Analyse d'algorithmes post-quantiques implantables en pratique

Lina Mortajine

▶ To cite this version:

Lina Mortajine. Analyse d'algorithmes post-quantiques implantables en pratique. Autre. Université de Lyon, 2021. Français. NNT : 2021LYSEM028 . tel-03885862

HAL Id: tel-03885862
https://theses.hal.science/tel-03885862v1

Submitted on 6 Dec 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

N°d'ordre NNT : 2021LYSEM028

THÈSE de DOCTORAT DE L'UNIVERSITÉ DE LYON
opérée au sein de
L'École des Mines de Saint-Étienne

École Doctorale N° 488
Sciences, Ingénierie, Santé

Spécialité de doctorat : Microélectronique

Soutenue publiquement/à huis clos le 18/10/2021, par :
Lina Mortajine

Analyse d'algorithmes post-quantiques implantables en pratique

---

Devant le jury composé de :

Laurent-Stéphane Didier	Professeur, Université de Toulon	Président
Sébastien Canard	Ingénieur, Orange Labs	Rapporteur
Sylvain Guilley	Professeur, Télécom Paris	Rapporteur
Jean-Max Dutertre	Professeur, École des Mines de Saint-Étienne	Examinateur
Nadia El Mrabet	Maître assistante, École des Mines de Saint-Étienne	Directrice de thèse
Pierre-Louis Cayrel	Maître de conférence, Université Jean-Monnet	Co-encadrant
Jean-Pierre Enguent	Responsable R&D, Wisekey	Invité

Spécialités doctorales	Responsables :	Spécialités doctorales	Responsables
SCIENCES ET GENIE DES MATERIAUX	K. Wolski Directeur de recherche	MATHEMATIQUES APPLIQUEES	M. Batton-Hubert
MECANIQUE ET INGENIERIE	S. Drapier, professeur	INFORMATIQUE	O. Boissier, Professeur
GENIE DES PROCEDES	F. Gruy, Maître de recherche	SCIENCES DES IMAGES ET DES FORMES	JC. Pinoli, Professeur
SCIENCES DE LA TERRE	B. Guy, Directeur de recherche	GENIE INDUSTRIEL	N. Absi, Maitre de recherche
SCIENCES ET GENIE DE L'ENVIRONNEMENT	V.Laforest, Directeur de recherche	MICROELECTRONIQUE	Ph. Lalevée, Professeur

EMSE : Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'Etat ou d'une HDR)

ABSI	Nabil	MR	Génie industriel	CMP
AUGUSTO	Vincent	MR	Génie industriel	CIS
AVRIL	Stéphane	PR	Mécanique et ingénierie	CIS
BADEL	Pierre	PR	Mécanique et ingénierie	CIS
BALBO	Flavien	PR	Informatique	FAYOL
BASSEREAU	Jean-François	PR	Sciences et génie des matériaux	SMS
BATTON-HUBERT	Mireille	PR	Mathématiques appliquées	FAYOL
BEIGBEDER	Michel	MA	Informatique	FAYOL
BILAL	Blayac	DR	Sciences et génie de l'environnement	SPIN
BLAYAC	Sylvain	PR	Microélectronique	CMP
BOISSIER	Olivier	PR	Informatique	FAYOL
BONNEFOY	Olivier	PR	Génie des Procédés	SPIN
BORBELY	Andras	DR	Sciences et génie des matériaux	SMS
BOUCHER	Xavier	PR	Génie Industriel	FAYOL
BRUCHON	Julien	PR	Mécanique et ingénierie	SMS
CAMEIRAO	Ana	PR	Génie des Procédés	SPIN
CHRISTIEN	Frédéric	PR	Science et génie des matériaux	SMS
DAUZERE-PERES	Stéphane	PR	Génie Industriel	CMP
DEBAYLE	Johan	MR	Sciences des Images et des Formes	SPIN
DEGEORGE	Jean-Michel	MA	Génie industriel	Fayol
DELAFOSSE	David	PR	Sciences et génie des matériaux	SMS
DELORME	Xavier	PR	Génie industriel	FAYOL
DESRAYAUD	Christophe	PR	Mécanique et ingénierie	SMS
DJENIZIAN	Thierry	PR	Science et génie des matériaux	CMP
BERGER-DOUCE	Sandrine	PR	Sciences de gestion	FAYOL
DRAPIER	Sylvain	PR	Mécanique et ingénierie	SMS
DUTERTRE	Jean-Max	PR	Microélectronique	CMP
EL MRABET	Nadia	MA	Microélectronique	CMP
FAUCHEU	Jenny	MA	Sciences et génie des matériaux	SMS
FAVERGEON	Loïc	MR	Génie des Procédés	SPIN
FEILLET	Dominique	PR	Génie Industriel	CMP
FOREST	Valérie	PR	Génie des Procédés	CIS
FRACZKIEWICZ	Anna	DR	Sciences et génie des matériaux	SMS
GAVET	Yann	MA	Sciences des Images et des Formes	SPIN
GERINGER	Jean	MA	Sciences et génie des matériaux	CIS
GONDRAN	Natacha	MA	Sciences et génie de l'environnement	FAYOL
GONZALEZ FELIU	Jesus	MA	Sciences économiques	FAYOL
GRAILLOT	Didier	DR	Sciences et génie de l'environnement	SPIN
GRIMAUD	Frederic	EC	Génie mathématiques et industriel	FAYOL
GROSSEAU	Philippe	DR	Génie des Procédés	SPIN
GRUY	Frédéric	PR	Génie des Procédés	SPIN
HAN	Woo-Suck	MR	Mécanique et ingénierie	SMS
HERRI	Jean Michel	PR	Génie des Procédés	SPIN
ISMAILOVA	Esma	MC	Microélectronique	CMP
KERMOUCHE	Guillaume	PR	Mécanique et Ingénierie	SMS
KLOCKER	Helmut	DR	Sciences et génie des matériaux	SMS
LAFOREST	Valérie	DR	Sciences et génie de l'environnement	FAYOL
LERICHE	Rodolphe	DR	Mécanique et ingénierie	FAYOL
LIOTIER	Pierre-Jacques	MA	Mécanique et ingénierie	SMS
MEDINI	Khaled	EC	Sciences et génie de l'environnement	FAYOL
MOLIMARD	Jérôme	PR	Mécanique et ingénierie	CIS
MOULIN	Nicolas	MA	Mécanique et ingénierie	SMS
MOUTTE	Jacques	MR	Génie des Procédés	SPIN
NAVARRO	Laurent	MR	Mécanique et ingénierie	CIS
NEUBERT	Gilles	PR	Génie industriel	FAYOL
NIKOLOVSKI	Jean-Pierre	Ingénieur de recherche	Mécanique et ingénierie	CMP
O CONNOR	Rodney Philip	PR	Microélectronique	CMP
PICARD	Gauthier	PR	Informatique	FAYOL
PINOLI	Jean Charles	PR	Sciences des Images et des Formes	SPIN
POURCHEZ	Jérémy	DR	Génie des Procédés	CIS
ROUSSY	Agnès	MA	Microélectronique	CMP
SANAUR	Sébastien	MA	Microélectronique	CMP
SERRIS	Eric	IRD	Génie des Procédés	FAYOL
STOLARZ	Jacques	CR	Sciences et génie des matériaux	SMS
VALDIVIESO	François	PR	Sciences et génie des matériaux	SMS
VIRICELLE	Jean Paul	DR	Génie des Procédés	SPIN
WOLSKI	Krzystof	DR	Sciences et génie des matériaux	SMS
XIE	Xiaolan	PR	Génie industriel	CIS
YUGMA	Gallian	MR	Génie industriel	CMP

Mise à jour : 04/02/2021

Remerciements

Je remercie chaleureusement toutes les personnes qui m'ont aidée, de près ou de loin, dans l'élaboration de ma thèse.

Tout d'abord, je tiens à exprimer toute ma reconnaissance à ma directrice de thèse, Nadia El Mrabet, qui m'a accompagnée durant ces trois dernières années. Elle a toujours su trouver les mots pour m'encourager et me guider dans mes premiers pas de chercheuse. Je la remercie chaleureusement pour tous ses conseils et son aide.

Un grand merci à Vincent Dupaquis et Othman Benchaalal qui se sont succédé dans le co-encadrement de ma thèse à Wisekey. Ils m'ont réservé un bon accueil pour travailler dans de bonnes conditions. Je les remercie pour la confiance qu'ils m'ont accordée et pour l'opportunité qu'ils m'ont offerte, cela a été déterminant dans l'élaboration de ma thèse. J'ai beaucoup appris auprès d'eux.
Merci à Pierre-Louis Cayrel qui, malgré la distance, a toujours été disponible pour répondre à mes questions. Ses remarques pertinentes m'ont été d'une grande utilité pour améliorer ce mémoire.

Un grand merci également à Sébastien Canard et Sylvain Guilley d'avoir accepté de prendre du temps pour relire cette thèse et d'en être les rapporteurs. Je remercie Laurent-Stéphane Didier et Jean-Max Dutertre de l'intérêt qu'ils ont porté à mes travaux en acceptant de faire partie de mon jury de thèse. Merci à Jean-Pierre Enguent d'avoir pris le temps de venir assister à ma soutenance, en tant que représentant de Wisekey. Je le remercie de s'être toujours montré intéressé par mes travaux.

Je remercie chaleureusement toutes les personnes que j'ai pu rencontrer à Wisekey (et les Nagraboys) avec qui j'ai pu échanger sur de nombreux sujets, ils ont permis de rendre ces dernières années tellement agréables. Merci à Michel de s'être occupé des démarches administratives pour ma thèse. Je tiens à remercier le "groupe des jeunes" : Jérôme, Juliette, Anne-Claire et Cyril, ils sont devenus au fil du temps de vrais amis et c'est toujours un plaisir de se retrouver. Merci également à Sandrine pour les soirées jeux de société qui ont toujours un réel succès.

Je tiens vivement à remercier toute l'équipe SAS de l'École des Mines pour leur accueil. C'était un réel plaisir d'aller travailler tous les jours à l'École ces derniers mois. Merci à tous les doctorants que j'ai pu y côtoyer durant ces dernières années,

ils ont grandement participé à rendre ces journées agréables : Élise, Anthony, Kevin, Raphaël, Clément, Arthur, Nathan, Baptiste, Rémi, Roukoz, William, Joseph, François, Alexandre et aujourd'hui docteures Mounia et Amina. Je remercie l'équipe administrative de l'École des Mines qui nous facilitent les démarches au quotidien.

Mes remerciements vont également à Tania Richmond et Agathe Cheriere, j'ai vraiment apprécié notre collaboration. Merci à Tania d'avoir pris le temps de répondre à mes questions (malgré son emploi du temps très chargé) et pour tous ses nombreux conseils toujours très utiles. Merci d'avoir fait le déplacement depuis Rennes pour assister à ma soutenance.

Bien sûr, l'aboutissement de cette thèse aurait été compliqué sans le soutien inconditionnel de mes amis (je sais qu'ils se reconnaîtront) et de ma famille. Merci d'avoir toujours trouvé les mots pour m'encourager et de m'apporter joie et bonne humeur au quotidien. Je remercie sincèrement mes parents, ma soeur et mon frère de s'être montrés aussi présents, merci pour leur patience, leur écoute et leur aide durant toutes ces années. Cette thèse n'aurait pu être ce qu'elle est sans eux.

Table des matières

Introduction **5**

1 État de l'art **12**
1.1 Mathématiques . . . 12
1.1.1 Introduction aux codes correcteurs d'erreurs . . . 12
1.1.1.1 Généralités sur les codes linéaires . . . 13
1.1.1.2 Encodage et décodage . . . 16
1.1.1.3 Codes LRPC et la structure de code utilisé . . . 16
1.1.1.4 Problèmes difficiles liés aux codes correcteurs . . . 18
1.1.1.5 Décodage des codes LRPC . . . 19
1.1.2 Introduction aux réseaux euclidiens . . . 20
1.1.2.1 Généralités sur les réseaux euclidiens . . . 21
1.1.2.2 Problèmes difficiles liés aux réseaux euclidiens . . . 23
1.1.2.3 Résolution des problèmes sur les réseaux . . . 26
1.2 Microélectronique . . . 28
1.2.1 Sécurité matérielle . . . 28
1.2.1.1 Généralités sur les attaques par canaux auxiliaires . . . 28
1.2.1.2 Contre-mesures . . . 35
1.2.2 Matériel utilisé . . . 36
1.3 Candidats étudiés . . . 37
1.3.1 ROLLO . . . 37
1.3.2 NTRU . . . 39
1.3.3 Crystals-Dilithium . . . 39

I Codes correcteurs d'erreurs **41**

2 Étude du candidat ROLLO **42**
2.1 Implantation de ROLLO-I . . . 44
2.1.1 Opérations sur $\mathbb{F}_2[z]/(P_m)$ . . . 44
2.1.2 Opérations dans $\mathbb{F}_{2^m}[Z]/(P_n)$ . . . 47
2.1.3 Évaluation des performances . . . 53
2.1.4 Nouvelle version de ROLLO . . . 54
2.2 Attaque par canaux auxiliaires . . . 56
2.2.1 Fuites d'information de l'implantation standard . . . 58

2.2.2 Fuites d'information de l'implantation de référence en temps constant . . . 61
2.2.3 Fuites d'information de l'implantation en temps constant sur *GitHub* . . . 66
2.2.4 Résultats expérimentaux des attaques par canaux auxiliaires . 70
2.3 Contre-mesures . . . 75
2.3.1 Pour l'implantation de la standardisation . . . 75
2.3.2 Pour les implantations en temps constant . . . 76
2.3.3 Résultats expérimentaux . . . 79
2.4 Conclusion . . . 82

II Réseaux euclidiens 83

3 Étude de NTRU 84
3.1 Attaque par canaux auxiliaires sur NTRU . . . 86
3.1.1 Attaques existantes . . . 86
3.1.2 Opération ciblée . . . 87
3.1.3 Attaques profilées . . . 88
3.1.3.1 Simulation . . . 88
3.1.3.2 Expérimentation . . . 93
3.2 Proposition de contre-mesures . . . 95
3.3 Conclusion . . . 97

4 Étude de Crystals-Dilithium 98
4.1 Opérations et coût mémoire des éléments . . . 100
4.1.1 Génération des clés . . . 102
4.1.2 Signature . . . 103
4.1.3 Vérification . . . 104
4.2 Optimisations réalisées . . . 105
4.2.1 Génération des clés . . . 106
4.2.2 Signature . . . 107
4.2.3 Vérification . . . 108
4.3 Résultats expérimentaux . . . 109
4.3.1 État de l'art . . . 110
4.3.2 Résultats obtenus . . . 110
4.4 Conclusion . . . 111

Conclusion et perspectives 112

Annexes 113

A Algorithmes 114
A.1 Algorithme d'inversion sur $\mathbb{F}_{2^m}$ tiré de [HMV04, Algo. 2.48] . . . 114
A.2 Algorithme de Fisher-Yates utilisé pour permuter aléatoirement les éléments d'une liste L . . . 115

B Codes sources des attaques **116**
B.1 Code source de l'attaque sur l'implantation de pivot de Gauss de référence, en temps constant (en utilisant *SageMath*) 116
B.2 Code source de l'attaque sur l'implantation du pivot de Gauss en temps constant disponible sur *GitHub* (en utilisant *SageMath*) 117

Table des figures

1.1 Codage du canal entre un expéditeur E et un récepteur R. 13
1.2 Distance minimale d et capacité de correction t d'un code $\mathcal{C}$ 15
1.3 Réseau euclidien $\mathcal{L}$ sur $\mathbb{R}^2$ avec $\{\mathbf{u}_1, \mathbf{u}_2\}$ et $\{\mathbf{v}_1, \mathbf{v}_2\}$ deux bases de $\mathcal{L}$. . 21
1.4 Réseau $\mathcal{L}$ sur $\mathbb{R}^2$ où $\mathbf{u}_1$ et $\mathbf{u}_2$ sont les vecteurs les plus courts et $\mathbf{v}$ le vecteur le plus proche de $\mathbf{w} \in \mathbb{R}^2$. 23
1.5 Attaque non profilée 31
1.6 Attaque profilée 34
1.7 Illustration du banc d'attaque pour l'acquisition des traces 37

2.1 Cryptosystèmes ROLLO-I (KEM) et ROLLO-II (PKE) 43
2.2 Réduction du quatrième mot de 32 bits du polynôme $c(z)$ modulo $P_m(z)$ 47
2.3 Nombre de cycles requis sur CORTEX-M3 par la méthode Karatsuba combinée avec la multiplication élémentaire en fonction du nombre de coefficients des polynômes donnés en entrée. 49
2.4 Opérations lors de la décapsulation et du déchiffrement 57
2.5 Opérations sur les lignes de la matrice en fonction des valeurs des masques. En rouge, les chemins menant à des XOR sur les lignes où s_p dénote la ligne pivot et s_i la ligne traitée. 67
2.6 Fonction tirée de l'implantation en temps constant sur *GitHub* 67
2.7 Partie de la trace de consommation de l'implantation du pivot de Gauss standard avec un zoom sur la première colonne - en rouge le motif représentant un XOR entre deux lignes (le coefficient est à 1), en vert le motif lorsqu'il n'y pas d'opération (le coefficient est à 0). . 71
2.8 Trace des deux premières boucles intérieures **pour** - en rouge le motif représentant le traitement d'une multiplication avec la valeur de masque à 1, et en vert celui d'un masque à 0. 72
2.9 Trace d'une première boucle intérieure **pour** - en rouge le motif représentant le traitement d'une multiplication lorsque $masque = 1$, et en vert lorsque $masque = 0$. 73
2.10 Trace de la deuxième boucle intérieure **pour** - en rouge le motif représentant le traitement d'une multiplication lorsque $masque = 1$, et en vert lorsque $masque = 0$. 74
2.11 Partie de la trace de consommation de l'exécution de la réduction de la matrice sous forme échelonnée avec un zoom sur le traitement des premières lignes de la colonne 1. 75

2.12 Partie de la trace de consommation de l'implantation du pivot de Gauss avec l'ajout d'opérations factices pour la standardisation avec un zoom sur la première colonne . . . 80
2.13 Partie de la trace de consommation de l'implantation du pivot de Gauss en temps constant avec masquage de la multiplication et du XOR . . . 82

3.1 Chiffrement NTRU . . . 85
3.2 Architecture d'un réseau de neurones convolutifs. La couche de sortie utilise la fonction d'activation Softmax (définie dans l'Équation 3.2). . 89
3.3 Méthode SOSD appliquée sur des traces dont un coefficient a été modifié (en abscisse échantillonnage de points et en ordonnée, le résultat de la somme présentée à l'équation 3.1). . . . 91
3.4 Évaluation de la fonction de perte durant la phase d'entraînement . . 93
3.5 Taux de succès de l'attaque suivant le nombre de traces apprises . . . 93
3.6 Partie d'une trace de la première boucle interne **pour** de l'Algorithme 14 et méthode SOST appliquée sur les traces dont le coefficient f_1 a été modifié . . . 94

