the subtraces as input, k-mean returns the bits b_k values. However, the syndrome still impacts the clustering process.

Syndrome impact on clustering process Indeed, during the syndrome rotation execution, ds is modified depending on b_k value. In other words, if b_k equals one, then the syndrome is rotated. If it is at 0, it is not. Thus the leakage locations are different for b_{k-1} depending on the previous bit, b_k. Therefore, after each bit b_k recover, the data set needs to be divided into two new data sets, one for $b_k = 1$ and the other for $b_k = 0$. This process is shown in Figure 7.16.

However, splitting the data sets in two at each b_k recover affects the number of bits in l we can obtain through k-mean execution.

Cluster Limits

The initial data set is composed of 142 subtraces. Thus if the subtraces are split into two almost equal new sets after each k-mean execution, only the bits up to b_7 can be retrieved. Indeed, the sets create after k-mean execution for b_7 contain two or three traces. So the sets are, in the majority, too small to have a successful clustering by k-mean. Indeed, when a set is composed of two subtraces, the possibility that both have the same bit value is high. Thus, k-mean either returns 0 and 1 randomly assigned to the traces or puts them

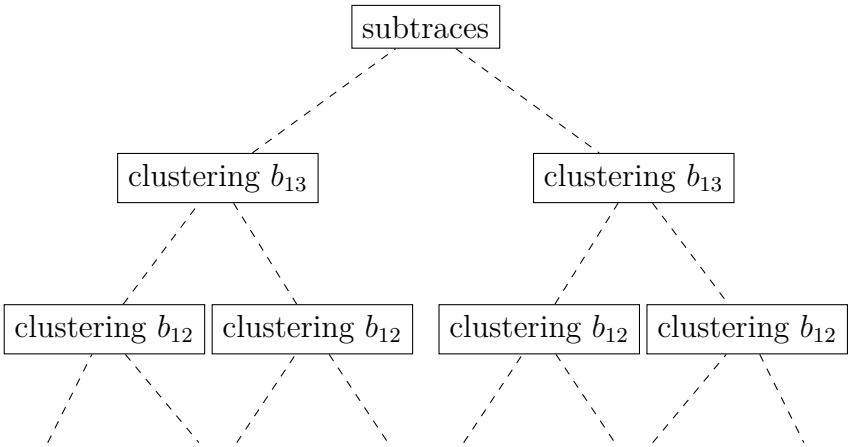


Figure 7.16: Syndrome rotation cosnequence

in the same cluster, but we are unable to determine the bit value. In both cases, we need to keep the two possible values.

Every data set is not split into two equal sets. Some sets are only composed of one subtraces. When it happens, the values of the following bits in the corresponding l cannot be find out. Although rare, it happens after k-mean execution for the bit b_{12}.

We notice another particularity of the small set, which impacts the clustering result. Some clusters have three to five traces for which the bit b_k has the same value. In this case, k-mean always creates two clusters, so some subtraces are mislabeled. However, mislabelling has a minor impact on our attack. It slightly increases the ISD complexity.

Our experiments show that some data sets with at least three subtraces still exist after b_7. Nevertheless, as in the C implementation attack, the leakages are less important for bit b_6 and difficult to exploit in our attack. Therefore, we stop the recovery of l at bit b_7.

At the end of the clustering step, we can recover for each position l the bits from b_{13} to b_9 without errors. The recovery process stops there for some positions, but we obtain b_8 and b_7 for most of them.

7.4.4 Key Recovery Through ISD

From Section 7.2.2, we have an idea of the ISD success according to the number of bits, especially with b_7 and b_8. However, as seen in the previous section, we are not able to obtain up to b_7 or b_8 for all the indexes as the sizes of clusters are too small. We distinguish three possibilities of stop for the bit recovery in l. Generally, at least four indexes with only up to b_9 recover, which implies blocks of length 2^9. The case of being blocked at bit b_{10} is rarer and does not occur for each private key. Nevertheless, if this is the case, the blocks have the size 2^{10}. The last possibility is when we are blocked in the recovery after the second bit,*i. e.* b_{12}. It is specific because it is only in the case $b_{13} = 1$ and $b_{12} = 1$. Indeed, in the level-I parameters $r = 12323$, therefore, there are only 35 values of indexes possible. However, it often has one or two traces, in this case, in the different private keys

generated.

By computing the total size of the different blocks for each index, it is clear that recovering the private key for the assembly implementation is more challenging than for the C one. Figure 7.17 gives the distribution of the private keys generated by the implementation according to the complexity of Prange ISD.

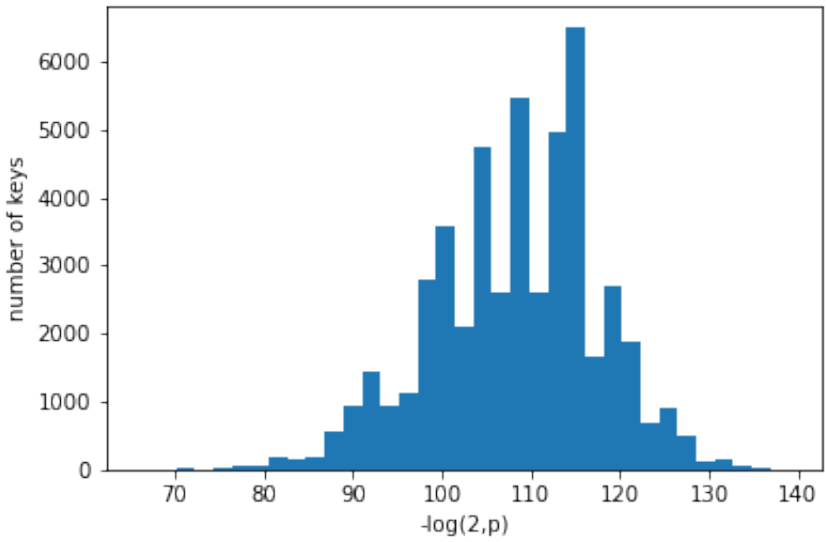


Figure 7.17: Distribution of keys according to the complexity of the Prange ISD.

From the distribution of the keys, Figure 7.17, we can say that the bare minimum of ISD complexity is 65, and most of the keys require a complexity between 100 and 120, which makes the private keys unattainable. Nevertheless, we are in a case where the Prange ISD with hints can reach a better complexity than the classic one.

Figure 7.18 represents the keys' distribution according to the Prange ISD's complexity with hints. The minimum complexity is around 40, and most keys are between 70 and 90. Thus, from the distribution, the keys with the smaller complexity are recoverable. However, this is not the case for the majority of them, even if the complexity is reduced between the Prange ISD without it and the one with hints.

The instruction SEL in assembly language reveals information on the private keys, but exploiting the leakage is difficult because of how the instruction works. Finally, only a tiny number of private keys can be fully recovered with the actual attack. Nonetheless, there are possibilities for improving the attack that has yet to be explored, such as the other Information-Set Decoding algorithms.