4.1 Version simplifiée de la signature Crystals-Dilithium . . . 99
4.2 Création d'un polynôme dans l'ensemble B_τ. . . . 103
4.3 Calcul de $\mathbf{t} = (t_0, \dots, t_{k-1})$ . . . 106
4.4 Calcul de $\mathbf{w}_1 = \texttt{HighBits}(\mathbf{A} \cdot \mathbf{y})$ . . . 107
4.5 Calcul de $\mathbf{z}$ . . . 107
4.6 Calcul de $\mathbf{A} \cdot \mathbf{z} - c\mathbf{t}$ . . . 109

Liste des tableaux

1 Notations et définitions 1
2 Acronymes 3
3 Niveau de sécurité requis pour la standardisation post-quantique du NIST. 10
4 Candidats finalistes et alternatifs de la standardisation post-quantique du NIST 11

1.1 Propriétés du microcontrôleur utilisé 36
1.2 Candidats du second tour à base de codes en métrique de Hamming et en métrique rang correcteurs 38
1.3 Taille des éléments en octet un niveau de sécurité 5. 38
1.4 Taille des nouveaux paramètres (en octet) 39
1.5 Taille des paramètres pour les schémas de signature du deuxième tour de la standardisation 40

2.1 Les paramètres de ROLLO pour les trois niveaux de sécurité 44
2.2 Gains mémoire avec la version modifiée de RSR 53
2.3 Temps d'exécution du cryptosystème ROLLO-I 53
2.4 Coût mémoire pour ROLLO-I (en octet) 54
2.5 Comparaison des performances entre ROLLO-I et ECDH pour deux niveaux de sécurité différents. 54
2.6 Les paramètres de ROLLO pour les trois niveaux de sécurité 55
2.7 Performances et coût mémoire de ROLLO-I-128 avec les nouveaux paramètres 56
2.8 Temps d'exécution de la décapsulation avec et sans contre-mesure . . 81
2.9 Temps d'exécution des implantations du pivot de Gauss en temps constant pour les paramètres de ROLLO-I-128 81

3.1 Paramètres pour la soumission NTRU 84
3.2 Ensembles et fonction utilisés suivant les paramètres 86
3.3 Taux de réussite de l'attaque sur une seule trace en fonction de la valeur du coefficient 94
3.4 Taux de réussite de l'attaque en fonction de la valeur du coefficient - STM32 97

4.1 Paramètres de Crystals-Dilithium pour les trois niveaux de sécurité requis par le NIST 100

4.2 Coût mémoire des éléments dans le procédé de génération des clés. . . 103
4.3 Coût mémoire des éléments dans le procédé de signature. 104
4.4 Coût mémoire des éléments dans le procédé de vérification de signature. 105
4.5 Comparaison de l'espace mémoire requis (en octets) pour exécuter les trois algorithmes composant la signature Cyrstals-Dilithium avec les paramètres du second tour ainsi que les performances de l'implantation 111
4.6 Résultats pour les paramètres du troisième tour 111

Notations et acronymes

TABLE 1 – Notations et définitions

Partie sur les codes correcteurs d'erreurs	
$\mathbb{F}_q$	le corps fini à q éléments avec q une puissance d'un nombre premier
$\mathbb{F}_{q^m}$	le corps fini à q^m éléments avec q une puissance d'un nombre premier
$\mathbb{F}_q^{n\times m}$	ensemble des matrices de taille $n \times m$ dont les coefficients sont dans $\mathbb{F}_q$
(P)	idéal engendré par le polynôme P
$\langle \mathbf{x} \rangle$	ensemble engendré par $\mathbf{x}$
I_n	matrice identité de taille $n \times n$
$\mathcal{C}$	un code correcteur d'erreurs
$\mathbf{G}$	matrice génératrice d'un code
$\mathbf{H}$	matrice de parité d'un code
$[n,k]_q$	code $\mathbb{F}_q$-linéaire de longueur n et de dimension k
$\mathbf{s}$	syndrome d'un mot de code
$w_H(\mathbf{x})$	poids de Hamming du mot de code $\mathbf{x}$
$\mathbb{F}_{q^m}^n$	l'espace vectoriel isomorphe à $\mathbb{F}_{q^m}[Z]/(P_n)$ avec P_n un polynôme irréductible sur $\mathbb{F}_{q^m}$ de degré n
$M(\mathbf{x})$	la matrice associée au vecteur $\mathbf{x} \in \mathbb{F}_{q^m}^n$
$Supp(\mathbf{x})$	le support de $\mathbf{x} \in \mathbb{F}_{q^m}^n$
$\lVert\mathbf{x}\rVert$	rang du vecteur $\mathbf{x}$
$d(\mathbf{x},\mathbf{y})$	distance en métrique de Hamming ou en métrique rang entre deux mots de code $\mathbf{x}$ et $\mathbf{y}$
$Card(A)$	cardinal de l'ensemble A

$\binom{n}{w}$	coefficient binomial de Newton
$\binom{m}{w}_q$	coefficient binomial de Gauss
$d_{\min}$	distance minimale d'un code
$[n, k, d_{\min}]$	code $\mathbb{F}_q$-linéaire de longueur n, de dimension k et de distance minimale $d_{\min}$
$\mathcal{B}(\mathbf{x}, t)$	boule de centre $\mathbf{x}$ et de rayon t
$\mathcal{IM}(\mathbf{g})$	matrice idéale générée à partir de $\mathbf{g}$
Partie sur les réseaux euclidiens	
$\mathcal{L}$	un réseau euclidien
B	base d'un réseau
$det(\mathcal{L})$	déterminant d'un réseau
$\lambda_1(\mathcal{L})$	distance minimale du réseau $\mathcal{L}$
$\rho(B)$	défaut d'orthogonalité d'un réseau $\mathcal{L}$ de base B
$dist(\mathbf{w}, \mathcal{L})$	distance entre $\mathbf{w}$ et le réseau $\mathcal{L}$
$\mathbf{\Phi}_1$	le polynôme $x - 1$
$\mathbf{\Phi}_n$	le polynôme $x^{n-1} + x^{n-2} + \cdots + 1$, avec $n \geq 2$
$\mathcal{R}$	l'anneau quotient $\mathbb{Z}[x]/(x^n - 1)$
$\mathcal{S}$	l'anneau quotient $\mathbb{Z}[x]/(\mathbf{\Phi}_n)$
$\mathcal{R}_q$	l'anneau quotient $\mathbb{Z}_q[x]/(x^n - 1)$
$\mathcal{S}_q$	l'anneau quotient $\mathbb{Z}_q[x]/(\mathbf{\Phi}_n)$
polynôme ternaire	un polynôme dont les coefficients sont dans $\{-1, 0, 1\}$
$\mathcal{T}$	l'ensemble des polynômes ternaires non nuls dont les degrés sont au plus $n - 2$
$\mathcal{T}(d)$	le sous-ensemble de $\mathcal{T}$ dont les polynômes ont $d/2$ coefficients à $+1$ et $d/2$ coefficient à -1
Propriété de corrélation non négative	un polynôme ternaire $\mathbf{f} = (f_0, f_1, \cdots, f_{n-2})$ tel que $\sum_i f_i f_{i+1} \geq 0$
$\mathcal{T}_+$	le sous-ensemble de $\mathcal{T}$ avec la propriété de corrélation non négative
$\mathcal{L}_m, \mathcal{L}_f, \mathcal{L}_g$	ensembles d'échantillonnage du message et de polynômes ternaires pour la clé

Lift	fonction injective $\mathcal{L}_m \to \mathbb{Z}[x]$ telle que $\mathbf{Lift(m)} \equiv \mathbf{m} \mod (3, \mathbf{\Phi}_n)$

Opérateurs	
$\otimes$	multiplication entre mots de n bits
$\star$	multiplication modulaire
$\cdot$	multiplication polynomiale dans $\mathbb{F}_{q^m}[Z]/(P_n)$
$\oplus$	opérateur logique OU EXCLUSIF
$\wedge$	opérateur logique ET
&	opérateur bit à bit ET
$\neg$	opérateur logique NON
$\xleftarrow{\$}$	élément généré de manière aléatoire dans un ensemble donné
$\xleftarrow{\$_\rho}$	élément généré de manière aléatoire dans un ensemble donné à partir d'une graine ρ
>> i	décalage logique de i bits vers la droite

Table 2 – Acronymes

AES	Advanced Encryption Standard
BDD	Bounding Decoding Problem
CMOS	Complementary Metal-Oxide Semiconductor
CPA	Correlation Power Analysis
CVP	Closest Vector Problem
DES	Data Encryption Standard
DPA	Differential Power Analysis
ECDSA	Elliptic Curve Digital Signature Algorithm
FPGA	Field Programmable Gate Array
IoD	Internet des Objets
KEM	Key Encapsulation Mechanism
LLL	algorithme de réduction de réseaux (Lenstra-Lenstra-Lovàsz)
LRPC	Low Rank Parity Check

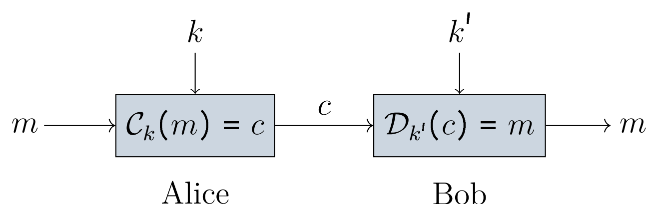
LWE	Learning With Errors
Mémoire Flash	mémoire de masse à semi-conducteurs ré-inscriptible
MLWE	Search Module Learning With Errors
MSIS	Search Module Short Integer Solution
NIST	National Institute of Standards and Technology
NP	non déterministe polynomial (problème dont le résultat peut être vérifié en temps polynomial)
NTT	Number Theoric Transform
PKE	Public Key Encryption
PQC	Post-Quantum Cryptography
RAM	Random-access memory
RSA	algorithme de chiffrement (Rivest-Shamir-Adleman)
RSD	Rank Syndrome Decoding
RSR	Rank Support Recovery
SHA	Secure Hash Algorithm
SHAKE	Secure Hash Algorithm and KECCAK
SIS	Short Integer Solution
SPA	Simple Power Analysis
SVP	Shortest Vector Problem
TRNG	True Random Number Generator

Introduction

Contexte de la thèse

Cette thèse se situe à la jonction entre la cryptographie, l'algorithmique et la sécurité matérielle d'une implantation, le tout dans un contexte industriel. Plus particulièrement, les travaux portent sur l'étude d'algorithmes post-quantiques pour des intégrations dans des environnements restreints tels que les microcontrôleurs afin de préparer la transition vers la cryptographie post-quantique et l'intégration de futurs standards.
Afin d'expliquer plus en détails le contexte de la thèse, nous revenons, dans un premier temps, sur les différentes notions composant cette dernière.

Cryptographie. À l'ère du numérique, les puces électroniques se développent de plus en plus et nous les retrouvons ainsi dans de nombreux objets de notre quotidien tels que les passeports, les cartes à puces, les badges ou encore les téléphones. Ces objets, amenés à interagir avec l'environnement extérieur, contiennent souvent nos informations privées, il devient alors essentiel de protéger ces interactions d'une menace extérieure. Pour cela, nous faisons appel à la cryptographie qui est une science regroupant un ensemble de procédés pour le chiffrement et déchiffrement de données afin d'assurer une communication sécurisée entre l'émetteur et le destinataire. La sécurité de l'information transmise repose sur les principes fondamentaux auxquels la cryptographie doit répondre tels que la confidentialité qui garantit que l'information n'est accessible qu'à ceux qui en ont l'accès, l'authenticité qui garantit la légitimité de l'information et l'intégrité qui garantit la non-altération de l'information par une personne non autorisée. Deux entités peuvent alors se transmettre des informations confidentielles par la connaissance d'un secret, appelé clé, leur permettant de chiffrer ou déchiffrer leurs messages. L'expéditeur peut alors chiffrer son message clair m par le biais d'une fonction de chiffrement $\mathcal{C}$ et d'une clé k et obtient un message chiffré c. Le destinataire déchiffre le texte chiffré c par le biais d'une fonction de déchiffrement $\mathcal{D}$ et une clé k'.



La cryptographie se distingue en deux catégories :

- La cryptographie symétrique, également appelée cryptographie à clé secrète, utilise la même clé pour chiffrer et déchiffrer, $k = k'$. Apparue dans l'antiquité avec le chiffrement de César qui consiste en un décalage de lettres, elle s'est amplement développée à la fin du 20e siècle avec des techniques de chiffrement plus complexes tels que le 3DES [Nat99], l'AES [Nat01] (standard actuel) ou encore Blowfish [Sch94], non standardisé mais également utilisé. Pour que l'on puisse communiquer avec des algorithmes symétriques, les entités doivent posséder une clé secrète commune qu'ils auront préalablement échangée via un canal sécurisé. Cependant, les données transitent souvent via des canaux publics posant alors la question de l'échange du secret de manière sécurisée entre les deux entités.
- La cryptographie asymétrique, développée en 1976 par Diffie et Hellman [DH76], a permis de répondre à ce besoin en se basant sur l'utilisation de deux clés, $k \neq k'$: la clé publique k, permettant le chiffrement, est connue de quiconque et la clé privée k', permettant le déchiffrement, est seulement connue de l'entité ayant généré la paire de clés. La cryptographie asymétrique regroupe différents protocoles permettant de répondre à certaines fonctionnalités :
  - le protocole d'échange de clés, tel que celui proposé par Diffie et Hellman [DH76], permet à Alice et Bob de se mettre d'accord sur une clé commune symétrique.
  - le mécanisme d'encapsulation de clé (*Key encapsulation mechanism* - KEM) est composé de trois algorithmes : un algorithme de génération de clés qui retourne une paire de clés (privée et publique), un algorithme d'encapsulation (Encap) qui utilise la clé publique pour générer une clé symétrique et un chiffré, et un algorithme de décapsulation (Décap) qui utilise la clé privée pour obtenir la même clé symétrique à partir du chiffré reçu.
  - le système de chiffrement à clé publique (*Public key encryption* - PKE) est également composé de trois algorithmes : un algorithme de génération de clés qui retourne une paire de clés (privée et publique), un algorithme de chiffrement qui utilise la clé publique pour chiffrer un message et génère donc un chiffré, et un algorithme de déchiffrement qui utilise la clé privée pour déchiffrer le message à partir du chiffré reçu.
  - la signature numérique consiste également à l'exécution de trois algorithmes : la génération de la paire de clés (privée et publique) se réalise via l'algorithme de génération de clé, l'algorithme de signature utilise une clé privée pour chiffrer un message et ainsi générer une signature, et l'algorithme de vérification utilise la clé publique pour déchiffrer la signature et ainsi comparer avec le message initial. Ce dernier retourne la valeur "vrai" ou "faux" en fonction du résultat de la comparaison.

Les systèmes de chiffrement asymétrique (ou cryptosystèmes) actuellement déployés basent leur sécurité sur la difficulté de problèmes mathématiques de la théorie des nombres tels que le problème de la factorisation de grands entiers en nombres premiers, utilisé dans la construction du cryptosystème RSA [RSA78] et le problème du logarithme discret, utilisé dans l'échange de clé Diffie-Hellman ou encore le protocole de signature ECDSA [JMV01]. Ces cryptosystèmes sont alors dits sûrs car de nos jours, il n'existe aucun algorithme permettant de résoudre les problèmes mathématiques de la théorie des nombres en temps polynomiale. Ainsi, la puissance de

calcul nécessaire à la résolution de ces problèmes est telle qu'ils nous est impossible de l'atteindre avec des ordinateurs classiques. Ces problèmes sont alors dits NP-complets ou NP-difficiles, où NP est la classe de complexité contenant les problèmes de décision qui peuvent être résolus par un algorithme polynomial non déterministe. Or, depuis quelques années, la menace de calculateurs beaucoup plus puissants que ceux que nous utilisons actuellement s'intensifie. Ces calculateurs reposent sur la technologie quantique dont le développement ne fait que s'accroître.

La menace quantique. Au début des années 1980, Richard Feynman est le premier à évoquer la notion de calculateur quantique mais, à cette époque, l'idée est vue comme utopique. En effet, ces ordinateurs qui sont composés de bits quantiques (qbits) utilisent deux lois fondamentales de la physique quantique : le principe de superposition et l'intrication quantique. Un des principaux problèmes à la construction de l'une de ces machines est le maintien des qbits dans leur état quantique. L'environnement joue un rôle primordial : en raison de leur structure fragile, ils doivent être maintenus à des températures proches du zéro absolu soit $-273,15^{\circ}$C. De plus, ces qbits doivent interagir avec un nouveau type de logique (des portes "quantiques") qui implantent ces nouvelles formes de traitement de l'information. Bien que le développement de ces ordinateurs soit bénéfique dans de nombreux domaines, il représente une réelle menace pour le domaine de la sécurité. En 1994, Peter Shor publia un algorithme capable de factoriser n'importe quel nombre entier de grande taille et de résoudre le problème du logarithme discret en un temps polynomial [Sho94], à l'aide d'un calculateur quantique. Bien qu'il n'existe pas encore d'ordinateurs quantiques capables de résoudre les problèmes de factorisation et du logarithme discret pour les paramètres actuellement utilisés, les recherches sur le sujet se multiplient, ce qui accroît la nécessité de trouver de nouveaux cryptosystèmes résistants face à un attaquant quantique avec la technologie actuelle, nous parlons alors de cryptographie post-quantique.

Vers la transition post-quantique. Depuis l'algorithme de Shor, la cryptographie post-quantique est devenue, au fil des années, un domaine de recherche très actif. Parmi les principaux problèmes qui ne peuvent être résolus en temps polynomial à l'aide d'algorithmes quantiques, nous trouvons les protocoles basés sur les réseaux euclidiens, les codes correcteurs d'erreurs, les systèmes d'équations multivariées, les fonctions de hachage ou encore les graphes d'isogénies. En 2016, face au développement de nombreux cryptosystèmes post-quantiques, le National Institute of Standards and Technology (NIST) rendit public un rapport sur la cryptographie post-quantique dans lequel il est question des progrès dans le domaine de l'informatique quantique et donc de l'avenir de la cryptographie telle que nous la connaissons actuellement [CJL+16]. Dès lors, ils annoncèrent le lancement du projet "Post-quantum Cryptography Standardization" visant à sélectionner parmi des candidats les prochains algorithmes à clé publique pour le chiffrement, le mécanisme d'encapsulation de clé et la signature numérique, susceptibles de remplacer le cryptosystème RSA et la cryptographie à base de courbes elliptiques. Le premier tour de la standardisation, qui démarra fin novembre 2017, comptait 69 candidats et fut suivi de deux autres tours de sélection réduisant ainsi la liste à 7 candidats finalistes, en

vue d'une standardisation, et 8 candidats alternatifs sélectionnés pour un quatrième et dernier tour. Ces candidats sont répertoriés dans le Tableau 4. Suivant le calendrier public du NIST, les premières versions des standards devraient être disponibles en 2024. Cet appel à soumissions est organisé pour encourager les acteurs universitaires, industriels et gouvernementaux à collaborer. En effet, certains des schémas proposés se basent sur des problèmes mathématiques assez récents en cryptographie, de plus, la taille des différents paramètres (clé privée, clé publique, chiffré) est plus importante que celle des cryptosystèmes en cours d'utilisation. Ces deux points posent alors les principales contraintes des futures implantations. La standardisation a ainsi permis une étude approfondie de différentes solutions. Durant ce processus, l'accent est mis sur trois principaux critères : la sécurité mathématique et quantique des candidats (les candidats doivent atteindre différents niveaux de sécurité donnés dans le Tableau 3), les performances des implantations et la sécurité des implantations face aux attaques dites physiques. C'est sur ces deux derniers points que repose l'aspect pratique de ces algorithmes qui est une base essentielle pour de futures intégrations, notamment dans des composants avec peu de ressources (par exemple, en termes d'espace mémoire).

L'intégration dans des microcontrôleurs et sécurité matérielle. En raison de leur structure, les algorithmes post-quantiques requièrent un plus grand nombre de ressources que ceux actuellement implantés. La taille de certains paramètres est de l'ordre du kilooctet (ko). Ainsi, l'implantation de cryptosystèmes post-quantiques dans des environnements restreints tels que les microcontrôleurs amènent plusieurs contraintes. Un des points importants demeure l'espace mémoire requis pour l'exécution des cryptosystèmes. De nos jours, de nombreux composants disposent de moins de 10 ko de mémoire vive pour l'exécution des opérations cryptographiques. De plus, les algorithmes post-quantiques étant plus complexes du point de vue mathématique, il est également nécessaire de prendre en compte le coût d'exécution de certaines nouvelles opérations. Une fois l'implantation de ces cryptosystèmes réalisée, se pose alors la question de la sécurité matérielle de l'implantation. En effet, bien que des cryptosystèmes puissent être prouvés sûrs mathématiquement, l'implantation de ces derniers dans des circuits intégrés laissent place à un autre type d'attaques, dit attaques physiques, qui exploitent l'activité physique de l'appareil cryptographique pour récupérer des informations sur le secret [Koc96].
Ainsi, afin de préparer la transition vers de futurs standards, plusieurs points peuvent être soulevés tels que le coût mémoire d'une implantation pour une intégration dans des composants déjà commercialisés ainsi que les points vulnérables de ces nouveaux schémas et le coût de leur protection face aux attaques physiques.

Lieu et objectifs de la thèse. Les travaux de la thèse s'articulent autour des points précédemment cités. Pour ce faire, la thèse s'est réalisée au sein de l'entreprise Wisekey Semiconductors, spécialisée dans le développement de puces sécurisées pour le marché de l'Internet des Objets (IoD). L'aspect sécurité des composants est géré par l'équipe Product Security de l'entreprise qui est chargée de maintenir à jour les bibliothèques cryptographiques pour des microcontrôleurs, d'assurer la sécurité face aux attaques matérielles et de s'occuper de la certification des produits.

Afin de préparer l'implantation de futurs standards post-quantiques, il était alors nécessaire, dans un premier temps, de comprendre les briques mathématiques à partir desquelles sont construits les cryptosystèmes, d'étudier le coût mémoire des implantations et leurs performances. Dans un second temps, l'étude portait sur l'analyse des opérations vulnérables une fois implantées afin de les protéger des attaques physiques.
Dans cette thèse, nous nous sommes concentrés sur les deux principales familles de la standardisation : les réseaux euclidiens et les codes correcteurs d'erreurs. Le choix a été fait d'étudier un mécanisme d'encapsulation de clé à base de codes correcteurs d'erreurs ROLLO, un système de chiffrement à clé publique à base de réseaux euclidiens NTRU et un schéma de signature Crystals-Dilithium.

Travaux de la thèse

Plan de la thèse.
Chapitre 1 - État de l'art. Dans ce chapitre, nous détaillons les différentes notions théoriques et problèmes sur lesquels reposent les cryptosystèmes étudiés. Nous expliquons également les différentes attaques physiques existantes et explicitons le matériel utilisé dans le cadre de la thèse. Nous concluons le chapitre en donnant les motivations des choix des soumissions étudiées.

Chapitre 2 - Étude du candidat ROLLO. Dans ce chapitre, nous abordons le candidat du second tour de la standardisation ROLLO. Ce dernier se base sur des problèmes dans les codes correcteurs d'erreurs en métrique rang et utilise une structure de codes particulièrement intéressante pour une implantation dans un environnement restreint. En première partie, nous nous intéressons à l'implantation du mécanisme d'encapsulation de clé de ROLLO. En seconde partie, afin de s'assurer de la sécurité de l'implantation, nous nous concentrons sur les opérations de ROLLO, vulnérables aux attaques par canaux auxiliaires et, plus particulièrement, sur une fonction utilisée à plusieurs reprises sur des données sensibles. Finalement, nous proposons différentes contre-mesures résistantes aux attaques proposées pour différents types d'implantations.

Chapitre 3 - Étude du candidat NTRU.
Dans ce chapitre, nous présentons notre étude réalisée sur le candidat NTRU et plus particulièrement l'aspect sécurité d'une implantation de ce cryptosystème. En premier lieu, nous revenons sur les attaques existantes et les contre-mesures proposées. En second lieu, nous présentons une étude d'attaques profilées sur le produit de convolution utilisé dans la soumission NTRU. Nous proposons également des contre-mesures pour cette implantation.

Chapitre 4 - Étude du candidat Crystals-Dilithium.
Dans ce chapitre, nous décrivons dans un premier temps le schéma de signature Crystals-Dilithium et donnons des estimations des coûts mémoire qu'induit l'exécution de ce candidat. Dans un second temps, nous présentons les modifications effectuées afin de réduire le coût mémoire de l'implantation. Nous concluons par la

présentation des résultats d'une implantation avec quelques fonctions implantées en assembleur ARM pour CORTEX-M3.

Contributions scientifiques. Les travaux présentés dans le chapitre 2 ont fait l'objet de l'article intitulé *Optimized and secure implementation of ROLLO-I* accepté à la conférence CBCrypto 2020 [MBC+20].
Le code des algorithmes de la version de ROLLO avec les paramètres soumis au second tour se trouve à l'adresse `https://github.com/lmortajine/PQC`.
Un deuxième article *Exploiting ROLLO's Constant-Time Implementations with a Single-Trace Analysis* portant sur les attaques des implantations en temps constant, également présentées dans le chapitre 2, est disponible sur IACR [CMRM21].
Les résultats présentés dans le chapitre 3 sont en cours de préparation mais non encore soumis.

Standardisation du NIST. Le Tableau 4 présente les candidats sélectionnés comme finalistes de la standardisation, ainsi que les candidats choisis pour un quatrième tour, en vue d'une future standardisation.

Le Tableau 3 présente le niveau de sécurité des candidats attendu par le NIST.

Niveau	Description de la sécurité
I	aussi difficile que de casser un AES-128 (recherche exhaustive de la clé)
II	aussi difficile que de casser un SHA-256 (recherche par collision)
III	aussi difficile que de casser un AES-192
IV	aussi difficile que de casser un SHA-384
V	aussi difficile que de casser un AES-256

Table 3 – Niveau de sécurité requis pour la standardisation post-quantique du NIST.

Schéma	Fonction	Famille
Finalistes		
Classic McEliece	KEM/PKE	codes
Crystals-Dilihium	Signature	réseaux
Crystals-Kyber	KEM/PKE	réseaux
Falcon	Signature	réseaux
NTRU	KEM/PKE	réseaux
Rainbow	Signature	multivariés
Saber	KEM/PKE	réseaux
Alternatifs		
BIKE	PKE	codes
GeMSS	Signature	multivariés
HQC	KEM/PKE	codes
Frodo-KEM	KEM	réseaux
NTRU-Prime	KEM/PKE	réseaux
Picnic	Signature	crypto symétrique
SIKE	KEM/PKE	graphes d'isogénies
SPHINCS+	Signature	fonctions de hachage

Table 4 – Candidats finalistes et alternatifs de la standardisation post-quantique du NIST

Chapitre 1

État de l'art

Dans ce chapitre, nous définissons, dans un premier temps, les deux familles de problèmes mathématiques sur lesquels reposent les cryptosystèmes étudiés pendant la thèse, à savoir les codes correcteurs d'erreurs et les réseaux euclidiens. Dans un second temps, nous nous penchons sur l'aspect sécurité d'une implantation d'un cryptosystème qui est un point important lors de l'étude pratique d'un algorithme. Nous terminons par les motivations concernant les choix des candidats étudiés dans la thèse : ROLLO, NTRU et Crystals-Dilithium.

Sommaire

1.1 Mathématiques **12**
1.1.1 Introduction aux codes correcteurs d'erreurs 12
1.1.2 Introduction aux réseaux euclidiens 20
1.2 Microélectronique **28**
1.2.1 Sécurité matérielle 28
1.2.2 Matériel utilisé 36
1.3 Candidats étudiés **37**
1.3.1 ROLLO 37
1.3.2 NTRU 39
1.3.3 Crystals-Dilithium 39

1.1 Mathématiques

1.1.1 Introduction aux codes correcteurs d'erreurs

Les codes correcteurs d'erreurs ont initialement été introduits pour répondre à un besoin lié à la transmission de données sur un canal de communication [Sha48]. Ce dernier n'étant pas totalement fiable, car sujet à des perturbations pouvant venir altérer les données transmises. Dans ce cas, le canal est dit bruité. Les codes correcteurs consistent alors à ajouter de la redondance aux données émises par l'expéditeur

afin de pouvoir détecter, voire corriger les erreurs ajoutées lors de la transmission de ces données. Ainsi, comme présenté dans la Figure 1.1, pour transmettre un message sur un canal bruité, l'expéditeur E encode son message m en un autre message c appartenant à un ensemble prédéfini, appelé code. L'information est alors transmise via le canal bruité. Dans le cas où elle n'a pas été trop altérée, le récepteur R décode l'information reçue $c + e$ afin d'obtenir le message initial m.

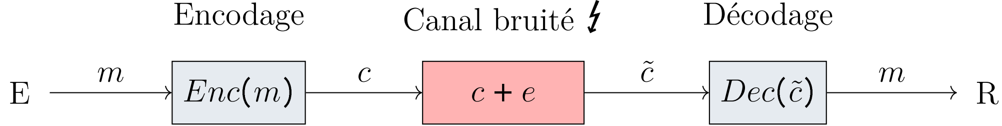


FIGURE 1.1 – Codage du canal entre un expéditeur E et un récepteur R.

Bien qu'il existe différentes structures de codes, nous nous concentrerons dans la suite sur les codes linéaires dont le décodage a été prouvé comme étant un problème NP-complet en 1978 [BMvT78], ce qui a permis à McEliece de proposer le premier cryptosystème à base de codes correcteurs, la même année [McE78]. Le cryptosystème se base alors sur l'encodage du message en un mot d'un code puis à l'ajout volontaire d'une erreur lors du chiffrement. Cette erreur est alors retrouvée par le décodage du chiffré et soustraite lors du déchiffrement afin d'obtenir le message initial.

1.1.1.1 Généralités sur les codes linéaires

Définition 1 (Code linéaire)**.** Soit $k, n \in \mathbb{N}$, un code linéaire $\mathcal{C}$ sur $\mathbb{F}_q$ est un sous-espace vectoriel de dimension k de $\mathbb{F}_q^n$. Ce code, noté $[n,k]_q$, est dit de longueur n et de dimension k. Un élément de l'ensemble $\mathcal{C}$ est appelé mot de code.

Définition 2 (Code dual)**.** Soit $\mathcal{C}$ un $[n,k]_q$-code, le code dual de $\mathcal{C}$, noté $\mathcal{C}^\perp$, est l'espace vectoriel orthogonal à $\mathcal{C}$ pour le produit scalaire usuel sur $\mathbb{F}_q^n$

$$\mathcal{C}^\perp = \{\mathbf{x} \in \mathbb{F}_q^n : \mathbf{x} \cdot \mathbf{y}^T = 0, \quad \forall \mathbf{y} \in \mathcal{C}\}.$$

Définition 3 (Matrice génératrice/Matrice de parité)**.** Un code $\mathcal{C}$ de paramètre $[n,k]_q$ peut être représenté de deux façons :
- Par un système générateur.
Soit $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_k)$ une base de $\mathcal{C}$, la matrice $\mathbf{G} \in \mathbb{F}_q^{k \times n}$, dont les lignes correspondent aux β_i pour $1 \le i \le k$, est appelée matrice génératrice du code $\mathcal{C}$, ainsi

$$\mathcal{C} = \{\mathbf{x} \cdot \mathbf{G} : \mathbf{x} \in \mathbb{F}_q^k\}.$$

- Par un système d'équations linéaires homogènes.
Une matrice génératrice du code dual $\mathcal{C}^\perp$ est appelée matrice de parité $\mathbf{H} \in \mathbb{F}_q^{(n-k) \times n}$ et ainsi

$$\mathcal{C} = \{\mathbf{x} \in \mathbb{F}_q^n : \mathbf{H} \cdot \mathbf{x}^T = \mathbf{0}_k^T\}.$$

La matrice $\mathbf{G}$ (respectivement $\mathbf{H}$) est dite sous forme systématique si et seulement si elle est de la forme $(\mathbf{I}_k|\mathbf{A})$ (respectivement $(\mathbf{I}_{n-k}|\mathbf{B})$), avec $\mathbf{A}$ (respectivement $\mathbf{B}$) une matrice quelconque dans $\mathbb{F}_q^{k \times (n-k)}$ (respectivement dans $\mathbb{F}_q^{(n-k) \times k}$).

Définition 4. (Syndrome) Soit $\mathcal{C}$ un code de paramètre $[n,k]_q$ et $\mathbf{x} \in \mathbb{F}_q^n$, le vecteur $\mathbf{s_x} = \mathbf{H} \cdot \mathbf{x}^T$ est appelé syndrome du mot $\mathbf{x}$ pour le code $\mathcal{C}$.

Les détection et correction d'erreurs par un code font appel à la notion de distance entre mots de code, les codes sont ainsi munis d'une métrique. En cryptographie, deux métriques sont principalement utilisées : la métrique de Hamming, introduite par Richard Hamming en 1950 [Ham50] et la métrique rang, introduite par Deslarte en 1978 [Del78], utilisée pour la première fois en cryptographie par Gabidulin en 1985 [Gab85].

Définition 5. (Métrique de Hamming)
• Soit $\mathbf{x} = (x_1, \cdots, x_n) \in F_q^n$, **le poids de Hamming** de l'élément $\mathbf{x}$ correspond au nombre de coordonnées non-nulles de $\mathbf{x}$. Autrement dit, le poids de Hamming est donné par

$$w_H(\mathbf{x}) = Card(\{i \in [\![1,n]\!] : x_i \neq 0\}).$$

• Soit $\mathcal{C}$ un $[n,k]_q$-code, la **distance de Hamming** entre deux mots de code $\mathbf{x}, \mathbf{y}$ de $\mathcal{C}$ correspond au nombre de coordonnées différentes entre les deux mots

$$d(\mathbf{x},\mathbf{y}) = w_H(\mathbf{x}-\mathbf{y}) = Card(\{i \in [\![1,n]\!] : x_i \neq y_i\}).$$

• Le nombre d'éléments de poids w appartenant à l'espace ambiant est donné par le binôme de Newton

$$Card(\{\mathbf{x} \in \mathbb{F}_q^n : w_H(\mathbf{x}) = w\}) = \binom{n}{w} = \frac{n!}{w!(n-w)!}.$$

Pour la métrique rang, nous considérons les codes $\mathbb{F}_{q^m}$-linéaires.

Définition 6. (Métrique rang)
• Soient $\boldsymbol{\beta} = (\beta_1, \ldots, \beta_m)$ une base de $\mathbb{F}_{q^m}$ (en tant qu'espace vectoriel) sur $\mathbb{F}_q$ et $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$. Chaque coordonnée x_i, pour $1 \geq i \geq n$, peut alors être associée à un vecteur $(x_{i,1}, \ldots, x_{i,m})$ par

$$x_i = \sum_{j=1}^{m} x_{j,i}\beta_j.$$

Un élément $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$ peut également être représenté sous forme matricielle sur $\mathbb{F}_q$

$$M(\mathbf{x}) = (x_{i,j})_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} \in \mathbb{F}_q^{n \times m}.$$

• Soit $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$, le **support** de $\mathbf{x}$ est le $\mathbb{F}_q$-sous-espace vectoriel de $\mathbb{F}_{q^m}$ engendré par les coordonnées de $\mathbf{x}$. Autrement dit, le support de $\mathbf{x}$ est défini comme

$$\mathrm{Supp}(\mathbf{x}) = \langle x_1, \cdots, x_n \rangle_{\mathbb{F}_q}.$$

• Soit $\mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{F}_{q^m}^n$, le **rang** ou **poids** de $\mathbf{x}$ peut être défini de manière analogue comme le rang de la matrice $M(\mathbf{x})$ associée au vecteur $\mathbf{x}$ ou comme la dimension de son support $\mathrm{Supp}(\mathbf{x})$. Autrement dit,

$$\|\mathbf{x}\| = Rang(M(\mathbf{x})) = dim(Supp(\mathbf{x})).$$

• Soit $\mathcal{C}$ un $[n,k]_{q^m}$-code, la distance entre deux mots de code $\mathbf{x},\mathbf{y}$ est définie par

$$d(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|.$$

• Le nombre de supports de rang w appartenant à l'espace $\mathbb{F}_{q^m}^n$ est donné par le binôme de Gauss

$$Card(\{\mathbf{x} \in \mathbb{F}_{q^m}^n : \|\mathbf{x}\| = w\}) = \binom{m}{w}_q = \prod_{i=0}^{w-1} \frac{q^m - q^i}{q^w - q^i}.$$

Considérons l'espace vectoriel K qui correspond à $\mathbb{F}_q$ en métrique de Hamming et $\mathbb{F}_{q^m}$ en métrique rang.

Définition 7 (Distance minimale)**.** La distance minimale d'un code $\mathcal{C} \subset K^n$ correspond à la plus petite distance entre deux mots de code différents

$$d_{\min} = \min_{\substack{(\mathbf{x},\mathbf{y})\in\mathcal{C}^2 \\ \mathbf{x}\neq\mathbf{y}}} d(\mathbf{x},\mathbf{y}).$$

Nous disons alors que le code $\mathcal{C}$ est de paramètre $[n,k,d_{\min}]$.