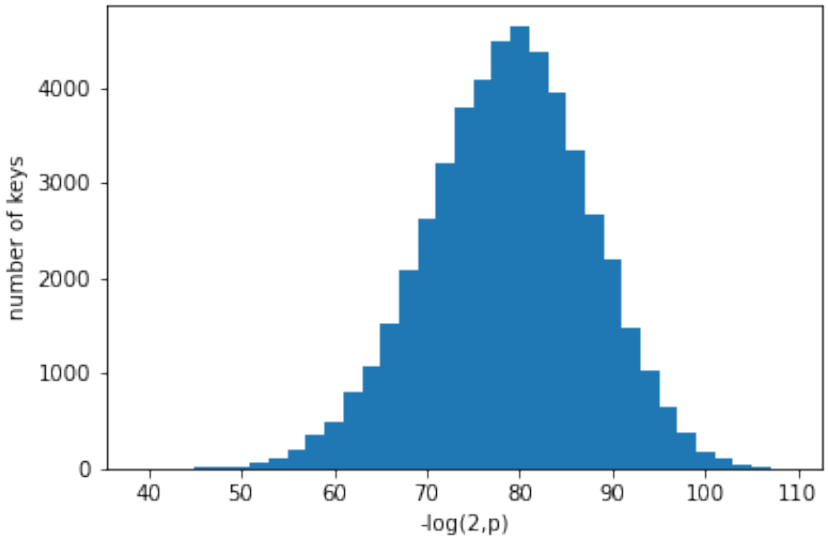


Figure 7.18: Distribution of keys according to the complexity of the Prange ISD with hints.

Resume and Conclusion

Resume

The important information on the attack of BIKE scheme are gathered here:

- Targeting the decapsulation, more precisely the Black-Grey Flip decoder implementation with the rotation of the syndrome
- The function execute the syndrome rotation using the binary decomposition of a integer
- The integers used are the indexes of the ones in the vector representation of the private key.
- An operation in the function allow us to obtain partially the indexes binary decomposition using the clustering algorithm algorithm
- The last bits are recover using the Prange's Information-Set Decoding
- There are two versions of the targeting operation one in C, the other in assembly.
- Both requires an unique trace of power consumption but their impact on the keys security are different.
- All the private keys generate are weak when the C implementation is executed
- Only a few keys are weak for the Assembly implementation

Through the attack of BIKE implementation for Cortex-M4, we show that the manipulations related of the private key are greatly sensitive to power consumption attack and so, even by begin protected against timing and cache attack. Nonetheless, by experimenting on two versions of the same implementation we show that the method of implementation has a significant impact on the security and the possibility of attack. Which lead us to propose countermeasure against our attacks on BIKE and ROLLO in the next Chapter.

Chapter 8

Countermeasures for ROLLO and BIKE

We proposed attacks on ROLLO and BIKE, respectively in Chapter 6 and Chapter 7, the logical question following these attacks is how to eliminate the leakage and make it impossible to attack. In Chapter 3, we introduced two types of countermeasures against power analysis attacks. We adapted them to protect ROLLO and BIKE. We work on two different protection methods in C for the two schemes, BIKE and ROLLO, one on the implementation protection and the other on the mathematical protection.

8.1 Secure Implementation

Protecting implementations involves removing the leakage to be unable to apply the attack. The masking countermeasure is ideal to achieve the protection requirement. Indeed, the masking hides the targeted information with its structure, and therefore, the leakage is unexploitable to attack as it does not refer to sensitive elements.

Remark 18. *As BIKE and ROLLO use $q = 2$, the boolean masking is the most indicated masking type.*

Let us start by protecting the ROLLO implementation.

8.1.1 ROLLO: Masking the Syndrome Matrix

The attack we proposed on ROLLO is based the recovery of the value contains in the variable *mask* during the execution of the following operation:

$$tmp = (mask * e0) \oplus (\neg mask * e1)$$

To reconstruct the syndrome matrix, we exploit the imbalance between the multiplications when the *mask* equals 0 and 1 . Specifically, we detect the case when there is a multiplication with only non-zeros factors and multiplications with a non-zero factor and a zero (see Section 6.2). So, if we want an efficient countermeasure, we need to be careful not to unbalance the operations. In other words, in the case of ROLLO, we aim at

performing a multiplication with only non-zeros operands independently of the value of $mask$. However, it is difficult to achieve.

So, we use the fact that the Gaussian Elimination is executed on a binary matrix to replace the multiplications with logical operations such as AND, OR, and XOR. Indeed, multiplication is more problematic to balance than an XOR while working on binary elements (another XOR cancels the first XOR with the same value).

Let us take the case of Algorithm 6 (Subsection 6.1.1) to explain. In this algorithm, the pivot is set up to 1 with the following operations:

$$mask = S_{row,col} \oplus S_{pivot,col}$$

$$tmp = (mask \otimes S_{row,*}) \oplus (\neg mask \otimes 0)$$

$$S_{pivot,*} = S_{pivot,*} \oplus tmp$$

In our countermeasure, we modified each operation and added a masking countermeasure with additional operations.

Remark 19. *The variable mask used in the initial Gaussian Elimination is not a masking countermeasure.*

We started by replacing the multiplication operation $mask \otimes S_{row,*}$ by a logical AND as follows:

$$mask \wedge S_{row,*}.$$

A modification of $mask$ is necessary to preserve the operation's result. If $mask$ is 1 then only the least significant bit of $S_{col,*}$ is preserved. It is due to the binary representation of $mask$, which is composed of 0s apart from the least significant bit. We, therefore, need to compute the value of $mask$ so that $mask$ is the same size as $S_{row,*}$ and only consists of 1 or 0 depending on the result of $S_{row,col} \oplus S_{pivot,col}$.

It is easy to implement in C thanks to the type of $mask$. In the constant-time implementation, $mask$ is of type *unint32_t*, which means that the value of $mask$ is written to 32-bit and that we can only write unsigned integers. In other words, $mask$ necessarily has a value between 0 and $2^{32} - 1$ and when we assign -1 to $mask$ then $mask$ actually takes the value $2^{32} - 1$. This property is used in the following operation:

$$mask = -(S_{row,col} \oplus S_{pivot,col})$$

To lighten the writing in the explanation, we leave the implementation details to the side and assume that $mask$ is the same size as $S_{row,*}$.

Our second step is eliminating null factors in operations (except for $mask$ or $\neg mask$). To do this, we split the pivot line into two parts, $pivot_1$ and $pivot_2$, such that $pivot_2 = S_{pivot,*} \oplus pivot_1$ with $pivot_1$ generated randomly.

We then perform the following operation to calculate tmp:

$$tmp = (mask \wedge (S_{row,*} \oplus pivot_2)) \vee (\neg mask \wedge pivot_1)$$

We finish by modifying the addition to the pivot row:

$$S_{pivot,*} = pivot_1 \oplus tmp$$

In the case, $row \leq pivot$ then the variable $dummy$ replaces $S_{pivot,*}$.

Using $pivot_1$ and $pivot_2$ instead of the pivot row masks the value of $pivot$ in operations and is a masking countermeasure.

For the elimination of 1 in Algorithm 7 (Subsection 6.1.1), the adaptation is similar, except that $S_{row,*}$ and the pivot row are swapped.

The countermeasure we proposed well balances the operations such that computing tmp is made using two XOR with at least a non-zero element, and one always returns zero and the other a non-zero value independently of $mask$. Furthermore, the pivot row or the dummy variable update is always made with a non-zero-only XOR.

However, such protection is costly regarding computation time with three XOR, two AND, and one OR, in addition to creating a random element. It leads to an increase in execution times of the Gaussian Elimination.