Définition 8 (Capacité de correction)**.** La capacité de correction t d'un code $\mathcal{C}$ de paramètre $[n,k,d_{\min}]$ est définie par

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

Remarque 1. Toute boule centrée sur un mot de code $\mathcal{C}$ de rayon t, la capacité de correction du code, contient un et un seul mot de code (Figure 1.2). Autrement dit,

$$\forall (\mathbf{x},\mathbf{y}) \in \mathcal{C}^2, \quad \mathbf{x} \neq \mathbf{y}, \quad \mathcal{B}(\mathbf{x},t) \cap \mathcal{B}(\mathbf{y},t) = \varnothing,$$

où $\mathcal{B}(\mathbf{z},t) = \{\mathbf{a} \in K : d(\mathbf{a},\mathbf{z}) \leq t\}$. Un mot de code se trouvant à une distance t d'un vecteur $\mathbf{x} \in K$ est l'unique mot de code le plus proche. Une erreur de poids t ajoutée à un mot de code peut alors être corrigée en cherchant le mot de code le plus proche. Un code linéaire de paramètres $[n,k,d_{\min}]$ sera donc dit t-correcteur d'erreurs et $(d_{\min}-1)$-détecteur d'erreurs.

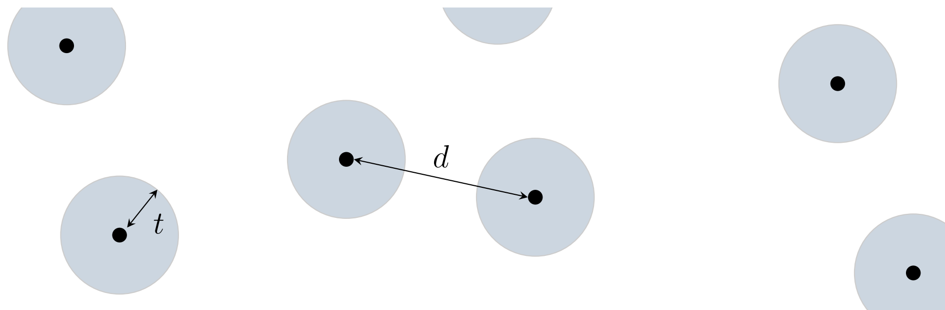


FIGURE 1.2 – Distance minimale d et capacité de correction t d'un code $\mathcal{C}$

1.1.1.2 Encodage et décodage

L'encodage est une application linéaire qui consiste à transformer un vecteur $\mathbf{x} \in \mathbb{F}_q^n$ en un mot de code.

Définition 9 (Encodage)**.** Soit $\mathcal{C}$ un code de paramètres $[n, k, d_{\min}]_q$ et de matrice génératrice G. La fonction d'encodage est définie par

$$\begin{array}{rccl} Enc: & \mathbb{F}_q^k & \to & \mathcal{C} \subseteq \mathbb{F}_q^n \\ & \mathbf{x} & \mapsto & \mathbf{c} = \mathbf{x} \cdot \mathbf{G}. \end{array}$$

Définition 10 (Décodage)**.** Soit $\mathcal{C}$ un code t-correcteur d'erreurs de paramètre $[n, k, d_{\min}]$. Nous posons, $\mathcal{C}' = \bigcap_{\mathbf{c} \in \mathcal{C}} \mathcal{B}(\mathbf{c}, t)$ et $\bar{\mathcal{C}'} = \mathbb{F}_q^n - \mathcal{C}'$. La fonction de décodage est alors définie par

$$Dec(\mathbf{x}) = \begin{cases} \mathbf{c} & \text{si } \mathbf{x} \in \mathcal{C}' \\ \text{Echec} & \text{si } \mathbf{x} \in \bar{\mathcal{C}'} \end{cases}.$$

Remarque 2. Afin de corriger un mot de code, il est nécessaire de connaître la structure du code utilisé afin de le décoder avec l'algorithme adéquat.

Les candidats de la standardisation post-quantique du NIST se basent sur différents codes utilisant les deux métriques (Table 1.2). Cependant, dans la suite, nous détaillons uniquement les codes LRPC (Low Rank Parity Check) en métrique rang car ils sont utilisés dans le cryptosystème étudié ROLLO.

1.1.1.3 Codes LRPC et la structure de code utilisé

Définition 11 (Code LRPC)**.** Un code LRPC $\mathcal{C}$ de paramètre $[n, k]_{q^m}$ et de rang d est un code linéaire sur $\mathbb{F}_{q^m}$, de longueur n et de dimension k, qui admet une matrice de parité $\mathbf{H} = (h_{i,j})_{\substack{0 \le i < n \\ 0 \le j < n-k}} \in \mathbb{F}_{q^m}^{n \times (n-k)}$, tel que l'espace vectoriel

$$F = \langle h_{i,j} \rangle_{\mathbb{F}_q} \tag{1.1}$$

est de dimension d.

Remarque 3. Dans un code LRPC de rang d, la matrice de parité $\mathbf{H}$ du code est de rang maximal d et de support F.

Remarque 4. La représentation d'un $[n, k]_{q^m}$-code nécessite $k \times (n-k)$ éléments dans $\mathbb{F}_{q^m}$, soit $k \times (n-k) \times m \times \lceil \log_2(q) \rceil$ bits. Ainsi, afin de réduire la représentation des codes, la structure des codes LRPC utilisés dans les cryptosystèmes ROLLO se base sur celle des $[2n, n]_{q^m}$-codes idéaux, donnant ainsi les codes LRPC idéaux.

Avant de définir les codes LRPC idéaux, il est nécessaire de noter qu'il existe un isomorphisme entre l'espace vectoriel $\mathbb{F}_{q^m}^n$ et le corps $\mathbb{F}_{q^m}[Z]/(P_n)$ défini par

$$\begin{array}{rccl} \phi: & \mathbb{F}_{q^m}^n & \to & \mathbb{F}_{q^m}[Z]/(P_n) \\ & (x_0, \ldots, x_{n-1}) & \mapsto & \sum_{i=0}^{n-1} x_i Z^i \end{array},$$

où P_n est un polynôme irréductible de degré n dans $\mathbb{F}_q$ et (P_n) est l'idéal de $\mathbb{F}_{q^m}[Z]$ engendré par P_n.

Définition 12 (Matrice idéale)**.** Soient $P_n \in \mathbb{F}_q[Z]$ un polynôme irréductible de degré n et $\mathbf{x} \in \mathbb{F}_{q^m}^n$ un vecteur de longueur n. Une matrice idéale générée à partir de $\mathbf{x}$ est une matrice carrée de taille n définie par

$$\mathcal{IM}(\mathbf{x}) = \begin{pmatrix} \mathbf{x}(Z) \\ Z \cdot \mathbf{x}(Z) \mod P_n \\ \vdots \\ Z^{n-1} \cdot \mathbf{x}(Z) \mod P_n \end{pmatrix}.$$

La matrice idéale est donc définie par sa première ligne.

Remarque 5. Nous pouvons faire correspondre le produit de deux vecteurs $\mathbf{x}, \mathbf{y} \in \mathbb{F}_{q^m}^n$, $\mathbf{x} \cdot \mathbf{y}$ avec le produit polynomial $\phi(\mathbf{x}) \cdot \phi(\mathbf{y}) = \mathbf{x}(Z) \cdot \mathbf{y}(Z) \mod P_n$. Ce dernier est équivalent à un produit vecteur/matrice donné par

$$\begin{aligned} \mathbf{x}(Z) \cdot \mathbf{y}(Z) \mod P_n &= \sum_{i=0}^{n-1} x_i Z^i \cdot \mathbf{y}(Z) \mod P_n \\ &= \sum_{i=0}^{n-1} x_i (Z^i \cdot \mathbf{y}(Z)) \mod P_n \\ &= (x_0, \dots, x_{n-1}) \cdot \mathcal{IM}(\mathbf{y}). \end{aligned}$$

Définition 13 (Code idéal)**.** Soient $P_n \in \mathbb{F}_q[Z]$ un polynôme irréductible de degré n et $(\mathbf{g}_1, \mathbf{g}_2) \in \mathbb{F}_{q^m}^n \times \mathbb{F}_{q^m}^n$ deux vecteurs. Un code $\mathcal{C}$ est un $[2n, n]_{q^m}$-code idéal associé à $\mathbf{g}_1, \mathbf{g}_2$ s'il admet une matrice génératrice

$$\mathbf{G} = \left(\mathcal{IM}(\mathbf{g}_1) \;\middle|\; \mathcal{IM}(\mathbf{g}_2) \right).$$

Remarque 6. D'après cette définition, le code $\mathcal{C}$ est défini par $\mathcal{C} = \{(\mathbf{a}\mathbf{g}_1, \mathbf{a}\mathbf{g}_2), \mathbf{a} \in \mathbb{F}_{q^m}^n\}$. Afin de réduire la représentation du code, l'inversibilité de $\mathbf{g}_1$ est souvent requise permettant ainsi d'obtenir le code $\mathcal{C} = \{(\mathbf{a}, \mathbf{a}\mathbf{g}), \mathbf{a} \in \mathbb{F}_{q^m}^n\}$ avec $\mathbf{g} = \mathbf{g}_1^{-1} \cdot \mathbf{g}_2$. Un $[2n, n]_{q^m}$-code $\mathcal{C}$ peut donc être représenté par $n \times m \times \lceil \log_2(q) \rceil$ bits.

Définition 14. (Code LRPC idéal).
Soit F un $\mathbb{F}_q$-sous-espace de $\mathbb{F}_{q^m}$ tel que $dim(F) = r$. Soient $\mathbf{h}_1, \mathbf{h}_2 \in \mathbb{F}_{q^m}^n$ tels que $\text{Supp}(\mathbf{h}_1, \mathbf{h}_2) = F$, et $P_n \in \mathbb{F}_q[X]$ un polynôme de degré n. Notons

$$\mathbf{H}_1 = \begin{pmatrix} \mathbf{x}(Z) \\ Z \cdot \mathbf{x}(Z) \mod P_n \\ \vdots \\ Z^{n-1} \cdot \mathbf{x}(Z) \mod P_n \end{pmatrix}^T \quad \text{et} \quad \mathbf{H}_2 = \begin{pmatrix} \mathbf{y}(Z) \\ Z \cdot \mathbf{y}(Z) \mod P_n \\ \vdots \\ Z^{n-1} \cdot \mathbf{y}(Z) \mod P_n \end{pmatrix}^T.$$

Un code LRPC idéal de paramètre $[2n, n]_{q^m}$ est un code qui admet une matrice de parité $\mathbf{H} = \begin{pmatrix} \mathbf{H}_1 & | & \mathbf{H}_2 \end{pmatrix}$ de rang n.

En cryptographie, nous cherchons à protéger le message encodé et le rendre ainsi seulement accessible au destinataire. Pour cela, nous devons cacher la structure du code utilisé. Les cryptosystèmes de la soumission ROLLO reposent leur sécurité sur des problèmes dans les codes correcteurs d'erreurs en métrique rang. Ces problèmes sont explicités dans la section suivante.

1.1.1.4 Problèmes difficiles liés aux codes correcteurs

Nous présentons dans cette section trois problèmes difficiles pour les codes idéaux : le problème du décodage par syndrome, le problème de recherche du support et l'indistinguabilité des codes LRPC idéaux. Ces problèmes permettent alors de construire des cryptosystèmes prouvés sûrs sémantiquement. La construction d'un cryptosystème basé sur les codes LRPC repose également sur l'algorithme de décodage de ces derniers.

Problème 1 (I-RSD - Ideal-Rank Syndrome Decoding)**.** Soient $P_n \in \mathbb{F}_q[X]$ un polynôme de degré n, un vecteur $\mathbf{h} \in \mathbb{F}_{q^m}^n$ et un syndrome $\mathbf{s}$ de poids w.
Le problème I-RSD consiste à trouver, s'il existe, un vecteur $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2) \in \mathbb{F}_{q^m}^{2n}$ de rang inférieur à w tel que

$$\mathbf{x}_1 + \mathbf{x}_2 \cdot \mathbf{h} = \mathbf{s} \mod P_n.$$

Remarque 7. Il a été démontré dans [AGH+19] que trouver un vecteur $\mathbf{x} \in \mathbb{F}_{q^m}^{2n}$ solution d'une instance de I-RSD est équivalent à retrouver le support de $\mathbf{x}$, c'est ainsi que le problème de recherche du support a été introduit.

Problème 2 (I-RSR - Ideal-Rank Support Recovery)**.** Soient $P_n \in \mathbb{F}_q[X]$ un polynôme de degré n, un vecteur $\mathbf{h} \in \mathbb{F}_{q^m}^n$ et un syndrome $\mathbf{s}$ de poids w.
Le problème I-RSR consiste à retrouver le support E de $\mathbf{e}_1$ et $\mathbf{e}_2$ de poids inférieur à w tel que

$$\mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{h} = \mathbf{s} \mod P_n.$$

De plus, un code LRPC idéal peut facilement être caché et réduit en ne dévoilant que la forme systématique de sa matrice de parité, rendant alors sa structure algébrique trop "faible" pour la retrouver. Les cryptosystèmes ROLLO se basent alors sur un deuxième problème difficile appelé "indistinguabilité des codes LRPC idéaux".

Problème 3 (Indistinguabilité des codes LRPC idéaux)**.** Soient $P_n \in \mathbb{F}_q[X]$ un polynôme de degré n et $\mathbf{h} \in \mathbb{F}_{q^m}^n$ un vecteur. Il est considéré difficile de distinguer un code aléatoire d'un code LRPC, de matrice de parité générée par $\mathbf{h}$ et P_n, et de poids d. Autrement dit, il est difficile de déterminer si $\mathbf{h}$ provient d'un tirage aléatoire ou si $\mathbf{h} = \mathbf{x}^{-1} \cdot \mathbf{y}$ avec $\mathbf{x}$ et $\mathbf{y}$ des vecteurs de même support de dimension d.

1.1.1.5 Décodage des codes LRPC

L'algorithme de décodage initial des codes LRPC a été introduit dans [GMRZ13a] et permettait de retrouver les coordonnées de l'erreur $\mathbf{e} \in \mathbb{F}_{q^m}^n$ ajoutée au message encodé lors du chiffrement à partir du syndrome $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{F}_{q^m}^n$. L'algorithme de décodage utilise également la connaissance de l'espace vectoriel de petite dimension F défini à l'équation 1.1. Puisque les problèmes de décodage par syndrome (I-RSD) et de recherche du support (I-RSR) sont équivalents, les cryptosystèmes de la soumission ROLLO se basent sur un algorithme de recherche du support de l'erreur (Rank Support Recovery - RSR) qui est alors considéré comme le secret partagé. L'algorithme RSR, présenté dans l'Algorithme 1 se base sur celui proposé initialement dans [GMRZ13a] et ne tient pas compte de la partie recouvrement des coordonnées de l'erreur.
Explication de l'algorithme de décodage : soient $F = \langle f_1, \ldots, f_n \rangle_{\mathbb{F}_q}$ un espace vectoriel de dimension d et $E = \langle e_1, \ldots, e_n \rangle_{\mathbb{F}_q}$ un espace vectoriel de dimension r. Soit $\mathbf{H} \in \mathbb{F}_{q^m}^{2n}$ (défini dans la Définition 14) une matrice de support F et $\mathbf{e}$ une erreur de support E. Considérons $\mathcal{C}$ le code LRPC de matrice de parité $\mathbf{H}$ et le syndrome de l'erreur $\mathbf{s} = \mathbf{H} \cdot \mathbf{e}^T$. L'idée générale de l'algorithme RSR se base sur le calcul du support du syndrome $S = \langle s_1, \ldots s_n \rangle$ qui est un sous-espace vectoriel de $EF = \langle \{e_i f_j, e_i \in E, f_j \in F\} \rangle_{\mathbb{F}_q}$. L'espace vectoriel EF étant au plus de dimension rd, nous avons $dim(S) \leq rd$. Ainsi, le calcul de l'intersection des $S_i = f_i^{-1} \cdot S$ pour $1 \leq i \leq d$ nous donne le support de l'erreur E.
L'algorithme RSR est probabiliste, il se peut qu'il échoue à retrouver le support E, notamment dans le cas où l'une des deux assertions suivantes est vérifiée :

- $dim(S) < dim(EF)$,
- $dim(\bigcap_{i=1}^{d} f_i^{-1} \cdot S) > r$.

La probabilité que l'un de ces deux évènements se produise, et donc la probabilité d'échec de l'algorithme, est détaillée dans [AMAB+20a].

Algorithme 1 : Algorithme RSR [AMAB+20a]

Entrées : $F = \langle f_1, \cdots, f_d \rangle_{\mathbb{F}_q}$ un $\mathbb{F}_q$-sous-espace de $\mathbb{F}_{2^m}$,
$s = (s_1, \cdots, s_n) \in \mathbb{F}_{2^m}^n$ le syndrome d'une erreur e et r le rang de e

Résultat : Le support E de l'erreur e

1 Calculer $S = \langle s_1, \cdots, s_n \rangle_{\mathbb{F}_q}$
2 Pré-calculer chaque $S_i = f_i^{-1} \cdot S$ pour $i = 1$ à d
3 $E \leftarrow \bigcap_{1 \leq i \leq d} S_i$
4 **retourner** E

La soumission ROLLO a fait l'objet de différentes modifications durant la standardisation du NIST. L'Algorithme 1 fait notamment partie de la toute dernière version soumise à la fin du second tour. Dans la première version de la soumission ROLLO [AMAB+19], une partie basée sur l'amélioration de l'algorithme RSR

détaillée dans [AGH+19] a été ajoutée à l'Algorithme 1 permettant de retrouver l'espace vectoriel EF dans le cas où $dim(S) < rd$. Cette amélioration est présentée dans l'Algorithme 2.

Algorithme 2 : Deuxième version de l'algorithme RSR [AMAB+19]

Entrées : $F = \langle f_1, \cdots, f_d \rangle$ un $\mathbb{F}_q$-sous-espace de $\mathbb{F}_{2^m}$, $s = (s_1, \cdots, s_n) \in \mathbb{F}_{2^m}^n$ le syndrome d'une erreur e et r le rang de e

Résultat : Le sous-espace vectoriel E

1. Calculer $S = \langle s_1, \cdots, s_n \rangle_{\mathbb{F}_q}$
2. Pré-calculer chaque S_i pour $i = 1$ à d
3. Pré-calculer chaque $S_{i,i+1} = S_i \cap S_{i+1}$ pour $i = 1$ à $d - 1$
4. **pour** *i allant de* 1 *à* $d - 2$ **faire**
   - `// Somme directe entre espaces vectoriels`
5. $tmp \leftarrow S + F \cdot (S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})$
6. **si** $\dim(tmp) \leq rd$ **alors**
7. $S \leftarrow tmp$;
8. $E \leftarrow \bigcap_{1 \leq i \leq d} f_i^{-1} \cdot S$
9. **retourner** E

Démonstration dans le cas général [AMAB+20a]
L'algorithme RSR retourne le bon résultat si $E = S_1 \cap \ldots \cap S_d$.
Supposons que $dim(EF) = dim(S) = rd$ et $dim(S_1 \cap \ldots \cap S_d) = r$. Par hypothèse, $f_i e_j \in S$ pour tout $i \in [\![1, d]\!]$ et $j \in [\![1, r]\!]$, donc $e_j \in S_i = f_i^{-1} S$ pour tout i, j. Ainsi, $E \subset S_i$ pour tout $i \in [\![1, d]\!]$ et donc $E \subset S_1 \cap \ldots \cap S_d$. De plus $dim(E) = dim(S_1 \cap \ldots \cap S_d)$, nous en déduisons que $E = S_1 \cap \ldots \cap S_d$.