With the parameters of level-I of ROLLO, $r = 83$ and $m = 67$, the decapsulation with multiple calls to the Gaussian elimination in constant-time initial takes 0.032703 second in average to be executed. Now, with the countermeasure, it takes 0.321089 second, *i. e.* ten times slower.

Remark 20. *We talk about average execution time in the case of ROLLO because some functions used in the constant-time implementation are not in constant-time but isochronous. It means that the difference in time execution is unrelated to the sensitive elements.*

8.1.2 BIKE

The implementation of BIKE is more complex to protect than the ROLLO one.

In BIKE, the origin of the leakage exploited in Chapter 7 is the use of the variable $mask$ in the following operation:

$$Rx = Rx \oplus (mask \wedge (Ry \oplus Rx))$$

Indeed, thanks to the power consumption, we determine if the 32 bits of $mask$ are 0s or 1s. The operations are then unbalanced, especially when Ry and Rx equal 0 (see Section 7.3).

The following operation thwarts the proposed attack:

$$Rx = (Rx \wedge \neg mask) \oplus (Ry \wedge mask) \tag{8.1}$$

Indeed, the fact that there are two logical ANDs with $mask$ and its negation, $\neg mask$, *i. e.* elements with 32 bits at 0 and 32 bits at 1, we should not be able to determine the initial value of $mask$.

However, a previous work proposed by Sim *et al.* reveals a leakage on the variable $\neg mask$ on a similar operation [74]. In the paper, the authors explained that the load of $\neg mask$ can be exploited with power analysis to determine its value. So, with this countermeasure we have a potential vulnerability.

Yet, this remaining leakage is linked to the implementation and we find a way to compute $\neg mask$ without having this vulnerability. Concretely, $mask = -b_i$ and $\neg mask = -(1 - b_i)$. This way, the implementation should not show the same vulnerability than the one exploited by Sim *et al.* . Indeed, by not using the binary one's complement operator and using a specific variable, we remove in our implementation of the countermeasure the leakage exploited by Sim *et al.* .

Nonetheless, with this countermeasure one potential vulnerability remains.

The value of Rx is updated to Rx or Ry according to the value of $mask$ (which is the expected behavior). Since Rx is represented by a unique register, it is only modified if it is updated with the value of Ry. If an attacker can determine whether the register has been modified or not, then, he can trace back the private key.

In practice, we noticed that in the assembly implementation generated by the compiler, the value of Rx is directly updated with the result of the XOR between itself and the precomputed value $mask \wedge (Ry \oplus Rx)$.

We thus need to go further to protect the implementation by adding a masking countermeasure.

Let us start by decomposing Rx into two elements Rx_1 and Rx_2 such that $Rx_2 = Rx \oplus Rx_1$ with Rx_1 randomly generated. With a similar process, we obtain Ry_1 and Ry_2 such that $Ry_2 = Ry \oplus Ry_1$.

We inject the shares into Equation (8.1) what gives

$$((Rx_1 \oplus Rx_2) \wedge \neg mask) \oplus ((Ry_1 \oplus Ry_2) \wedge mask)$$

Then, we tweak a bit this expression so that shares are not XORed together.

$$Rx = (Rx_1 \oplus Ry_1) \oplus ((Ry_1 \oplus Rx_2) \wedge \neg mask)) \oplus (((Rx_1 \oplus Ry_2) \wedge mask) \tag{8.2}$$

This operation is far more complex than the original one, but we obtain the correct result and it should protect the implementation against the aforementioned weakness.

Indeed, now, the register which will store the result (namely Rx or Ry) initially contains $Rx_1 \oplus Ry_1$. It will thus be modified whatever the value of $mask$ is.

Let us prove the correctness of the countermeasure.

Correctness of the Countermeasure

Let us take Rx, Ry, Rx_1, Rx_2, Ry_1, and Ry_2 such that $Rx_2 = Rx \oplus Rx_1$, $Ry_2 = Ry \oplus Ry_1$. We have two cases for $mask$ fills with 0s or with 1s. Let start with the case where mask is filled with 0s.

Case with $mask$ filled with 0.

When $mask$ is filled with 0s no rotation of the syndrome is executed. Then, Equation (8.2) is simplified as follows:

$$\begin{aligned} Rx &= Rx \oplus Rx_2 \oplus Ry_1 \\ &= Rx \oplus (Rx_2 \oplus Ry_1) \\ &= (Rx_1 \oplus Ry_1) \oplus (Rx_2 \oplus Ry_1) \\ &= Rx_1 \oplus Rx_2 \end{aligned}$$

where $Rx_1 \oplus Rx_2$ is the initial Rx. So, no rotation is executed, and it is the correct result.

Case $mask$ filled with 0.

Now, let $mask$ be filled with 1, so the syndrome is rotated. Equation (8.2) is equivalent to the following:

$$\begin{aligned} Rx &= Rx \oplus Rx_1 \oplus Ry_2 \\ &= Rx \oplus (Rx_1 \oplus Ry_2) \\ &= (Rx_1 \oplus Ry_1) \oplus (Rx_1 \oplus Ry_2) \\ &= Ry_1 \oplus Ry_2 \end{aligned}$$

where $Ry_1 \oplus Ry_2$ is Ry. Therefore, the initial Rx becomes Ry. In other words, a rotation is executed. We obtain the correct result.

Side-channel Security Assessment

Now, that the correctness has be proven, we want appraise the security gain provide by the countermeasure.

We check if any possibility of leakage is easy to find thanks to methods such as t-tests. To compare with the results between without countermeasure and with countermeasure,

we focus on the t-test SOST (Subsection 5.3.3) and the vulnerability of bit b_{13}, see Subsection 7.3.2. If the countermeasure is effective then we should, see a reduction of the values between the SOST with and without the countermeasure.Figure 8.1 is the result of SOST for b_{13} with the countermeasure applied. Let us recall that the values correspond to the square of the classical SOST thus the baseline for leakage detection is $(4.5)^2 = 20.25$.

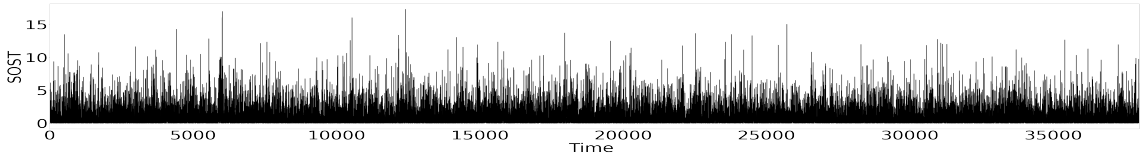


Figure 8.1: SOST for b_{13} with the countermeasure

Comparing Figure 8.1 with the SOSTs computed for C implementation, Figure 7.10 and Figure 7.13b, we see that the countermeasure protects the implementation, and if there is a leakage, it is less easy to exploit. Indeed, before the countermeasure, we obtained maximum values of SOST 4000 with the non-zero syndrome and 25000 with a zero syndrome, while with the countermeasure, we obtained a little bit less than 17. As the maximum value of the SOST with the countermeasure is close to 17 but under 20.5, we conclude that there is no significant leakage in our countermeasure.