1.1.2 Introduction aux réseaux euclidiens

Les réseaux euclidiens sont des ensembles périodiques de points de l'espace dont l'étude purement mathématique remonte aux 18e et 19e siècles avec les travaux de Lagrange, Gauss puis Hermite qui portaient sur l'étude des formes quadratiques. À la fin du 19e siècle, Minkowski introduit alors la géométrie des nombres permettant de donner une autre interprétation à ces objets mathématiques. Il faudra attendre la fin du 20e siècle pour que les réseaux euclidiens connaissent un essor considérable en algorithmique avec l'article [LLL82] dans lequel Lenstra, Lenstra et Lovász décrivent un algorithme de réduction de réseaux, connu sous le nom de LLL et utilisé dans de nombreuses applications comme celles que l'on trouve en cryptanalyse et en cryptographie. La cryptographie basée sur les réseaux euclidiens fut initiée par Ajtai en 1996 [Ajt96] qui a prouvé la complexité de problèmes dans les réseaux euclidiens et a ainsi permis l'utilisation de ces derniers pour la construction de cryptosystèmes ou encore de débloquer certaines fonctionnalités tel que le chiffrement complètement homomorphe [Gen09].

1.1.2.1 Généralités sur les réseaux euclidiens

Nous considérons l'espace vectoriel euclidien $\mathbb{R}^n$ muni de son produit scalaire usuel.

Définition 15 (Réseau euclidien)**.** Un réseau $\mathcal{L}$ est un sous-groupe discret de $\mathbb{R}^n$.

Remarque 8. Un tel sous-groupe est nécessairement isomorphe au groupe abélien $\mathbb{Z}^d$ (avec $d \leqslant n$). L'entier d est appelé dimension du réseau.

Proposition 1. Un sous-ensemble $\mathcal{L}$ est un réseau de dimension d si et seulement si il existe $\mathbf{b}_0, \cdots, \mathbf{b}_{d-1} \in \mathbb{R}^n$ tels que :

$$\mathcal{L} = \mathbb{Z}\mathbf{b}_0 + \cdots + \mathbb{Z}\mathbf{b}_{d-1}.$$

Ainsi, si $B = \{\mathbf{b}_0, \cdots, \mathbf{b}_{d-1}\}$ est une base de $\mathcal{L}$, nous notons

$$\mathcal{L} = \{\mathbf{a}.\mathbf{B} |\ \mathbf{a} \in \mathbb{Z}^d\},$$

où $\mathbf{B} \in \mathbb{R}^{d\times n}$ est une matrice de rang d.
Nous notons $\mathcal{L} = \mathcal{L}(B)$.

Exemple 1. La Figure 1.3 présente un réseau euclidien de dimension 2 avec deux bases.

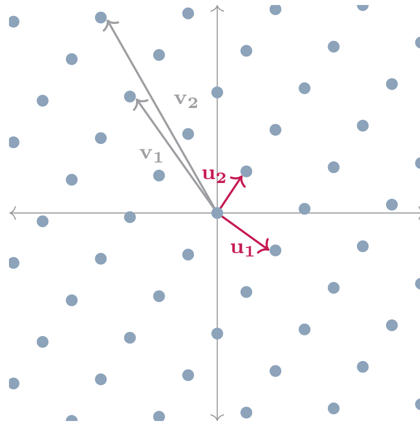


FIGURE 1.3 – Réseau euclidien $\mathcal{L}$ sur $\mathbb{R}^2$ avec $\{\mathbf{u}_1, \mathbf{u}_2\}$ et $\{\mathbf{v}_1, \mathbf{v}_2\}$ deux bases de $\mathcal{L}$.

Un réseau $\mathcal{L}$ de dimension $n \geqslant 2$ admet une infinité de bases toutes équivalentes d'un point de vue algébrique.

Définition 16. Soit $\mathcal{L}$ un réseau euclidien et B, B' deux systèmes de vecteurs de $\mathbb{R}^n$, alors $\mathcal{L}(B) = \mathcal{L}(B')$, si et seulement si il existe une matrice unimodulaire[1] $\mathbf{U}$ telle que $B = B'\mathbf{U}$.

Exemple 2. En considérant les bases de l'*Exemple* 1, $\{\mathbf{u}_1, \mathbf{u}_2\}$ et $\{\mathbf{v}_1, \mathbf{v}_2\}$, nous avons $\mathbf{v}_1 = -2\mathbf{u}_1 + \mathbf{u}_2$ et $\mathbf{v}_2 = -3\mathbf{u}_1 + 2\mathbf{u}_2$. Ainsi,

$$\begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 \end{pmatrix} \times \begin{pmatrix} -2 & -3 \\ 1 & 2 \end{pmatrix},$$

1. une matrice carrée à coefficients entiers dont le déterminant vaut ± 1.

avec $det(U) = \begin{vmatrix} -2 & -3 \\ 1 & 2 \end{vmatrix} = -1.$

Remarque 9. Soit σ une permutation de d éléments et B une base du réseau euclidien $\mathcal{L}$ de dimension d, alors $B = \{\mathbf{b}_{\sigma(1)}, \ldots, \mathbf{b}_{\sigma(d)}\}$ est une base du réseau $\mathcal{L}$.
Ceci nous permet de considérer, sans perte de généralité, une base ordonnée du réseau $\mathcal{L}$:

$$\|\mathbf{b}_1\| \leq \ldots \leq \|\mathbf{b}_d\|.$$

Définition 17 (Déterminant)**.** Soit $B \in \mathbb{R}^{n \times d}$ une base d'un réseau $\mathcal{L}$. Le déterminant du réseau $det(\mathcal{L}(B))$ est défini par

$$det(\mathcal{L}(B)) = \sqrt{det(BB^T)}.$$

Définition 18 (Distance minimale)**.** Pour un réseau $\mathcal{L}$, la distance minimale de $\mathcal{L}$ correspond à la plus petite distance entre deux points du réseau, autrement dit, la distance minimale est donnée par

$$\lambda_1(\mathcal{L}) = \min\{\|\mathbf{u} - \mathbf{v}\| \mid \mathbf{u}, \mathbf{v} \in \mathcal{L}, \mathbf{u} \neq \mathbf{v}\}.$$

Proposition 2. Soit $\mathcal{L}$ un réseau euclidien, nous avons alors

$$\lambda_1(\mathcal{L}) = \min\{\|\mathbf{v}\| \mid \mathbf{v} \in \mathcal{L} \setminus \{0\}\}.$$

Un vecteur $\mathbf{v} \in \mathcal{L}$ tel que

$$\|\mathbf{v}\| = \lambda_1(\mathcal{L})$$

est dit le vecteur le plus court de $\mathcal{L}$.

Remarque 10. Il est clair que si B est une base orthogonale de $\mathcal{L}$, alors un plus court vecteur de B est un plus court vecteur de $\mathcal{L}$.

Définition 19 (Défaut d'orthogonalité)**.** Soit $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ une base de $\mathcal{L}$. Le défaut d'orthogonalité est défini par :

$$\rho(B) = \frac{\prod_{i=1}^{d} \|\mathbf{b}_i\|^2}{det(\mathcal{L}(B))^2}.$$

Si B est orthogonale, alors $\rho(B) = 1$.

Définition 20. (Distance au réseau) Soit $\mathcal{L}$ un réseau euclidien, soit $\mathbf{w} \in \mathbb{R}^n$, la distance entre $\mathbf{w}$ et $\mathcal{L}$ est définie par

$$dist(\mathbf{w}, \mathcal{L}) = \min_{\mathbf{v} \in \mathcal{L}} \|\mathbf{w} - \mathbf{v}\|.$$

Définition 21. (Vecteur le plus proche) Soit $\mathcal{L}$ un réseau euclidien, soit $\mathbf{w} \in \mathbb{R}^n$, un vecteur $\mathbf{v} \in \mathcal{L}$ est le plus proche de $\mathbf{w}$ si

$$dist(\mathbf{w}, \mathcal{L}) = \|\mathbf{w} - \mathbf{v}\|.$$

1.1.2.2 Problèmes difficiles liés aux réseaux euclidiens

Nous pouvons classifier les problèmes difficiles sur les réseaux euclidiens en deux catégories :
- Ceux dans le pire des cas : ces problèmes sont directement liés à une base B arbitraire d'un réseau $\mathcal{L}$ et ont été prouvés NP-difficiles.
- Ceux dans le cas moyen : ces problèmes sont liés à une base B choisie selon une distribution donnée.
Les travaux d'Ajtai [Ajt96] et de Regev [Reg05] ont permis d'établir une connexion entre ces deux catégories de problèmes en démontrant que certains problèmes dans le cas moyen étaient aussi difficiles à résoudre que les problèmes dans le pire des cas. Cette réduction favorisera alors le développement de cryptosystèmes basés sur des problèmes dans le cas moyen, plus performants et efficaces que ceux à base de problèmes dans le pire des cas. Dans cette section, nous présentons quelques problèmes dans le pire des cas et des problèmes dans le cas moyen sur lesquels reposent la plupart des cryptosystèmes à base de réseaux euclidiens actuels.

• **Problème dits dans le pire des cas**
Les deux problèmes fondamentaux sont la recherche d'un plus court vecteur dans un réseau et la recherche du vecteur le plus proche.

Problème 4 (SVP - Shortest Vector Problem)**.** Étant donnée une base B arbitraire d'un réseau $\mathcal{L}$, le problème SVP consiste à trouver un plus court vecteur non nul du réseau $\mathcal{L}$.

Problème 5 (CVP - Closest Vector Problem)**.** Soit un vecteur $\mathbf{w} \in \mathbb{R}^n$ qui n'est pas dans $\mathcal{L}$, le problème CVP consiste à trouver un vecteur le plus proche de $\mathbf{w}$.

La Figure 1.4 illustre les problèmes SVP et CVP en dimension 2.

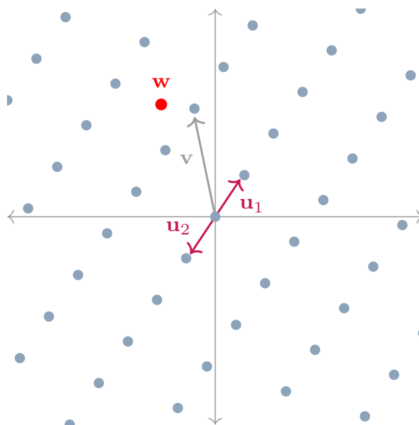


FIGURE 1.4 – Réseau $\mathcal{L}$ sur $\mathbb{R}^2$ où $\mathbf{u}_1$ et $\mathbf{u}_2$ sont les vecteurs les plus courts et $\mathbf{v}$ le vecteur le plus proche de $\mathbf{w} \in \mathbb{R}^2$.

Remarque 11. Il existe également des problèmes d'approximation qui appartiennent à différentes classes de complexité en fonction du facteur d'approximation donné.

Une des généralisations du problème SVP (Approximate Shortest Vector Problem - SVP_γ) consiste à trouver une approximation d'un vecteur le plus court à un facteur γ [2] près. Autrement dit, étant donné une base B d'un réseau $\mathcal{L}$ de dimension n, trouver un vecteur non nul $\mathbf{v} \in \mathcal{L}$ tel que

$$\|\mathbf{v}\| \leqslant \gamma(n) \cdot \lambda_1(\mathcal{L}).$$

Il va de soi que lorsque $\gamma(n) = 1$, cela nous amène à résoudre le problème SVP. Le problème SVP_γ a été prouvé NP-difficile pour $\gamma = 2^{\log^{1/2-\epsilon} n}$.
De manière similaire, il existe des variantes du problème CVP, l'une d'entre elles consiste à trouver le vecteur le plus proche d'un élément dans $\mathbb{R}^n$ se trouvant à une certaine distance du réseau.

Définition 22 (BDD_α - Bounded Distance Decoding Problem)**.** Soient $\mathcal{L}$ réseau et $\mathbf{w} \in \mathbb{R}^n$ un vecteur tel que $dist(\mathbf{w}, \mathcal{L}) < \alpha \cdot \lambda_1(\mathcal{L})$, le problème BDD consiste à trouver un vecteur de $\mathcal{L}$ le plus proche de $\mathbf{w}$.

Ce dernier problème a été prouvé NP-difficile pour $\alpha < 1/\sqrt{2}$.

Remarque 12. Bien que ces problèmes aient été utilisés dans la construction de cryptosystèmes tels que GGH [GGH97] et Ajtai-Dwork [AD97], ces derniers ont été cassés en pratique. Malgré un haut niveau de sécurité, les cryptosystèmes basés sur les problèmes dans les réseaux sont généralement inefficaces, principalement en raison de la taille des clés qu'ils requièrent. Pour cette raison, les protocoles utilisés dans la pratique reposent sur des réseaux idéaux et cycliques.

• **Problèmes dits dans le cas moyen**
Les constructions de systèmes cryptographiques reposent alors sur cette deuxième classe de problèmes. La complexité de certains d'entre eux est due à la distribution choisie.
Deux problèmes, introduits par Ajtai et Regev, furent alors fondateurs pour le développement de la cryptographie à base de réseaux euclidiens.

Problème 6 (SIS - Short Integer Solution)**.** Soit $A \in \mathbb{Z}_q^{n\times m}$ une matrice $n \times m$ qui consiste en m vecteurs aléatoires $\mathbf{a}_1, \cdots, \mathbf{a}_m \in \mathbb{Z}_q^n$. Le problème consiste à trouver un vecteur $\mathbf{z} \in \mathbb{Z}^m$ tel que

$$A\mathbf{z} = \mathbf{0} \mod q.$$

Cela revient à trouver le plus court vecteur dans le réseau

$$\mathcal{L}^\perp(A) = \{\mathbf{z} \in \mathbb{Z}^m | A\mathbf{z} = \mathbf{0} \mod q\}$$

Problème 7 (LWE - Learning With Errors)**.** Soit deux entiers $n \geqslant 1$, $q \geqslant 2$ et soit χ une loi de probabilité définie sur $\mathbb{Z}_q$.
Soit $A_{\mathbf{s},\chi}$ la loi de probabilité obtenue en choisissant un vecteur $\mathbf{a} \in \mathbb{Z}_q^n$ uniformément aléatoire, $e \in \mathbb{Z}_q$ suivant la loi χ et retourne $(\mathbf{a}, \langle \mathbf{a}, \mathbf{s} \rangle + e)$ [3], pour un vecteur fixe $\mathbf{s} \in \mathbb{Z}_q^n$.
Étant donné $A_{\mathbf{s},\chi}$, le problème LWE consiste à trouver le secret $\mathbf{s}$.

2. γ étant une fonction de la dimension du réseau $\gamma = \gamma(n)$
3. Les additions sont faites modulo q.

Remarque 13. Le problème LWE est équivalent au problème BDD dans les réseaux. En effet, en considérant le réseau

$$\mathcal{L}(A) = \{\mathbf{z} \in \mathbb{Z}^m | \mathbf{z}^T = \mathbf{s}^T A \mod q;\ \mathbf{s} \in \mathbb{Z}_q^n\} = \pi^{-1}(Im(A)),$$

où π est la surjection canonique de $\mathbb{Z}^m$ dans $\mathbb{Z}_q^m$. Nous pouvons alors trouver $\mathbf{z}^t$ en résolvant le problème BDD dans le réseau $\mathcal{L}(A)$ puis résoudre l'équation

$$\mathbf{z}^T = \mathbf{s}^T A \mod q$$

afin de trouver $\mathbf{s}$.

Variants des problèmes SIS et LWE. Afin de construire des cyptosystèmes plus performants, des variants des deux précédents problèmes sur les anneaux de polynômes ont été introduits. Nous décrivons alors l'anneau de polynômes principalement utilisé en cryptographie. Soit $n, q \geq 0$ deux entiers. L'anneau des polynômes (de rang n, modulo q) est l'anneau quotient

$$\mathcal{R}_q = \mathbb{Z}_q[x]/(x^n - 1), \tag{1.2}$$

où $\mathbb{Z}_q = \mathbb{Z}/q\mathbb{Z}$.
Un polynôme $f \in \mathcal{R}_q$ sera de la forme

$$f(x) = f_0 + f_1 x + \cdots + f_{n-1}x^{n-1},$$

avec $f_0, \cdots, f_{n-1} \in \mathbb{Z}_q$.
Le polynôme f est ainsi représenté par le vecteur

$$\mathbf{f} = (f_0, \cdots, f_{n-1}).$$

L'addition de deux polynômes $f, g \in \mathcal{R}_q$ correspond à l'addition de deux vecteurs associés à $f(x)$ et $g(x)$.
Considérons

$$f(x) = p_0 + p_1 x + \cdots + p_{n-1}x^{n-1} \quad \text{et} \quad g(x) = q_0 + q_1 x + \cdots + q_{n-1}x^{n-1}.$$

La somme $f(x) + g(x)$ est identifiée par le vecteur

$$\mathbf{f} + \mathbf{g} = (f_0 + g_0, \cdots, f_{n-1} + g_{n-1}).$$

La multiplication de deux polynômes $f, g \in \mathcal{R}_q$ est définie par

$$f(x) \star g(x) = h(x) \text{ avec } h_k = \sum_{i+j \equiv k \mod n} f_i g_{k-i} \mod q \tag{1.3}$$

Nous définissons alors le produit de convolution sous forme vectorielle par

$$(f_0, \cdots, f_{n-1}) \star (g_0, \cdots, g_{n-1}) = (h_0, \cdots, h_{n-1}).$$

Les opérations sur $\mathcal{R}_q$ sont considérées "centrées", par exemple $a \mod q$ retourne un entier compris entre $-q/2$ et $q/2$.

Nous pouvons présenter les problèmes sur lesquels se basent le cryptosystème NTRU et le schéma de signature Crystals-Dilithium, tous deux étudiés durant cette thèse.

En 1996, Höstein, Pipher et Silverman [HPS98] proposèrent un nouvel algorithme de chiffrement basé sur un problème, aujourd'hui appelé le problème NTRU et ils choisirent pour domaine de calcul non pas directement un réseau euclidien mais l'anneau de polynôme $\mathcal{R}_q$.

Problème 8 (NTRU)**.** Soient n, q deux nombres premiers définissant $\mathcal{R}_q$ (Équation 1.2) et χ une loi de probabilité sur $\mathcal{R}_q$. Soit $f(x), g(x)$ deux polynômes échantillonnés à partir de χ. Étant donné $h(x) = g(x)/f(x) \mod q$, le problème NTRU consiste à trouver le polynôme g ou f.

Remarque 14. Le problème NTRU est considéré comme difficile depuis son introduction et peut se ramener au problème du plus court vecteur dans un réseau. En effet, étant donné $\mathbf{h}$ défini dans le problème 8, il existe un polynôme $u \in \mathcal{R}$ tel que

$$f(x) \star h(x) - qu(x) = g(x).$$

Nous pouvons alors former le réseau $\mathcal{L}$ suivant

$$\mathcal{L} = \{(\mathbf{f}, \mathbf{g}) \in \mathcal{R}^2 : \mathbf{f} \star \mathbf{h} - q\mathbf{u} = \mathbf{g}\}.$$

$\mathcal{L}$ est le réseau NTRU associé à $\mathbf{h}$, on le note $\mathcal{L}_{\mathbf{h}}^{\mathrm{NTRU}}$.

Problème 9 (MLWE - Search Module Learning With Errors)**.** Le problème MLWE a été introduit en 2012 [LS14] et il est une généralisation du problème LWE (Problème 7). Étant donné un secret $\mathbf{s} \in \mathcal{R}_q^k$ et des échantillons $(\mathbf{a}_i, b_i = \langle \mathbf{a}_i, \mathbf{s} \rangle + e_i)$ avec $a_i \in \mathcal{R}_q^k$ choisi aléatoirement suivant la loi uniforme et e_i choisi suivant la loi χ, le problème MLWE consiste à trouver le secret $\mathbf{s}$.

Remarque 15. Dans le cas où $k = 1$, cela nous ramène au problème RLWE (Ring-LWE) introduit dans [SSTX09, LPR10].

Problème 10 (MSIS - Module Short Integer Solution)**.** De manière similaire, le problème SIS (Problème 6) se généralise en un problème sur les anneaux/modules. Soit χ la distribution uniforme et β un paramètre du problème. Soient n, q deux entiers premiers définissant $\mathcal{R}_q$ et m, k deux entiers positifs. Étant donné $a_i \in \mathcal{R}_q^k$ pour $1 \leq i \leq m$ choisi aléatoirement suivant χ, le problème MSIS consiste à trouver $z_i \in \mathcal{R}$ pour $1 \leq i \leq m$ tels que

- $< \mathbf{a}, \mathbf{z} >= 0$
- $||\mathbf{z}|| < \beta$ avec $\mathbf{z} = (z_1, \ldots, z_m) \in \mathcal{R}^m$.

Remarque 16. Dans le cas où $k = 1$, cela nous amène à résoudre le problème Ring-SIS introduit dans [Mic07].

1.1.2.3 Résolution des problèmes sur les réseaux

La complexité des problèmes difficiles sur les réseaux euclidiens reposent sur la base donnée définissant le réseau. En effet, dans un réseau euclidien, certaines bases sont "meilleures" que d'autres, nous distinguons alors :

— Les bases formées de vecteurs assez courts et plutôt orthogonaux. Ces bases sont dites réduites et permettent la résolution de certaines instances de problèmes sur les réseaux.
— Les bases formées de vecteurs relativement longs et très peu orthogonaux. Ces dernières sont considérées comme des "mauvaises" bases et rendent difficile la résolution des problèmes sur les réseaux.

Afin de résoudre des problèmes sur les réseaux euclidiens, des algorithmes de réductions de réseaux ont été développés dans le but de trouver, à partir d'une base générée aléatoirement, un ou plusieurs vecteurs relativement courts du réseau.

Procédé d'orthogonalisation de Gram-Schmidt

Toute base B peut être transformée en une base orthogonale pour le même espace vectoriel en utilisant la méthode d'orthogonalisation de Gram-Schmidt.
Soit une base $B = \{\mathbf{b}_1, \cdots, \mathbf{b}_d\} \in \mathbb{R}^{n \times d}$ générant un espace vectoriel V. Alors $OGS(B) = \{\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*\}$ est défini par

$$\begin{cases} \mathbf{b}_1^* = \mathbf{b}_1, \\ \mathbf{b}_i^* = \mathbf{b}_i - \sum_{j<i} \mu_{i,j} b_j^* \text{ avec } \mu_{i,j} = \dfrac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}. \end{cases}$$

La matrice de passage est une matrice triangulaire inférieure avec des 1 sur la diagonale. Il est important de noter qu'en général nous n'obtenons pas une base du réseau, car les $\mu_{i,j}$ ne sont pas des entiers.

Algorithme de réduction de réseaux

En 1982, Lenstra, Lenstra et Lovász développèrent un algorithme de réduction de réseau [LLL82] , appelé LLL qui détermine une base réduite, dite LLL-réduite, en un temps polynomial. Soit $B = \{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ une base d'un réseau euclidien $\mathcal{L}$ de dimension d. La réduction d'une telle base se réalise par :

1. application du procédé d'orthogonalisation de Gram-Schmidt sur la base B ;
2. translations successives des vecteurs $\mathbf{b}_i$ de telle sorte que la nouvelle base translatée B' engendre le même réseau et $OGS(B) = OGS(B')$;
3. permutation des vecteurs pour obtenir une base ordonnée, soit du plus court au plus long vecteur.

Ce procédé permet d'obtenir une base B' dont le défaut d'orthogonalité est le plus proche possible de 1 pour qu'elle soit presque orthogonale.
Pour de petites dimensions de réseaux, l'algorithme LLL donne une solution très approchée au problème SVP$_\gamma$. Cependant, l'algorithme n'est pas aussi efficace lorsqu'il s'agit de grandes dimensions. La difficulté des cryptosystèmes dépend alors de l'incapacité de cet algorithme à résoudre les problèmes sur les réseaux à un facteur près d'environ $\mathcal{O}(\sqrt{n})$, où n est la dimension du réseau. Il reste néanmoins amplement utilisé pour la cryptanalyse.
Il existe également des algorithmes de réduction par bloc, nous pouvons notamment citer l'algorithme BKZ 2.0 [CN11] connu pour être aujourd'hui le meilleur algorithme

permettant de résoudre les problèmes dans le cas pratique, et donc très utilisé pour des preuves de sécurité.

1.2 Microélectronique

1.2.1 Sécurité matérielle

Bien que de nombreux systèmes cryptographiques soient prouvés sûrs mathématiquement, ceux implantés dans certains dispositifs tels que les circuits intégrés laissent place à de nouvelles attaques dites attaques physiques.
Ces attaques s'appuient sur l'exploitation de fuites émanant d'un dispositif exécutant un algorithme cryptographique afin de récupérer l'information secrète (par exemple la clé privée).
Bien que ces attaques puissent être classées de différentes manières, nous distinguons deux grandes familles d'attaques physiques :

- Les attaques actives où l'attaquant se permet de perturber le dispositif cryptographique afin de provoquer des comportements inhabituels des opérations exécutées dans le but d'obtenir des informations sur le secret. Ces attaques sont parfois irréversibles (e.g découpage du circuit) ou pseudo-réversibles telles des attaques par injections de fautes avec un laser consistant à introduire volontairement des erreurs dans le cryptosystème et exploiter le résultat fauté ou non.
- Les attaques passives où l'attaquant se contente d'observer le comportement du dispositif lors de l'exécution de l'algorithme cryptographique et analyse les fuites physiques d'une implantation matérielle. Dans ce cas, nous parlons d'attaques par canaux auxiliaires. La plupart de ces attaques sont non invasives.

Dans cette thèse, nous nous intéressons à ce dernier type d'attaques.

1.2.1.1 Généralités sur les attaques par canaux auxiliaires

Les premières attaques par canaux auxiliaires, sur des implantations cryptographiques, furent introduites par Paul Kocher [Koc96]. Suite à des attaques temporelles sur des implantations telles que RSA ou l'échange de clés Diffie-Hellman, il put retrouver les clés utilisées dans ces derniers. Depuis, d'autres attaques par canaux auxiliaires ont été développées telles que l'analyse de la consommation de courant [KJJ99] et l'étude du rayonnement électromagnétique émis lors de l'exécution d'une primitive cryptographique.
Physiquement, lorsque nous réalisons des opérations arithmétiques ou logiques, cela entraîne des commutations des signaux internes dans les circuits intégrés (technologie CMOS), ces opérations ont donc une influence sur la consommation de puissance du circuit. Dans le cas où le secret se trouve dans le signal, une attaque par analyse de la consommation peut alors être réalisée. La source des fuites d'informations peut alors s'expliquer comme la différence d'activité physique lors de l'écriture de nouvelles valeurs en sortie d'une opération. Ces informations sont acquises à l'aide d'un oscilloscope haute vitesse et d'une sonde de consommation de courant (dans

le cas d'une analyse de la consommation de courant) ou d'une sonde électromagnétique (lorsque l'on parle d'analyse du champ électromagnétique). Les graphes de mesures de consommation ainsi obtenus sont appelés des traces. Une analyse statistique peut alors être réalisée sur les traces conduisant au recouvrement de la donnée secrète. Pour la majorité de ces attaques, la seule contrainte existante est le besoin qu'une donnée (par exemple l'entrée ou la sortie de l'algorithme attaqué) interagisse à une étape de l'algorithme avec la donnée secrète recherchée afin que cette dernière soit retrouvée. Les fuites d'informations résultantes des interactions entre la donnée connue et la donnée secrète peuvent alors être directes (multiplications, additions, XOR, ...) ou indirectes (choix d'un résultat intermédiaire dépendant de la donnée secrète). Les attaques par analyse de la consommation peuvent également être classifiées en deux catégories.

1. *Les attaques non profilées*
Pour ces attaques, nous faisons la supposition que l'attaquant puisse acquérir une ou plusieurs traces avec un secret inconnu fixé et différentes données d'entrées connues. Parmi ces attaques, nous pouvons citer l'analyse simple de la consommation de courant (*Simple Power Analysis* - SPA) qui consiste à identifier les séquences d'un algorithme par l'observation d'une trace ou de peu de traces de consommation. La SPA nécessite une bonne connaissance de l'implantation de l'algorithme cryptographique. L'idée derrière la SPA se base sur le principe suivant

si $bit = 0$ **alors**
$r = op_1(x)$
sinon
$r = op_2(x)$
fin si

Si la consommation de courant (amplitude, temps d'exécution) est différente pour op_1 et op_2, nous pouvons ainsi être en mesure de déterminer quelle opération a été réalisée en fonction de la valeur du bit traité. Lorsque les bits dépendent directement du secret ou si la connaissance de ces bits permettent de retrouver le secret, le signal acquis à partir d'un oscilloscope nous permet alors de récupérer directement les valeurs sensibles.
Lorsque l'environnement est trop bruité ou que le signal acquis est trop faible, nous utilisons alors d'autres méthodes d'analyses utilisant des outils statistiques et un plus grand nombre de traces pour déduire le secret. Elles permettent notamment de retrouver les informations sur les secrets en exploitant les corrélations statistiques entre la mesure de la consommation d'un matériel et les données traitées. Pour cela, les attaques par analyse statistique se basent sur un modèle de fuite permettant de prédire cette consommation. Cette dernière étant dépendante du nombre de bits de sortie qui changent d'état, deux modèles de fuite sont principalement utilisés dans la littérature : les modèles du *poids de Hamming* et de la *distance de Hamming*. Les attaques par analyse statistique se basent sur la stratégie "diviser pour régner" qui consiste à résoudre un problème initial à partir des solutions des sous-problèmes. Le but de l'attaque est donc de retrouver des parties du secret indépendamment les unes des autres. L'analyse se fait généralement par octet et se décrit en plusieurs étapes :

1. Identification de l'opération à attaquer : trouver dans l'implantation de l'algorithme cryptographique l'opération conduisant à une corrélation entre le courant consommé par le circuit et le secret. Ainsi, la sortie de cette opération définit une variable intermédiaire sensible $v_{k,e}$ dépendant d'une partie du secret k inconnu et d'une entrée connue e. Les différentes opérations menant à cette valeur intermédiaire doivent pouvoir être calculées par le biais d'un modèle logiciel avec pour seule inconnue la clé secrète.
2. Acquisitions des traces : faire l'acquisition de N traces de consommation à l'instant de l'opération attaquée avec un secret (par exemple clé privée) fixe et inconnu, et différentes entrées connues. Nous notons alors k la partie de la clé recherchée, e_i les entrées données et tr_{k,e_i} les traces acquises suite à l'exécution de l'algorithme avec les entrées e_i.
3. Modèle de prédiction : pour chaque hypothèse de clé $\tilde{k}$ et pour chaque entrée e_i, l'attaquant calcule avec un modèle logiciel la valeur intermédiaire $v_{\tilde{k},e_i}$ supposée être manipulée par l'appareil cryptographique. Il choisit alors un modèle de prédiction permettant de faire correspondre cette valeur à la fuite estimée. Ces prédictions sont notées $p(v_{\tilde{k},e_i})$.
4. Étape de distinction : l'attaquant classe les prédictions $p(v_{\tilde{k},e_i})$ en fonction des traces tr_{k,e_i}. Pour cela, il utilise un distingueur lui permettant d'obtenir l'hypothèse la plus probable pour la partie du secret recherchée.

Une des premières attaques de ce type fut l'analyse différentielle de la consommation (*Differential Power Analysis* - DPA) [KJJ99]. Lors de la phase de prédiction, les traces tr_{k,e_i} sont triées en fonction d'un bit de la valeur intermédiaire $v_{\tilde{k},e_i}$ (par exemple, son bit de poids faible), il s'agit alors d'un modèle monobit. Ainsi, pour chaque hypothèse de clé $\tilde{k}$ les traces sont classées en deux ensembles $G_{0,\tilde{k}}$ et $G_{1,\tilde{k}}$ suivant la manipulation d'un bit à "1" ou à "0", par exemple, en considérant le bit de poids faible, nous avons

$$tr_{k,e_i} \in \begin{cases} G_{0,\tilde{k}} & \text{si } v_{\tilde{k},e_i} = 0 \mod 2 \\ G_{1,\tilde{k}} & \text{si } v_{\tilde{k},e_i} = 1 \mod 2. \end{cases}$$

Afin de déterminer la bonne hypothèse de clé, nous effectuons la différence des moyennes des deux groupes. Autrement dit, le distingueur utilisé est

$$D(\tilde{k}) = \frac{\sum_{i|v_{\tilde{k},e_i}=0} tr_{k,e_i}}{|G_{0,\tilde{k}}|} - \frac{\sum_{i|v_{\tilde{k},e_i}=1} tr_{k,e_i}}{|G_{1,\tilde{k}}|}.$$

Plus la différence des traces de consommation entre les deux ensembles $G_{0,\tilde{k}}$ et $G_{1,\tilde{k}}$ diffère, plus la classification est juste. En effet, lorsque l'hypothèse de clé est fausse, les deux ensembles contiendront en moyenne le même nombre de traces correspondant à un bit de poids faible "0" et un bit de poids faible "1". Dans le cas où l'hypothèse de clé est correcte, la classification des traces sera plus réaliste. Nous en déduisons que la clé k recherchée correspond à l'hypothèse $\tilde{k}$ pour laquelle la courbe $D(\tilde{k})$ présente la valeur la plus élevée. Bien que la DPA soit une attaque très

efficace, elle est parfois sujette à des "pics fantômes" qui désignent les traces pour lesquelles $\tilde{k} \neq k$ présentent des biais statistiques élevés, pouvant ainsi perturber la bonne identification de la clé. Ce problème fut résolu par l'introduction, en 2004, de l'attaque par corrélation (Correlation Power Analysis - CPA) [BCO04] dont le but est de calculer directement une corrélation entre la consommation de courant et des valeurs de prédictions $p(v_{\tilde{k},e_i})$ pour chacune des parties du secret. Le modèle de prédiction p peut être par exemple le poids de Hamming ou la distance de Hamming. Cette attaque utilise ainsi comme distingueur le coefficient de corrélation de Pearson permettant de quantifier la dépendance linéaire entre deux variables aléatoires et définie dans notre cas par

$$C(\tilde{k}) = \frac{\sum_{i=1}^{N}(tr_{k,e_i} - \overline{tr_k}).(p(v_{\tilde{k},e_i}) - \overline{p(v_{\tilde{k}})})}{\sqrt{\sum_{i=1}^{N}(tr_{k,e_i} - \overline{tr_k})^2}\sqrt{\sum_{i=1}^{N}(p(v_{\tilde{k},e_i}) - \overline{p(v_{\tilde{k}})})^2}},$$

avec $\overline{tr_k} = \frac{1}{N}\sum_{i=1}^{N} tr_{k,e_i}$ et $\overline{p(v_{\tilde{k}})} = \frac{1}{N}\sum_{i=1}^{N} p(v_{\tilde{k},e_i})$.

Plus la valeur $C(\tilde{k})$ tend vers ± 1, plus la corrélation entre la consommation tr_{k,e_i} et la prédiction $p(v_{\tilde{k},e_i})$ est significative. Ainsi, afin de retrouver la partie de la clé k, il nous suffit de retrouver la valeur qui maximise le coefficient de Pearson.

La Figure 1.5 résume les étapes, expliquées précédemment, d'une attaque non profilée sur un système de déchiffrement $\mathcal{D}$.

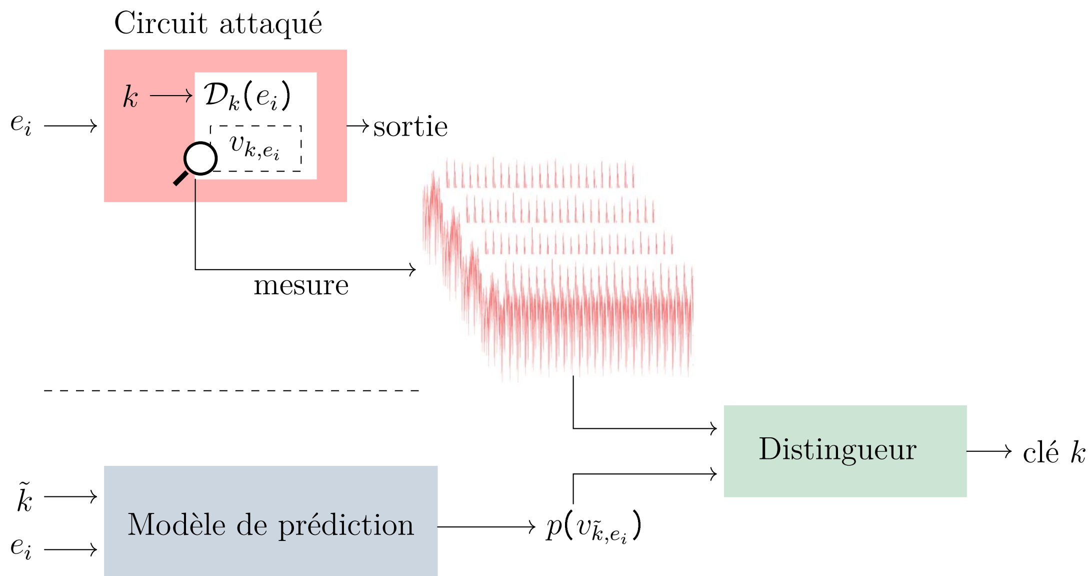


FIGURE 1.5 – Attaque non profilée

2. *Les attaques profilées*

Les attaques profilées utilisent également des outils statistiques pour l'analyse de la consommation, cependant, pour ces dernières, nous nous plaçons dans le pire

scénario, autrement dit l'attaquant a accès à un dispositif clone et "ouvert" du circuit intégré attaqué. L'adversaire a donc un contrôle sur les données d'entrée ainsi que celles, ou une partie des données, se trouvant sur le circuit (e.g la clé privée). Les attaques profilées sont ainsi vues comme les attaques par canaux auxiliaires les plus puissantes. Suite à l'identification de l'opération sensible de l'algorithme implanté, l'attaque se déroule en deux étapes :

1. Phase de profilage : cette étape consiste à trouver un modèle qui caractérise le comportement de fuites du circuit attaqué. Cette caractérisation se réalise par l'acquisition de traces profilées correspondantes aux mesures de consommation de l'opération ciblée avec des données d'entrée e_i et des valeurs de la variable sensible k connues. Ainsi, l'attaquant a en sa possession des couples (e_i, k) associés à la trace de consommation correspondante. Pour cette phase, il est nécessaire de s'assurer que le nombre de traces acquises est suffisant, nous donnant ainsi assez d'information pour chaque partie du secret.
2. Phase d'attaque : l'attaquant réalise l'acquisition de quelques traces à partir du dispositif attaqué et dont le secret lui est inconnu. Il est alors en mesure d'identifier le secret ou une partie de ce dernier, en utilisant le modèle de caractérisation construit à la phase précédente.