The drawback with our countermeasure is the execution time. Indeed, without the countermeasure, the execution of the decapsulation is in 0.006318 second, with the countermeasure it grows up to 0.108021. In other words, the function with the countermeasure is 17 times slower than without it. It is explained by the way we implemented the countermeasure. In addition to increasing the number of necessary logical operations, we choose to consistently generate new randoms for each rotation, which is time-consuming as we need two arrays of the same length as the syndrome of random elements. It is possible to reduce the execution time by fewer calls to random generators. However, the reduction must be carefully considered, as using the same random too many times can lead to an attack.

Another way to protect without significantly impacting the implementation is to make the attack mathematically unfeasible.

8.2 Mathematical Countermeasures

Another way to protect the implementation is to make the attack mathematically impossible. We are not looking to remove the leakage but to make it unexploitable by the attacker as it would require too much computation afterward.

8.2.1 ROLLO

In the case of ROLLO, we propose two countermeasures based on the shuffling to prevent our attack.

Shuffling the Rows

Our first proposition is to shuffle the order of the rows of the syndrome matrix every time Algorithm 6 and Algorithm 7 are executed. This means that instead of following a sequential order where the operations on first row are executed first and then for the second row and so on, the operations will be done for the fifth row first, followed by the $r-1$-th row, and so on.

The randomization of the treatment order is a permutation of numbers from 0 to $r-1$. It can be executed by any algorithm generating a random permutation of a finite set. For instance, one can use the Fisher-Yates method, described in Algorithm 15.

Algorithm 15: Fisher-Yates

Input: L a list of n element
Output: The shuffled list L

```
1 for i ← r − 1 to 0 do
2 |   j ← random() mod i
3 |   L_i ↔ L_j
```

In the case of ROLLO implementation, we execute a permutation of the list L right before each algorithm execution. Furthermore, we choose the pivot row randomly, *i. e.* randomly select a row and permute it with the initial pivot row.

The shuffling of the rows does not remove the leakage, so an attacker can still obtain the values of $mask$. However, the attacker is unable to reconstruct the initial syndrome matrix because he cannot link the recovered values to their initial position in matrix syndrome. Indeed, the permutation of the pivot row combined with the permutation of the rows treatment before each for loop ensures a possibility of $r!$ combinations for the column. Moreover, the same process is executed for each column for a total of $(r!)^m$ possibilities for the syndrome matrix. Let us use the parameters for ROLLO-I-128 (the smaller), $r = 83$ and $m = 67$, to give an idea of the difficulty of the attack. We have $(83)^{67} \approx (3.94 \times 10^{124})^{67}$, so approximately 2^{27231} possibilities.

Shuffling the Syndrome

Shuffling the treatment of the row order before each algorithm is an efficient countermeasure. However, it requires $2m$ calls to a permutation list to permute L. We look for a countermeasure requiring fewer calls to the permutation algorithms (which can be costly in execution time).

The idea is simple: instead of shuffling the rows for each column, we shuffle the rows of the syndrome matrix once before the Gaussian Elimination. Obviously, an attacker can recover the elements composing the syndrome, but he cannot go further in the attack, and there still are $(n!)$ possible combinations for the syndrome. Without the syndrome s, the attacker cannot recover h_0 from c. The result return by $c^{-1} * s' \mod P$ is wrong except when $s' = s$.

With the parameters of ROLLO-I-128, we obtain approximately 2^{413} possibilities for the syndrome.

Even though shuffling only the syndrome before the Gaussian Elimination is weaker than shuffling the order of execution for all the calls, it is still strong enough to stop the key recovery attack.

8.2.2 BIKE

Similarly to the countermeasure for ROLLO implementation, we imagined a countermeasure based on shuffling to prevent the attacker from tracing back the private key without eliminating the vulnerability.

Shuffling the Binary Representation

In the BIKE implementation, the proposed attack targets a vulnerability in a function that rotates the syndrome according to an index l. The fact that the rotation function is executed independently for each index l in $\mathscr{L}$ eliminates two possibilities of shuffling countermeasures. First, shuffling the order in which the indexes in $\mathscr{L}$ are processed. We do not need to get them in order in the attack as long as we can recover them. For the second, we need to briefly remember that the rotation function precisely uses the binary representation of l and executes some operation according to each bit. Each bit is independent. Thus, one possibility is to shuffle the order of the indexes between the operations for each bit. For instance, we execute the rotation of the syndrome with the most significant bit for all the indexes, then shuffle the order of the indexes before doing the syndrome rotation with the next bit. However, this countermeasure conflicts with the fact that the rotation function is executed independently for each index. Consequently, the two countermeasures are only applicable if we modify the independent side of the execution of the syndrome rotation function, which is not our aim by doing shuffling.

Nonetheless, there is a third possibility of shuffling countermeasures impacting each index individually. The idea is to shuffle the order of processing of the bits in the binary index l representation. For example, with BIKE Level-1 parameters, l is represented by fourteen bits from b_{13} to b_0. The ninth most significant bits are processed independently, so the operations for bit b_6 can proceed before those for bit b_{12}. The execution order does not impact the final result.

The shuffling of the execution order of the bits increases the number of possible values for l. Indeed, if we know how many 1s and 0s there are in the index (except for the five less significant bits), we do not know their positions, so we have multiple possibilities. The exact number of possibilities depends on the Hamming weight of l. For instance, if on the 9 bits we have a Hamming weight of 1, then we have $\binom{9}{1} = 9$ combinations to test, while if the Hamming weight is 4, we have $\binom{9}{4} = 126$ combinations. As we executed the shuffling for each l in $\mathscr{L}$, we have the following number of possible combinations :

$$\prod_{l \in \mathscr{L}} \binom{j-5}{w_H(l_{j-1,\ldots,5})},$$

where j is the number of bits in the binary representation of l and $l_{j-1,\ldots,5}$ the $j-5$ most significant bits.

For example, always with the parameters of BIKE Level-I, if all the indexes have a Hamming weight of 1, therefore, there would be $\left(\binom{9}{1}\right)^{142} = (9)^{142} \simeq 2^{450}$ possibilities for the private key to test with the Information-Set Decoding algorithm. As this case is unlikely, there are generally more possibilities. It is, therefore, difficult to determine the private key with the information obtained.

However, this countermeasure is based on the idea that bits cannot be distinguished. However, we can distinguish the different bits in the BIKE execution measurements. Each bit has a different power consumption pattern (see Figure 7.8 in Subsection 7.3.1). To make this countermeasure applicable, we must modify the function and operations to make the bits indistinguishable.

Since shuffling is particularly complex to implement in the case of this function, we thought of another solution.

Light masking

The masking countermeasure presented in Subsection 8.1.2 allows us to eliminate the vulnerability in the implementation. However, it requires many calls to a random number generator to ensure implementation protection. In the following masking countermeasure, we use lighter masking than in the other masking countermeasures, as it requires only one random number for each index. We are allowed to do this because we are not looking to eliminate the leakage but to block this attack even if we obtain information from it. The idea is to generate a random number $rand$ between 0 and r-1 and execute the rotation function with $l + rand \mod r$ instead of the index l. In other words, we are masking the value of the index, and thus, l can not be determined even if the value of $l + rand \mod r$ is obtained through side-channel attacks. The problematic aspect of this countermeasure is to obtain the correct syndrome rotation, i.e., a rotation of l positions, without revealing the value of $rand$. For example, one solution to get the correct rotated syndrome is to

execute another rotation of $r - rand \mod r$ position on the syndrome right after the first rotation of $l + rand \mod r$. The problem is that if we have the values of $l + rand \mod r$ and $r - rand \mod r$, we know $rand$; thus, we also know l.

A simple solution to this problem comes from the implementation. In the BIKE implementation, the rotated syndrome is only needed in one function that adds the values in the rotated syndrome ($\mathbf{s}$) to an array ($\mathbf{count}$). For instance, for the i-th element of the syndrome :

$$\mathbf{count}_i = \mathbf{count}_i + \mathbf{s}_i,$$

So, we can cancel the rotations of $rand$ positions of the syndrome thanks to this function without modifying the rotated syndrome by doing as follows:

$$\mathbf{count}_i = \mathbf{count}_i + \mathbf{s}_{i-rand \mod r}$$

Concretely, the function to add the rotated syndrome to the array **count** is more complex to ensure constant time, but it is easy to replace the call to the i-th element of the syndrome by $(i - rand)$-th one.

Nevertheless, an implementation constraint impacts $rand$, so on the security offered by the countermeasure. In BIKE implementation, all the variables are represented by arrays of 32-bit words, including the syndrome. Thus, any allocations from one array to another are made by 32-th bits. Moreover, the manipulations under the size of 32-bit, such as a shift of 4 bits, require specific operations. Consequently, reversing the additional rotation generated by $rand \mod 32$ is more complex. By restraining the random number $rand$ to values between 0 and $\lceil \frac{r-1}{32} - 1 \rceil$, and executing the syndrome rotation with the index $l + rand * 32 \mod \lceil \frac{r-1}{32} \rceil$, it allows the initial rotation of values below 32 to be maintained, so that we do not have to manage it later.

The masking countermeasure increases the difficulty of recovering the private key. Indeed, we have $(\lceil \frac{r-1}{32} \rceil) - 1$ possibilities for each index in the private key. In total, we have the following number of combinations:

$$\prod_{l \in \mathscr{L}} \left(\left\lceil \frac{r-1}{32} \right\rceil - 1 \right).$$

Each combination must be verified with the Information-Set Decoding algorithm, which incurs a computational cost. To illustrate, using level-1 parameters for a BIKE, the countermeasure generates $\left(\lceil \frac{12322}{32} \rceil - 1 \right)^{142} = (385)^{142} \simeq 2^{1220}$ potential private keys, making it computationally challenging.

In this chapter, we proposed different countermeasures for ROLLO and BIKE. Two of them are masking based such that the leakages are eliminated from the implementations. However, we showed that the execution time of the scheme increased a lot in

consequence. We also proposed three other countermeasures based on the shuffling. Such countermeasures do not eliminated the leakages, however, recovering the private key from the information is beyond current computation capacity. The study of countermeasures is a wide-ranging subject and needs to be explored in greater depth in order to propose solutions tailored to the problems and constraints involved.

Conclusion

This thesis has taken place during the post-quantum cryptography standardization organized by NIST. The process is drawing close, with schemes selected in July 2022 to be standardized as Crystal-Kiber. The study of some schemes continues with the perspective of a potential standardization, such as those based on error-correcting codes. The standardization process has provided us with a database for studying the vulnerability of cryptosystem implementations to side-channel attacks. We were particularly interested in optimized implementation in constant time. These offer protection against timing or cache attacks but do not guarantee protection against consumption attacks, as we have shown in this thesis.

Achievement

We have highlighted vulnerabilities in two distinct schemes: BIKE and ROLLO.

ROLLO

In the ROLLO implementation in constant time, we showed that one multiplication was at the origin of the reconstruction of the syndrome. It then enabled us to trace back to the private key. In this case, it wasn't multiplication alone that caused the vulnerability but how it was used. Thanks to the power analysis, we can detect whether the multiplication is carried out with only one of the factors at zero or none of the factors are at zero. Using this difference, we can reconstruct the syndrome as a binary matrix. This vulnerability is all the more important as the difference in power consumption is visible to the naked eye and requires only a single trace to find the information needed to attack the private key.

BIKE

The Cortex-M4 implementation of BIKE has a vulnerability that requires the K-mean algorithm to be combined with the Information-Set Decoding algorithm to obtain the private key. The K-mean algorithm allows us to extract data from a single consumption trace of the execution of the scheme, enabling us to reduce the number of possibilities for the private key. We obtain the private key only after executing the Information Set Decoding algorithm with reduced possibilities. Our work has highlighted the differences in vulnerability between programming languages for the same attack. Indeed, the vulnerability detected in the C version allows us to access all the private keys generated by the

BIKE implementation. At the same time, only a tiny fraction of these private keys can be recovered with the assembler implementation.

Contermeasures

We have proposed different types of countermeasures adapted to the BIKE and ROLLO schemes. These countermeasures keep the constant time while preventing the attacks we have proposed. The first countermeasure eliminates the vulnerability in the implementation. The counterpart is that the execution time of the scheme is increased. The other idea we have described does not eliminate the vulnerability but prevents the attack by making it impossible to reconstruct the sensitive element from the data obtained by the current power analysis attack.

Perspectives

In this thesis, we have highlighted vulnerabilities in constant-time implementations of ROLLO and BIKE.

The natural next step is to continue the vulnerability analysis on these schemes. In our work, we have detected other potential vulnerabilities that we had previously put to one side. Analyzing them would enable us to move towards greater protection of the implementations thanks to countermeasures. The other part of the study is to take a closer look at the countermeasures we have proposed because we have proposed countermeasures adapted to the vulnerabilities we have highlighted. We have not tested them against other auxiliary channel attacks; they could show vulnerabilities we did not suspect.

The difference between metrics in the case of a side-channel attack is another field worth exploring. We have explored two metrics in this thesis: the rank metric with ROLLO and the Hamming metric with BIKE. However, there are other metrics that are beginning to be developed for cryptography, such as Lee metric.

Lee metric is also represented in the recent standardization process for post-quantum signature algorithms organized by the NIST with the FuLeeca [69] scheme. This brings us to a fourth avenue for pursuing this work, namely, the security of signature scheme implementations.

BIBLIOGRAPHY

[1] Carlos Aguilar Melchor, Nicolas Aragon, Paulo Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Güneysu, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor, *BIKE*, Round 3 Submission to the NIST Post-Quantum Cryptography Call, v. 4.2, version 4.2, Sept. 2021, URL: `https://bikesuite.org`.

[2] Carlos Aguilar-Melchor, Nicolas Aragon, Magali Bardet, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Alain Couvreur, Jean-Christophe Deneuville, Philipe Gaborit, Adrien Hauteville, and Gilles Zémor, *ROLLO-Rank-Ouroboros, LAKE & LOCKER*, 2019, URL: `https://pqc-rollo.org/`.

[3] Martin Albrecht, Carlos Cid, Kenneth G Paterson, Cen Jung Tjhai, and Martin Tomlinson, "NTS-KEM", *in*: *NIST PQC Round* 2 (2019), pp. 4–13.

[4] Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Yann Connan, Jérémie Coulaud, Philippe Gaborit, and Anaïs Kominiarz, "The Rank-Based Cryptography Library", *in*: *Code-Based Cryptography Workshop*, Springer, 2021, pp. 22–41.

[5] Nicolas Aragon, Philippe Gaborit, Adrien Hauteville, and Jean-Pierre Tillich, "A New Algorithm for Solving the Rank Syndrome Decoding Problem", *in*: *2018 IEEE International Symposium on Information Theory (ISIT)*, IEEE, 2018, pp. 2421–2425.