Les méthodologies de caractérisation et d'attaque varient allant de l'utilisation de matrices de covariance, telle que l'analyse par modèles (template) [CRR03], à l'utilisation de réseaux neuronaux tels que l'attaque par apprentissage profond [MPP16]. Un template est un ensemble de distributions de probabilités qui décrit l'allure des traces en fonction de différents secrets (par exemple, clé privée). En pratique, les mesures de la consommation observées dans des situations identiques présentent des variations imprévisibles dues principalement au bruit, les signaux sont alors aléatoires. Cependant, certaines caractéristiques sont conservées d'une mesure à l'autre telles que la moyenne ou la variance du signal. Afin de décrire les signaux aléatoires, nous utilisons alors des modèles probabilistes. Le bruit est généralement considéré comme gaussien dont la densité de probabilité d'une variable est définie par

$$f_k(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp^{-(x-\mu)^2/2\sigma^2},$$

où μ représente la moyenne et σ l'écart-type. La distribution gaussienne, qui est centrée autour de la moyenne, peut facilement se généraliser à la situation multivariée. Ce qui est intéressant dans le cas du traitement du signal puisque, en considérant qu'une valeur prise à un instant t est considérée comme une variable aléatoire, les traces sont alors vues comme des vecteurs de dimension n. Dans le cas de la distribution gaussienne multivariée, la moyenne μ correspond ainsi à un vecteur de n moyennes, noté $\boldsymbol{\mu}$ et défini pour une clé fixe k par

$$\boldsymbol{\mu}_k = \begin{pmatrix} \mu_{k,0} \\ \mu_{k,2} \\ \vdots \\ \mu_{k,n-1} \end{pmatrix}, \text{ avec } \mu_{k,t} = \frac{1}{N}\sum_{i=0}^{N} tr_{k,e_i}(t).$$

La variance $(\sigma)^2$ est remplacée par une matrice de covariance $\mathbf{\Sigma} \in \mathcal{M}_n(\mathbb{R})$ définie pour une clé k par

$$\mathbf{\Sigma}_k = \begin{pmatrix} var_{k,0} & cov_{k,0,1} & \dots & cov_{k,0,n-1} \\ cov_{k,1,0} & var_{k,1} & \dots & cov_{k,1,n} \\ \vdots & \vdots & \ddots & \vdots \\ cov_{k,n-1,0} & cov_{k,n-1,1} & \dots & var_{k,n-1} \end{pmatrix},$$

avec

$$\begin{aligned} var_{k,t} &= \frac{1}{N}\sum_{i=0}^{N}(tr_{k,e_i}(t) - \mu_{k,t}) \\ cov_{k,t_1,t_2} &= \frac{1}{N}\sum_{i=0}^{N}(tr_{k,e_i}(t_1) - \mu_{k,i})(tr_{k,e_i}(t_2) - \mu_{k,i}), \end{aligned}$$

où t_1 et t_2 sont des instants de la trace différents.
L'échantillon de traces pour une clé k se modélise alors par la densité de probabilité

$$f_k(\mathbf{x}) = \frac{1}{\sqrt{|\mathbf{\Sigma}_k|2\pi}^n}\exp(-(\mathbf{x} - \boldsymbol{\mu})^T\mathbf{\Sigma}_k^{-1}((\mathbf{x} - \boldsymbol{\mu}_k).$$

Si nous cherchons à modéliser la totalité des traces pour chaque valeur possible de la sous-clé, nous aurions besoin de distributions à n dimension. Or, en pratique, le nombre d'échantillons par trace est très élevé, rendant impossible la construction de nos templates. Cependant, tous les points d'une trace ne sont pas importants, certains points ne contiennent aucune information sur la donnée sensible. Le but est alors de trouver les endroits dans la trace où la partie du secret recherchée se retrouve dans la consommation de courant du circuit. Les points contenant le plus d'information sur la donnée sensible sont appelés points d'intérêts (Point of interest - POIs). Afin que les templates soient réalisables avec un coût mémoire et un temps d'exécution raisonnables, il est nécessaire de choisir peu de POIs. Pour cela, nous sélectionnons les points de traces variant fortement suivant les sous-clés utilisées. Une des méthodes couramment utilisée est le calcul de la somme des carrés des écarts (SOST) [GLRP06] qui se base sur un test statistique permettant de distinguer les signaux bruités et défini par

$$SOST(t) = \sum_{k_1,k_2\in\mathcal{K}} \frac{(\mu_{k_1,t} - \mu_{k_2,t})^2}{\frac{var_{k_1,t}}{n} - \frac{var_{k_2,t}}{n}}, \tag{1.4}$$

où $\mathcal{K}$ représente l'ensemble des sous-clés.

Phase de profilage. Une fois que l'attaquant a sélectionné les POIs, il doit calculer la moyenne et la matrice de covariance afin de modéliser les traces pour chaque valeur possible de la sous-clé $k \in \mathcal{K}$. Pour cela, en supposant que $|\mathcal{K}| = N_{\mathcal{K}}$, il regroupe en $N_{\mathcal{K}}$ classes les traces en fonction de la valeur de la sous-clé k manipulée. L'attaquant peut alors calculer le vecteur de moyennes et la matrice de covariance intra-classe

pour chaque POIs. Il obtient ainsi $N_{\mathcal{K}}$ vecteur de moyennes et matrices de covariance modélisant les traces en fonction de la valeur de la sous-clé k.
Phase d'attaque. Une fois le template construit, l'attaquant acquiert peu de traces, voire une trace dans le cas où le signal est moins bruité, du circuit ciblé avec une clé inconnue. L'attaque consiste alors à classer cette nouvelle trace. Pour cela, il sélectionne les I POIs sur la trace attaquée

$$\mathbf{a} = \begin{pmatrix} a_0 \\ a_2 \\ \vdots \\ a_{I-1} \end{pmatrix}.$$

Pour chaque valeur de clé k, l'attaquant calcule les différentes probabilités $f_k(\mathbf{a})$. Plus la valeur retournée par la fonction de densité est élevée, plus la probabilité que la trace corresponde au coefficient évalué est élevée. Lorsque l'attaquant fait l'acquisition de plusieurs traces, il peut alors combiner les résultats des fonctions de densités pour obtenir une meilleure classification.
La Figure 1.6 présente les étapes d'une attaque profilée sur un système de déchiffrement $\mathcal{D}$.

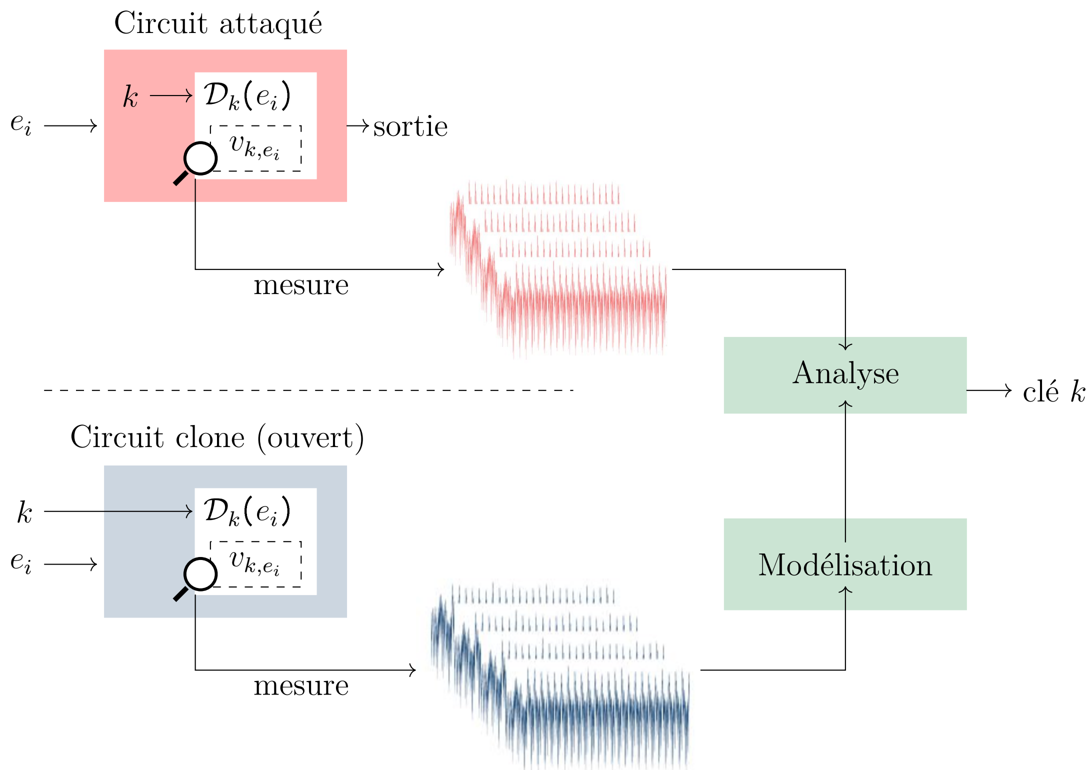


FIGURE 1.6 – Attaque profilée

1.2.1.2 Contre-mesures

Une fois les vulnérabilités d'une implantation mises en évidence, il est nécessaire de la protéger des différentes attaques réalisables. Plusieurs méthodes peuvent ainsi agir aussi bien au niveau matériel, afin de limiter ou bloquer les fuites, qu'au niveau mathématique/algorithmique afin de modifier les opérations de bas niveau et utiliser une méthodologie impactant moins l'analyse de la consommation.
Dans cette thèse, nous nous concentrons sur ce deuxième type de contre-mesures de façon à ce que les solutions proposées soient théoriquement protégées contre les attaques par canaux auxiliaires. Nous distinguons deux familles de contre-mesures logicielles. La première, appelée masquage (masking), consiste à rendre les données intermédiaires manipulées indépendantes des valeurs sensibles. La deuxième consiste à dissimuler les fuites (hiding), au niveau logiciel, cela peut se réaliser par le biais d'ajouts d'aléas dans l'algorithme, permettant de désynchroniser les traces.

1.*Masquage (masking en anglais)*
Supposons que l'on cherche à protéger une donnée secrète qui interagit avec une donnée connue de l'adversaire, alors les fuites d'informations F peuvent s'exprimer comme suit :

$$\mathrm{F} = O(\mathrm{Secret}, \mathrm{Donnée}),$$

$O()$ étant l'activité observable (les fuites) de l'opération attaquée. Nous cherchons alors à rendre l'information secrète décorrélée des fuites observables. Pour ce faire, nous introduisons une troisième variable $Masque$, non déterministe, cachée du point de vue de l'attaquant et idéalement de même entropie que la donnée qu'elle protège. La fuite devient alors

$$\mathrm{F} = O(\mathrm{Secret}, \mathrm{Donnée}, \mathrm{Masque}).$$

Puisque $Masque$ n'est pas observable, l'analyse des traces devient alors très compliquée, voire non réalisable dans certains cas. En pratique, $Masque$ correspond à une décomposition du secret k en nombres aléatoires appelés "masques". Le masquage est dit d'ordre d si la décomposition est réalisée en $d+1$ masques $m_0, \ldots, m_d$. Pour cela, les d premiers masques sont tirés aléatoirement et le dernier masque est calculé comme suit

$$m_d = s \mathbin{\blacksquare} m_0 \mathbin{\blacksquare} \cdots \mathbin{\blacksquare} m_{d-1},$$

où $\blacksquare$ est un opérateur de groupe. Ainsi, nous distinguons le type de masquage en fonction de l'opérateur utilisé.

— Le masquage booléen qui consiste à séparer la variable à masquer par le biais de l'opération booléenne "ou exclusif" (XOR),

$$k = \bigoplus_{i=0}^{d} m_i,$$

— le masquage arithmétique pour lequel la variable sensible est séparée par des additions modulaires,

$$k = \sum_{i=0}^{d} m_i,$$

— le masquage multiplicatif qui utilise la multiplication modulaire pour le masquage du secret

$$k = \bigotimes_{i=0}^{d-1} (m_i)^{-1} \otimes m_d.$$

Ce dernier type de masquage peut être combiné à la fonction de Dirac lorsque l'on cherche à masquer la valeur 0 [GPQ11].

Les masques doivent être aléatoires pour chaque exécution de l'algorithme et leur valeur doit être calculée et conservée au fur et à mesure de l'exécution afin de les retirer à la fin de l'opération protégée.

2. *Traitement aléatoire (shuffling en anglais)*
Lorsque nous effectuons une attaque verticale, telle qu'une attaque DPA/CPA, nous réalisons une prédiction pour chaque valeur possible du secret, à chaque instant de la trace de consommation.Ainsi, nous pouvons, avec des outils statistiques, déterminer à quel point de mesure la fuite liée à valeur sensible apparaît. Cependant, si cette fuite se produit à différents endroits temporels à chaque exécution, l'attaque sera difficilement réalisable. Pour ce faire, nous pouvons traiter de manière aléatoire les opérations dans la fonction ciblée ou mélanger les instructions. Cela permet ainsi de créer une désynchronisation des traces d'une exécution à l'autre et rend les attaques verticales inefficaces. Cependant, cette contre-mesure ne casse pas le lien entre la consommation de courant et les données sensibles manipulées, pour une meilleure résistance, il est courant de combiner traitement aléatoire et masquage.

1.2.2 Matériel utilisé

Les résultats de cette thèse se basent principalement sur un FPGA (Xilinx Virtex) prototypé suivant l'architecture du microcontrôleur de 32 bits MS6001 [Wis18], disponible dans le commerce.

Microprocesseur	ARM SecureCore® SC300™(équivalent à un CORTEX-M3)
Mémoire RAM	24 ko - 4 ko pour les opérations cryptographiques
Mémoire FLASH	1 Mo
Fréquence horloge	50 MHz

Table 1.1 – Propriétés du microcontrôleur utilisé

Un coprocesseur cryptographique permettant d'effectuer les opérations sur $\mathbb{F}_p$, avec p un nombre premier, et $\mathbb{F}_{2^m}$, ainsi qu'un générateur aléatoire matériel (*True Random Number Generator - TRNG*) sont intégrés à la carte.

Ce système embarqué est principalement utilisé pour les attaques par canaux auxiliaires, ce qui nous permet d'avoir une fuite d'information avec moins de perturbations. Nous avons pu acquérir des traces en consommation de courant en mesurant la

tension aux bornes d'une résistance de 1Ω reliée à la masse. Les traces de consommation ont été capturées à partir d'un oscilloscope Lecroy SDA 725Zi-A avec une bande passant de 2.5 GHz. La Figure 1.7 illustre le banc d'attaque permettant l'acquisition des traces pour les attaques par canaux auxiliaires réalisées.

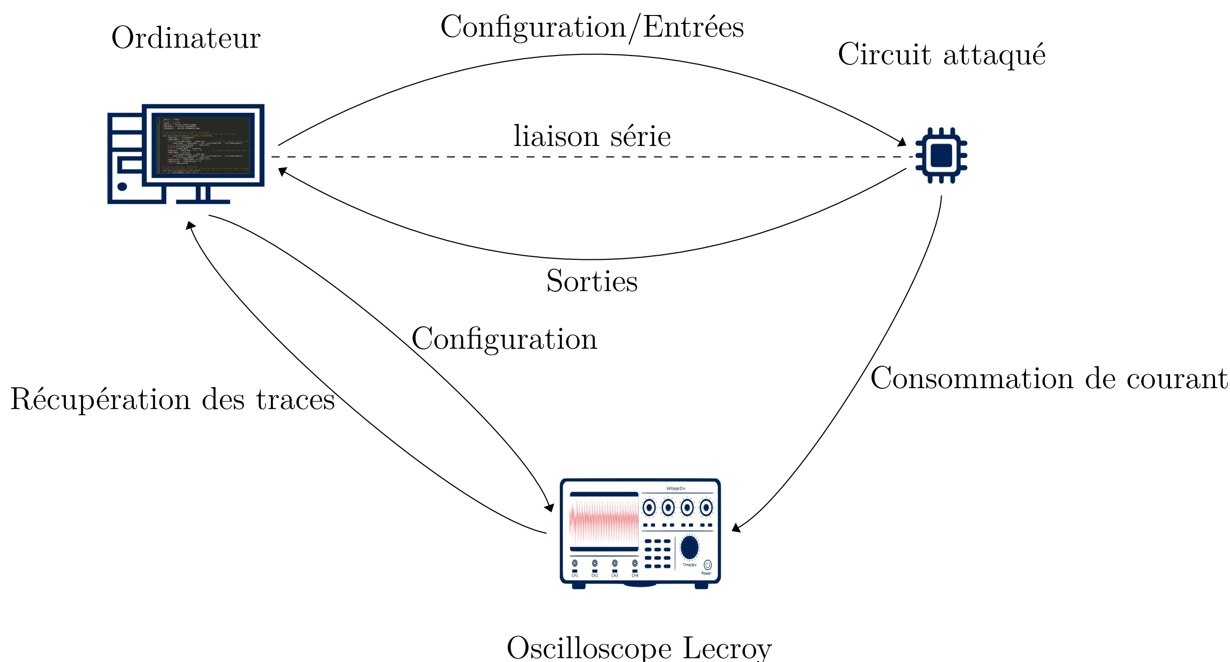


FIGURE 1.7 – Illustration du banc d'attaque pour l'acquisition des traces

1.3 Candidats étudiés

1.3.1 ROLLO

De nombreux candidats à base de codes correcteurs d'erreurs ont été soumis à la standardisation post-quantique du NIST. Pas moins de 20 schémas de signature et de chiffrement et mécanismes d'encapsulation de la clé confondus étaient en lice pour le premier tour. Parmi ces candidats, la métrique de Hamming et la métrique rang s'opposent. Dans cette thèse, nous nous concentrons sur le second tour de la standardisation dont les six candidats à base de codes correcteurs d'erreurs retenus sont présentés dans le Tableau 1.2.
Nous avons décidé de choisir un candidat parmi ceux du second tour afin d'en faire une implantation dans un système embarqué à ressources restreintes. Ce travail permet notamment de préparer la transition vers la cryptographie post-quantique souvent jugée trop coûteuse en temps d'exécution et en taille mémoire.