[6] Sarah Arpin, Tyler Raven Billingsley, Daniel Rayor Hast, Jun Bo Lau, Ray Perlner, and Angela Robinson, "A Study of Error Floor Behavior in QC-MDPC Codes", *in*: *International Conference on Post-Quantum Cryptography*, Springer, 2022, pp. 89–103.

[7] Roberto M. Avanzi, Simon Hoerder, Dan Page, and Michael Tunstall, "Side-Channel Attacks on the McEliece and Niederreiter Public-Key Cryptosystems", English, *in*: *Journal of Cryptographic Engineering* 1.*4* (Nov. 2011), pp. 271–281, ISSN: 2190-8508, DOI: `10.1007/s13389-011-0024-9`.

[8] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Gerardo Pelosi, and Paolo Santini, "LEDAcrypt: QC-LDPC Code-Based Cryptosystems with Bounded Decryption Failure Rate", *in*: *Code-Based Cryptography: 7th International Workshop, CBC 2019, Darmstadt, Germany, May 18–19, 2019, Revised Selected Papers 7*, Springer, 2019, pp. 11–43.

[9] Magali Bardet, Pierre Briaud, Maxime Bros, Philippe Gaborit, Vincent Neiger, Olivier Ruatta, and Jean-Pierre Tillich, "An Algebraic Attack on Rank Metric Code-Based Cryptosystems", *in*: *Advances in Cryptology – EUROCRYPT 2020*, ed. by Anne Canteaut and Yuval Ishai, Cham: Springer International Publishing, 2020, pp. 64–93, ISBN: 978-3-030-45727-3.

[10] Magali Bardet, Maxime Bros, Daniel Cabarcas, Philippe Gaborit, Ray Perlner, Daniel Smith-Tone, Jean-Pierre Tillich, and Javier Verbel, "Improvements of Algebraic Attacks for Solving the Rank Decoding and MinRank Problems", *in*: *Advances in Cryptology – ASIACRYPT 2020*, ed. by Shiho Moriai and Huaxiong Wang, Cham: Springer International Publishing, 2020, pp. 507–536, ISBN: 978-3-030-64837-4.

[11] Anja Becker, Antoine Joux, Alexander May, and Alexander Meurer, "Decoding Random Binary Linear Codes in 2 n/20: How 1+ 1= 0 Improves Information Set Decoding", *in*: *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*, Springer, 2012, pp. 520–536.

[12] Elwyn Berlekamp, Robert McEliece, and Henk Van Tilborg, "On the Inherent Intractability of Certain Coding Problems", *in*: *IEEE Transactions on Information Theory* 24.*3* (1978), pp. 384–386.

[13] Daniel J Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, et al., "Classic McEliece: Conservative Code-Based Cryptography", *in*: *NIST submissions* 1.*1* (2017), pp. 1–25.

[14] Daniel J. Bernstein, Tung Chou, and Peter Schwabe, "McBits: Fast Constant-Time Code-Based Cryptography", *in*: *Cryptographic Hardware and Embedded Systems (CHES)*, ed. by Guido Bertoni and Jean-Sébastien Coron, Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250–272, ISBN: 978-3-642-40349-1.

[15] Dan Boneh, Richard A. DeMillo, and Richard J Lipton, "On the Importance of Checking Cryptographic Protocols for Faults", *in*: *International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, 1997, pp. 37–51.

[16] Dominic Bucerzan, Pierre-Louis Cayrel, Vlad Dragoi, and Tania Richmond, "Improved Timing Attacks against the Secret Permutation in the McEliece PKC", *in*: *International Journal of Computers Communications & Control* 12.*1* (2016), pp. 7–25.

[17] Anne Canteaut and Nicolas Sendrier, "Cryptanalysis of the Original McEliece Cryptosystem", *in*: *Advances in Cryptology—ASIACRYPT'98: International Conference on the Theory and Application of Cryptology and Information Security Beijing, China, October 18–22, 1998 Proceedings*, Springer, 1998, pp. 187–199.

[18] C. Chen, T. Eisenbarth, I. von Maurich, and R. Steinwandt, "Horizontal and Vertical Side Channel Analysis of a McEliece Cryptosystem", *in*: *IEEE Transactions on Information Forensics and Security (TIFS)* 11.*6* (June 2016), pp. 1093–1105, ISSN: 1556-6013, DOI: 10.1109/TIFS.2015.2509944.

[19] Cong Chen, Thomas Eisenbarth, Ingo von Maurich, and Rainer Steinwandt, "Differential Power Analysis of a McEliece Cryptosystem", English, *in*: *Applied Cryptography and Network Security (ACNS)*, ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, vol. 9092, Lecture Notes in Computer Science (LNCS), New York, NY, USA: Springer International Publishing, 2015, pp. 538–556, ISBN: 978-3-319-28165-0, DOI: 10.1007/978-3-319-28166-7_26.

[20] Ming-Shing Chen, Tung Chou, and Markus Krausz, "Optimizing BIKE for the Intel Haswell and ARM Cortex-M4", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* 2021.*3* (July 2021), pp. 97–124, DOI: 10.46586/tches.v2021.i3.97-124.

[21] Ming-Shing Chen, Tim Güneysu, Markus Krausz, and Jan Philipp Thoma, "Carry-Less to BIKE Faster", *in*: *International Conference on Applied Cryptography and Network Security (ACNS)*, Springer, 2022, pp. 833–852.

[22] Agathe Cheriere, Nicolas Aragon, Tania Richmond, and Benoît Gérard, "BIKE Key-Recovery: Combining Power Consumption Analysis and Information-Set Decoding", *in*: *International Conference on Applied Cryptography and Network Security (ACNS)*, Springer, 2023, pp. 725–748.

[23] Agathe Cheriere, Lina Mortajine, Tania Richmond, and Nadia El Mrabet, "Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis", *in*: *Workshop on Coding and Cryptography (WCC)* (2022).

[24] Agathe Cheriere, Lina Mortajine, Tania Richmond, and Nadia El Mrabet, "Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis", *in*: *Designs, Codes and Cryptography* (2023), pp. 1–22.

[25] Tung Chou, "McBits Revisited", *in*: *Cryptographic Hardware and Embedded Systems – CHES 2017: 19th International Conference, Proceedings*, ed. by Wieland Fischer and Naofumi Homma, Taipei, Taiwan: Springer International Publishing, Aug. 2017, pp. 213–231, ISBN: 978-3-319-66787-4, DOI: 10.1007/978-3-319-66787-4_11.

[26] Tung Chou, "QcBits: Constant-Time Small-Key Code-Based Cryptography", *in*: *International Conference on Cryptographic Hardware and Embedded Systems*, Springer, 2016, pp. 280–300.

[27] Tung Chou and Jin-Han Liou, "A Constant-Time AVX2 Implementation of a Variant of ROLLO", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES)* (2022), pp. 152–174.

[28] Multi contributor, *Clean, Portable, Tested Implementations of Post-Quantum Cryptography.* URL: `https://github.com/pqclean/pqclean`.

[29] Multi contributor, *Post-Quantum Crypto Library for the ARM Cortex-M4*, URL: `https://github.com/mupq/pqm4`.

[30] Whitfield Diffie and Martin E Hellman, "New Directions in Cryptography", *in*: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*, 1976, pp. 365–390.

[31] Nir Drucker, Shay Gueron, and Dusan Kostic, *Constant-Time Implementations in some Proposed KEMs: the Case of Rollo and RQC*, June 2020.

[32] Nir Drucker, Shay Gueron, and Dusan Kostic, "Fast Polynomial Inversion for Post Quantum QC-MDPC Cryptography", *in*: *Cyber Security Cryptography and Machine Learning*, ed. by Shlomi Dolev, Vladimir Kolesnikov, Sachin Lodha, and Gera Weiss, Cham: Springer International Publishing, 2020, pp. 110–127, ISBN: 978-3-030-49785-9.

[33] Nir Drucker, Shay Gueron, and Dusan Kostic, *On Constant-Time QC-MDPC Decoding with Negligible Failure Rate*, Cryptology ePrint Archive, Report 2019/1289, 2019.

[34] Nir Drucker, Shay Gueron, and Dusan Kostic, "QC-MDPC Decoders with Several Shades of Gray", *in*: *International Conference on Post-Quantum Cryptography*, Springer, 2020, pp. 35–50.

[35] Andre Esser, Alexander May, Javier Verbel, and Weiqiang Wen, "Partial Key Exposure Attacks on BIKE, Rainbow and NTRU", *in*: *Annual International Cryptology Conference*, Springer, 2022, pp. 346–375.

[36] Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zemor, "Low-Rank Parity-Check Codes and their Application to Cryptography", *in*: *International Workshop on Coding and Cryptography (WCC)*, ed. by Lilya Budaghyan, Tor Helleseth, and Matthew G. Parker, ISBN 978-82-308-2269-2, Bergen, Norway, 2013.

[37] Philippe Gaborit, Olivier Ruatta, and Julien Schrek, "On the Complexity of the Rank Syndrome Decoding Problem", *in*: *IEEE Transactions on Information Theory (IT)* 62.*2* (2015), pp. 1006–1019.

[38] Philippe Gaborit and Gilles Zémor, "On the Hardness of the Decoding and the Minimum Distance Problems for Rank Codes", *in*: *IEEE Transactions on Information Theory (IT)* 62.*12* (2016), pp. 7245–7252.

[39] Andrea Galimberti, Davide Galli, Gabriele Montanaro, William Fornaciari, and Davide Zoni, "FPGA Implementation of BIKE for Quantum-Resistant TLS", *in*: *2022 25th Euromicro Conference on Digital System Design (DSD)*, IEEE, 2022, pp. 539–547.

[40] Robert Gallager, "Low-Density Parity-Check Codes", *in*: *IRE Transactions on information theory* 8.*1* (1962), pp. 21–28.

[41] Lov K Grover, "A Framework for Fast Quantum Mechanical Algorithms", *in*: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 53–62.

[42] Qian Guo, Clemens Hlauschek, Thomas Johansson, Norman Lahr, Alexander Nilsson, and Robin Leander Schröder, "Don't Reject this: Key-Recovery Timing Attacks due to Rejection-Sampling in HQC and BIKE", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2022), pp. 223–263.

[43] Richard W Hamming, "Error Detecting and Error Correcting Codes", *in*: *The Bell system technical journal* 29.*2* (1950), pp. 147–160.

[44] Stefan Heyse, Amir Moradi, and Christof Paar, "Practical Power Analysis Attacks on Software Implementations of McEliece", English, *in*: *Proceedings of the Third international conference on Post-Quantum Cryptography (PQCrypto 2010)*, ed. by Nicolas Sendrier, vol. 6061, Lecture Notes in Computer Science (LNCS), Darmstadt, Germany: Springer, June 9, 2010, pp. 108–125, ISBN: 978-3-642-12928-5, DOI: 10.1007/978-3-642-12929-2_9.

[45] Anna-Lena Horlemann, Sven Puchinger, Julian Renner, Thomas Schamberger, and Antonia Wachter-Zeh, "Information-Set Decoding with Hints", *in*: *Code-Based Cryptography Workshop*, Springer, 2021, pp. 60–83.

[46] Paul Kocher, Joshua Jaffe, and Benjamin Jun, "Differential Power Analysis", *in*: (1999), pp. 388–397.

[47] Paul C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", *in*: *Advances in Cryptology – CRYPTO*, ed. by Neal Koblitz, Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[48] Jérôme Lablanche, Lina Mortajine, Othman Benchaalal, Pierre-Louis Cayrel, and Nadia El Mrabet, "Optimized Implementation of the NIST PQC Submission ROLLO on Microcontroller", *in*: *Cryptology ePrint Archive* (2019).

[49] Pil Joong Lee and Ernest F Brickell, "An Observation on the Security of McEliece's Public-Key Cryptosystem", *in*: *Workshop on the Theory and Application of of Cryptographic Techniques*, Springer, 1988, pp. 275–280.

[50] Jeffrey S Leon, "A Probabilistic Algorithm for Computing Minimum Weights of Large Error-Correcting Codes", *in*: *IEEE Transactions on Information Theory* 34.*5* (1988), pp. 1354–1359.

[51] J MacQueen, "Classification and Analysis of Multivariate Observations", *in*: *5th Berkeley Symp. Math. Statist. Probability*, 1967, pp. 281–297.

[52] Ingo von Maurich, Lukas Heberle, and Tim Güneysu, "IND-CCA Secure Hybrid Encryption from QC-MDPC Niederreiter", *in*: *Post-Quantum Cryptography: 7th International Workshop, PQCrypto 2016, Fukuoka, Japan, February 24-26, 2016, Proceedings 7*, Springer, 2016, pp. 1–17.

[53] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loic Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Phillippe Gaborit, and Gilles Zemor, "Rank Quasi-Cyclic (RQC), NIST Submission, 2017.", *in*: *Internet: http://pqc-rqc. org* ().

[54] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, Gilles Zémor, and IC Bourges, "Hamming Quasi-Cyclic (HQC)", *in*: *NIST PQC Round* 2.*4* (2018), p. 13.

[55] Rafael Misoczki, Jean-Pierre Tillich, Nicolas Sendrier, and Paulo SLM Barreto, "MDPC-McEliece: New McEliece Variants from Moderate Density Parity-Check Codes", *in*: *2013 IEEE international symposium on information theory*, IEEE, 2013, pp. 2069–2073.

[56] Lina Mortajine, Othman Benchaalal, Pierre-Louis Cayrel, Nadia El Mrabet, and Jérôme Lablanche, "Optimized and Secure Implementation of ROLLO-I", *in*: *Code-Based Cryptography Workshop*, Springer, 2020, pp. 117–137.

[57] *MUPQ git: Implementation Second Round NIST Schemes for ARM Cortex-M4*, Source code available at `https://github.com/mupq/mupq/tree/Round2/crypto_kem`.

[58] NewAE, *NewAE Hardware Product Documentation CW308 UFO*, URL: `https://rtfm.newae.com/Targets/CW308%20UFO/`.

[59] NIST, *Glossary of Cryptography and Cybersecurity by NIST*, URL: `https://csrc.nist.gov/glossary`.

[60] NIST, *Post-Quantum Cryptography*, URL: `https://csrc.nist.gov/projects/post-quantum-cryptography`.

[61] NIST, *Status Report on the Second Round of the NIST Post-Quantum Cryptography Standardization Process*, URL: `https://nvlpubs.nist.gov/nistpubs/ir/2020/NIST.IR.8309.pdf`.

[62] NIST, *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*, URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[63] Mohammad Reza Nosouhi, Syed W Shah, Lei Pan, Yevhen Zolotavkin, Ashish Nanda, Praveen Gauravaram, and Robin Doss, "Weak-Key Analysis for BIKE Post-Quantum Key Encapsulation Mechanism", *in*: *IEEE Transactions on Information Forensics and Security* 18 (2023), pp. 2160–2174.

[64] Martin Petrvalský, Tania Richmond, Miloš Drutarovský, Pierre-Louis Cayrel, and Viktor Fischer, "Differential Power Analysis Attack on the Secure Bit Permutation in the McEliece Cryptosystem", *in*: *RadioElektronika 2016* (Apr. 2016), pp. 132–137, DOI: 10.1109/RADIOELEK.2016.7477382.

[65] Eugene Prange, "The Use of Information Sets in Decoding Cyclic Codes", *in*: *IRE Transactions on Information Theory* 8.*5* (1962), pp. 5–9.

[66] Jan Richter-Brockmann, Ming-Shing Chen, Santosh Ghosh, and Tim Güneysu, "Racing BIKE: Improved Polynomial Multiplication and Inversion in Hardware", *in*: *Cryptology ePrint Archive* (2021).

[67] Jan Richter-Brockmann, Johannes Mono, and Tim Güneysu, "Folding BIKE: Scalable Hardware Implementation for Reconfigurable Devices", *in*: *IEEE Transactions on Computers* 71.*5* (2021), pp. 1204–1215.

[68] Vincent Rijmen and Joan Daemen, "Advanced Encryption Standard", *in*: *Proceedings of federal information processing standards publications, national institute of standards and technology* 19 (2001), p. 22.

[69] Stefan Ritterhoff, Georg Maringer, Sebastian Bitzer, Violetta Weger, Patrick Karl, Thomas Schamberger, Jonas Schupp, and Antonia Wachter-Zeh, "FuLeeca: A Lee-based Signature Scheme", *in*: *Cryptology ePrint Archive* (2023).

[70] Ronald L. Rivest, Adi Shamir, and Len Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *in*: *Communications of the ACM* 21.*2* (1978), pp. 120–126.

[71] Mélissa Rossi, Mike Hamburg, Michael Hutter, and Mark E. Marson, "A Side-Channel Assisted Cryptanalytic Attack against QcBits", *in*: *Cryptographic Hardware and Embedded Systems – CHES 2017*, ed. by Wieland Fischer and Naofumi Homma, Cham: Springer International Publishing, 2017, pp. 3–23.

[72] Peter W. Shor, "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", *in*: *SIAM Journal on Computing* 26.*5* (1997), pp. 1484–1509.

[73] Abdulhadi Shoufan, Falko Strenzke, H. Gregor Molter, and Marc Stöttinger, "A Timing Attack against Patterson Algorithm in the McEliece PKC", English, *in*: *Proceedings of the 12th International Conference on Information, Security and Cryptology (ICISC 2009)*, ed. by Donghoon Lee and Seokhie Hong, vol. 5984, Lecture Notes in Computer Science (LNCS), Seoul, Korea: Springer Berlin Heidelberg, 2010, pp. 161–175, ISBN: 978-3-642-14422-6, DOI: 10.1007/978-3-642-14423-3_12.

[74] 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", *in*: *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), pp. 180–212.

[75] Falko Strenzke, "A Timing Attack against the Secret Permutation in the McEliece PKC", English, *in*: *Proceedings of the Third international conference on Post-Quantum Cryptography (PQCrypto 2010)*, ed. by Nicolas Sendrier, vol. 6061, Lecture Notes in Computer Science (LNCS), Darmstadt, Germany: Springer Berlin Heidelberg, 2010, pp. 95–107, ISBN: 978-3-642-12928-5, DOI: 10.1007/978-3-642-12929-2_8.

[76] Falko Strenzke, "Timing Attacks against the Syndrome Inversion in Code-based Cryptosystems", English, *in*: *The 5th International Workshop on Post-Quantum Cryptography (PQCrypto 2013)*, ed. by Philippe Gaborit, vol. 7932, Lecture Notes in Computer Science (LNCS), Limoges, France: Springer, 2013, pp. 217–230, ISBN: 978-3-642-38615-2, DOI: 10.1007/978-3-642-38616-9_15.

[77] Falko Strenzke, Erik Tews, H Gregor Molter, Raphael Overbeck, and Abdulhadi Shoufan, "Side Channels in the McEliece PKC", *in*: *International Workshop on Post-Quantum Cryptography*, Springer, 2008, pp. 216–229.

[78] Asuka Wakasugi and Mitsuru Tada, "Security Analysis for BIKE, Classic McEliece and HQC against the Quantum ISD Algorithms", *in*: *Cryptology ePrint Archive* (2022).

COLLEGE DOCTORAL BRETAGNE

MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Résistance aux attaques par canaux auxiliaires de primitives cryptographiques basés sur les codes correcteurs d'erreurs

Mot clés : Canaux auxilaires, Analyse de consommation, Temps constant, Code correcteur d'erreur

Résumé : Depuis une trentaine d'années, nous avons connaissance d'attaques ciblant des implantations de cryptosystèmes, exploitant des informations physiques telles que le temps d'exécution. Il est donc naturel de se demander quelles menaces représentent ces attaques pour les implantations de schémas post-quantiques qui seront déployées dans l'industrie. Dans cette thèse, nous nous intéressons plus particulièrement à la résistance des algorithmes cryptographiques à base de codes correcteurs d'erreurs, vis à vis des attaques par canaux auxiliaires. Nous nous sommes focalisés sur deux schémas, ROLLO et BIKE, candidats au second tour de la standardisation post-quantique du NIST. Nous montrons à travers nos travaux que leurs implantations en temps constant sont notamment vulnérables aux attaques par analyse de consommation de courant. Pour mettre en évidence ces vulnérabilités, nous utilisons des techniques telles que l'apprentissage automatique et l'algèbre linéaire. De plus, pour les deux schémas, une seule trace de la consommation de courant est nécessaire pour remonter à la clé privée.

Suite à la mise en évidence de ces vulnérabilités, nous proposons des stratégies de contre-mesures visant à prévenir ces attaques tout en maintenant le temps constant.

Title: Side-Channel Resistance of Cryptographic Primitives Based on Error-Correcting Codes

Keywords: Side channel, Power analysis, Constant time, Error-correcting code

Abstract: For about three decades, we have been aware of attacks targeting implementations of cryptosystems, exploiting physical information such as execution time. Naturally, questions arise about the threats these attacks pose to the upcoming industrial deployments of post-quantum schemes. In this thesis, we focus on the resistance of code-based cryptographic algorithms against side-channel attacks. We specifically studied two schemes, ROLLO and BIKE, which were candidates for the second round of post-quantum standardization organized by NIST. Through our research, we demonstrate that their constant-time implementations are notably vulnerable to attacks using power consumption analysis. To demonstrate these vulnerabilities, we employ techniques such as machine learning and linear algebra. Furthermore, for both schemed, the attackd require a single trace of power consumption to recover the private key. Following the identification of these vulnerabilities, we propose countermeasure strategies to prevent these attacks while maintaining constant-time operation.