	Métrique	KEM	PKE	Code	Structure du code utilisée
BIKE [ABB+17a]	Hamming	✓		MDPC	quasi-cyclique
Classic McEliece [BCL+17]	Hamming	✓	✓	Goppa binaires	
HQC [AMAB+17]	Hamming	✓	✓	Reed-Muller/Reed-Solomon	doublement circulant
LEDAcrypt [BBC+20]	Hamming	✓	✓	LDPC	quasi-cyclique
ROLLO [AMAB+19]	Rang	✓	✓	LRPC	idéal
RQC [AMAB+20c]	Rang	✓	✓	Gadibulin	aléatoire idéal

Table 1.2 – Candidats du second tour à base de codes en métrique de Hamming et en métrique rang correcteurs

Un des principaux critères pour le choix du cryptosystème a été l'espace mémoire (principalement RAM) disponible sur le système pour exécuter le protocole cryptographique. Nous avons donc, dans un premier temps, comparé les tailles des éléments manipulés des différents candidats. Ces tailles, pour 256-bits de sécurité classique, sont rapportées dans le Tableau 1.3. Nous avons également regardé les tailles de certains buffers intermédiaires, spécifiques aux implantations de références et nécessaires pour leur exécution. Tous ces paramètres pris en compte, notre choix s'est porté sur la soumission ROLLO, plus particulièrement ROLLO-I. Bien que RQC présente également des tailles de paramètres intéressantes, les paramètres de ROLLO-I restent plus avantageux pour une implantation embarquée. Les opérations de ROLLO-II et ROLLO-III étant similaires, ces derniers pourraient, de plus, être intégrés rapidement pour des futures implantations.

Paramètre \ Schéma	Classic McEliece	BIKE			HQC	LEDAcrypt	RQC	ROLLO		
		I	II	III				**I**	II	III
Clé publique	1044992	8188	4094	9033	14754	4616	2284	**947**	2,493	1138
Clé privée	13892	548	548	532	532	764	2324	**1,894**	4986	2196
Chiffré	240	8188	4094	9033	14818	4616	4552	**947**	2,621	2196

Table 1.3 – Taille des éléments en octet un niveau de sécurité 5.

Suite au second tour de la standardisation du NIST, aucune des deux propositions en métrique rang, ROLLO [AMAB+19] et RQC [AMAB+20c], n'a été sélectionnée pour le troisième tour. Ces derniers n'étant pas été assez étudiés, ils ont été considérés comme pas assez matures pour une standardisation. En effet, une attaque algébrique sur le décodage de codes en métrique rang publiée en 2020 [BBB+20] a amélioré la complexité des attaques existantes, amenant à des modifications sur les paramètres des deux soumissions. À la fin du second tour, de nouvelles versions de ROLLO et RQC ont alors été publiées sur les sites respectifs des soumissions. Les nouvelles tailles des paramètres sont données dans le Tableau 1.4. Dans la nouvelle version, la soumission ROLLO est composée du mécanisme d'encapsulation de clé ROLLO-I et du chiffrement à clé publique ROLLO-II.

	RQC	ROLLO-I	ROLLO-II
Clé publique	4090	1371	2559
Clé privée	4130	2742	5118
Chiffré	8164	1371	2687

TABLE 1.4 – Taille des nouveaux paramètres (en octet)

Bien que ROLLO n'ait été sélectionné pour le troisième tour de la standardisation, dans son rapport [MAA+20], le NIST encourage la poursuite de l'étude pour les cryptosystèmes à base de codes en métrique rang qui sont encore considérés comme une bonne alternative, surtout de par les tailles des clés qui rivalisent avec celles de la famille des réseaux euclidiens. Si bien qu'en septembre 2020, une nouvelle implantation de ROLLO-I-128, en temps constant, a été publiée sur GitHub [AMAB+20b]. Ainsi, nous aborderons également, dans la suite, ces nouvelles implantations de ROLLO et plus précisément les vulnérabilités d'une opération nous permettant de retrouver la clé privée.
Deux articles récents concernant les attaques par canaux auxiliaires sur les codes en métrique rang ont été publiés [AG19, SSPB19]. Ces travaux portent sur des attaques temporelles suite à l'échec de décodage des codes LRPC. Le premier papier n'impacte pas la soumission ROLLO et dans le deuxième, seul un cas pratique concernant le candidat du premier tour McNie a été explicité.

1.3.2 NTRU

Lors du premier tour de la standardisation du NIST, trois soumissions se basaient sur le protocole de chiffrement NTRU, introduit par Hoffstein, Pipher et Silverman [HPS98] et dont la sécurité repose sur le problème du plus court vecteur dans un réseau euclidien : NTRUEncrypt, NTRU-HRSS-KEM et NTRUPrime. Contrairement à la soumission ROLLO qui repose sa sécurité sur des problèmes dans les codes en métrique rang assez récents, la sécurité du protocole NTRU est analysée depuis de nombreuses années ; de plus, parmi les cryptosystèmes post-quantiques, il bénéficie de tailles de clés raisonnables. Dès le premier tour, ces schémas nous ont donc semblé assez prometteurs.
Pour le second tour, les soumissions NTRUEncrypt et NTRU-HRSS-KEM ont fusionné afin de former le candidat NTRU. Ce dernier se base sur le schéma Saito-Xagawa-Yamakawa qui est une variation du schéma proposé dans NTRU-HRSS-KEM. La soumission est composée d'un protocole de chiffrement à clé publique et d'un mécanisme d'encapsulation de clé. Notre choix s'est porté sur le système de chiffrement et plus particulièrement sur la sécurité de ce dernier face aux attaques par canaux auxiliaires.

1.3.3 Crystals-Dilithium

Les schémas de signature sont très utilisés dans les composants électroniques. Il semblait alors important de réaliser une première étude de ce qu'impliquerait l'im-

plantation d'un schéma de signature post-quantique.
Durant le second tour de la standardisation du NIST, trois schémas de signature à base de réseaux euclidiens étaient encore en lice. Parmi ces derniers, donnés dans le Tableau 1.5, Crystals-Dilithium et Falcon présentaient des tailles de paramètres plus intéressantes pour une implantation embarquée avec peu de mémoire. Cependant, la taille des buffers internes à l'implantation ont dû être pris en compte car plusieurs éléments générés dans ces cryptosystèmes nécessitent une utilisation mémoire beaucoup plus conséquente.

Algo. / Param.	Crystals-Dilithium		Falcon		qTesla	
Niveau de sécurité	I	III	I	III	I	III
Clé publique	1472	1760	897	1793	14880	38432
Clé privée	3504	3856	4097	8193	5184	12352
Signature.	2701	3366	690	1330	2592	5664
Problème	MSIS		NTRU		RLWE	

Table 1.5 – Taille des paramètres pour les schémas de signature du deuxième tour de la standardisation

Bien que Falcon soit légèrement plus compact, Crystal-Dilithium présente d'autres avantages tels qu'une implantation plus simple. Cela est notamment dû au fait que Crystals-Dilithium n'utilise pas d'échantillonneur gaussien. De plus, une mauvaise implantation de ce dernier peut facilement mener à des attaques par canaux auxiliaires et se révèle souvent compliqué à masquer.

Première partie

Codes correcteurs d'erreurs

Chapitre 2

Étude du candidat ROLLO

Sommaire

2.1 Implantation de ROLLO-I 44
2.1.1 Opérations sur $\mathbb{F}_2[z]/(P_m)$ 44
2.1.2 Opérations dans $\mathbb{F}_{2^m}[Z]/(P_n)$ 47
2.1.3 Évaluation des performances 53
2.1.4 Nouvelle version de ROLLO 54
2.2 Attaque par canaux auxiliaires 56
2.2.1 Fuites d'information de l'implantation standard 58
2.2.2 Fuites d'information de l'implantation de référence en temps constant 61
2.2.3 Fuites d'information de l'implantation en temps constant sur *GitHub* 66
2.2.4 Résultats expérimentaux des attaques par canaux auxiliaires 70
2.3 Contre-mesures 75
2.3.1 Pour l'implantation de la standardisation 75
2.3.2 Pour les implantations en temps constant 76
2.3.3 Résultats expérimentaux 79
2.4 Conclusion 82

À la suite de l'étude des tailles de paramètres et implantations des candidats à base de codes correcteurs d'erreurs du second tour (voir section 1.3.1), ROLLO nous semblait être le meilleur choix pour une implantation avec peu d'espace mémoire. Ce candidat est issu de la fusion de trois soumissions du premier tour qui se basent sur les codes LRPC expliqués dans la Section 1.1.1.3 :

— LAKE [ABD+17a], un mécanisme d'encapsulation de clé,
— Locker [ABB+17b], un chiffrement à clé publique,
— Ourobouros-R [ABD+17b], un mécanisme d'encapsulation de clé,

respectivement renommés ROLLO-I, ROLLO-II et ROLLO-III. Dans la suite, nous référençons la soumission composée de ces trois schémas comme la soumission officielle du second tour. En effet, une deuxième version, composée de ROLLO-I et

ROLLO-II avec de nouveaux paramètres, a été soumise à la fin du second tour mais n'a pas été prise en compte par le NIST pour la sélection des finalistes. Nous étudions dans ce chapitre les deux soumissions.
La Figure 2.1 présente le fonctionnement de ROLLO-I et ROLLO-II pour lesquels nous avons utilisé les notations suivantes

- $A \stackrel{\$}{\leftarrow} \mathbb{F}_{q^m}^k$ définit l'opération qui choisit aléatoirement k vecteurs de l'espace vectoriel $\mathbb{F}_{q^m}$, ainsi $A \in \mathbb{F}_{q^m}^k$.
- $(\mathbf{u}, \mathbf{v}) \stackrel{\$}{\leftarrow}_l A$ définit l'opération qui choisit de manière aléatoire $2n$ combinaisons linéaires de l'élément $A \in \mathbb{F}_{q^m}^k$, ainsi $\mathbf{u}, \mathbf{v} \in \mathbb{F}_{q^m}^n$ et $Supp(\mathbf{u}, \mathbf{v}) = A$.

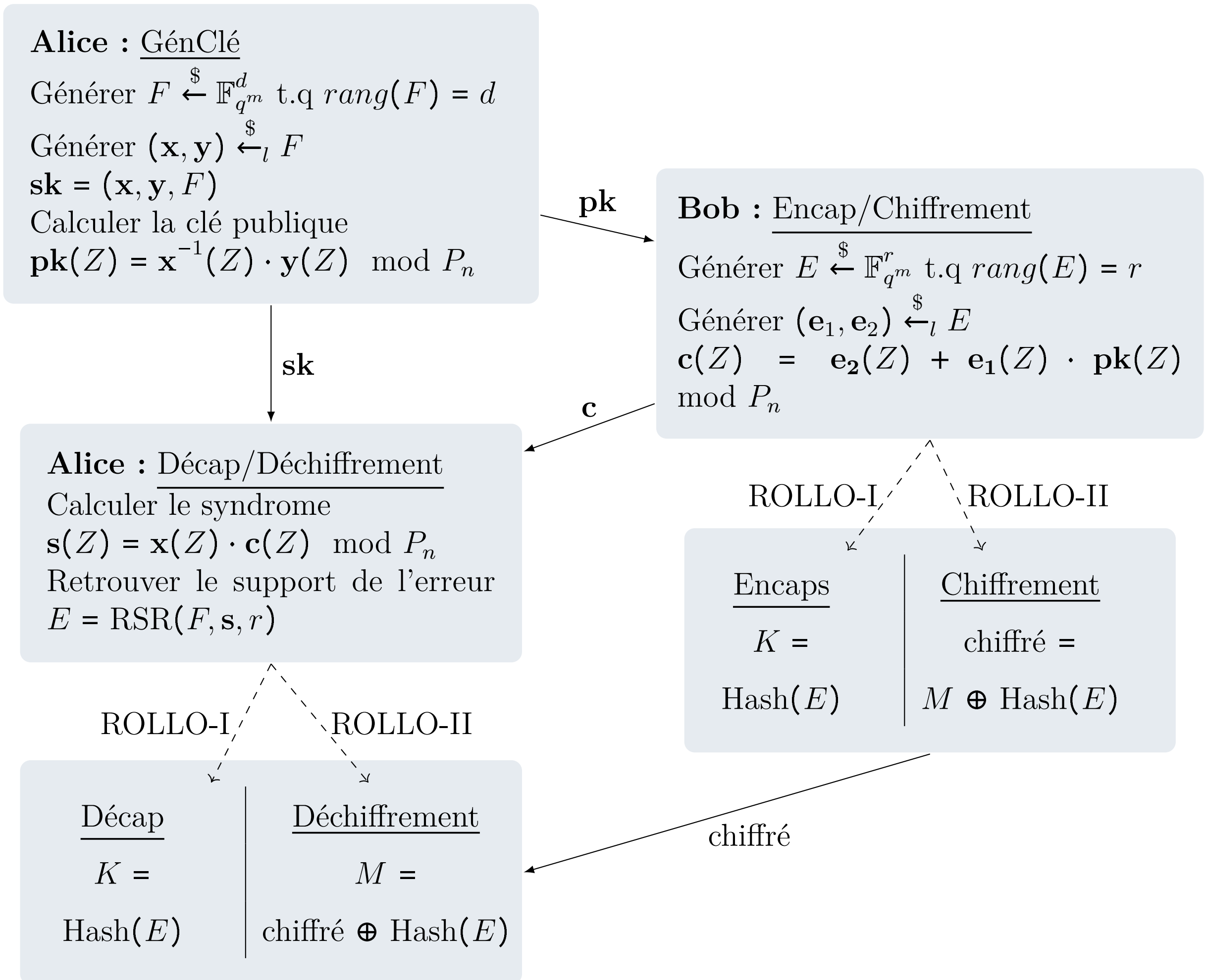


FIGURE 2.1 – Cryptosystèmes ROLLO-I (KEM) et ROLLO-II (PKE)

Comme présenté dans le Tableau 1.3, la taille des paramètres de ROLLO-I est plus avantageuse pour une implantation embarquée que ceux de ROLLO-II. Toutefois, les opérations de ROLLO-II étant similaires, ces dernières pourraient être intégrées rapidement.
Ce chapitre s'articule alors, dans un premier temps, autour des choix d'algorithmes pour l'implantation de ROLLO-I dans l'environnement restreint en espace mémoire décrit dans la section 1.2.2, ainsi que les performances de l'implantation sur cette

cible. Dans un second temps, nous étudions la sécurité de différentes implantations de ROLLO et détaillons des contre-mesures face aux attaques trouvées.

2.1 Implantation de ROLLO-I

Dans cette partie, nous nous focalisons sur la soumission officielle du second tour, disponible sur le site du NIST [1].

L'ensemble des paramètres est donné dans le Tableau 2.1. Le nom de chaque instance fournit le niveau de sécurité classique recherché, par exemple ROLLO-I-128 atteint un niveau de sécurité de 128 bits. Comme décrit dans le chapitre 1, les paramètres m et n correspondent respectivement aux degrés des polynômes irréductibles P_m et P_n utilisés dans la construction des corps $\mathbb{F}_q[z]/(P_m)$ et $\mathbb{F}_{q^m}[Z]/(P_n)$. Pour les trois niveaux de sécurité, q est fixé à 2, le paramètre d correspond au rang du support de la clé privée, noté F et r au rang du support des erreurs, noté E.

Instance	d	r	P_m	P_n
ROLLO-I-128	6	5	$z^{79} + z^9 + 1$	$Z^{47} + Z^5 + 1$
ROLLO-I-192	7	6	$z^{89} + z^{38} + 1$	$Z^{53} + Z^6 + Z^2 + Z + 1$
ROLLO-I-256	8	7	$z^{113} + z^9 + 1$	$Z^{67} + Z^5 + Z^2 + Z + 1$

Table 2.1 – Les paramètres de ROLLO pour les trois niveaux de sécurité

Détaillons maintenant les algorithmes requis pour le cryptosystème ROLLO-I, dans les anneaux $\mathbb{F}_2[z]/(P_m)$ et $\mathbb{F}_{2^m}[Z]/(P_n)$. Pour ces opérations, nous avons pris en compte l'architecture 32-bit du système sur lequel nous implantons ROLLO-I (voir Section 1.2.2).

2.1.1 Opérations sur $\mathbb{F}_2[z]/(P_m)$

Les opérations sur $\mathbb{F}_2[z]/(P_m)$ ont un coût mémoire négligeable car elles manipulent des éléments de $\lceil m/32 \rceil \times 4$ octets, soit des éléments de 12 octets pour ROLLO-I-128 et ROLLO-I-192 et de 16 octets pour ROLLO-I-256. Cependant, une opération dans le corps $\mathbb{F}_{2^m}[Z]/(P_n)$ nous amène à réaliser plusieurs opérations dans le corps $\mathbb{F}_2[z]/(P_m)$. Ainsi, afin de conserver un temps d'exécution raisonnable, il nous fallait implanter des algorithmes rapides en $\mathbb{F}_2[z]/(P_m)$. Dans cette section, nous présentons les différentes opérations et algorithmes sur lesquels nous avons basé notre implantation sur la carte.

Addition : l'addition entre deux éléments de $\mathbb{F}_2[z]/(P_m)$ correspond à un XOR binaire entre mots de 32 bits.

1. https://csrc.nist.gov/Projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-2-submissions

Inversion : pour l'inversion d'un élément sur $\mathbb{F}_2[z]/(P_m)$, nous avons implanté l'algorithme d'Euclide étendu pour les polynômes binaires, tiré de [HMV04, Algo. 2.48] et rappelé en Annexe A.1 (Algorithme 17).

Multiplication : pour calculer $r(z) = a(z) \times b(z)$, où $a, b \in \mathbb{F}_2[z]/(P_m)$, nous utilisons la technique de la multiplication avec une fenêtre glissante de gauche à droite, décrite dans l'algorithme [HMV04, Algo. 2.36] et présentée avec une fenêtre de taille $w = 4$ dans l'Algorithme 3.

Algorithme 3 : Multiplication polynomiale en utilisant la technique de glissement de gauche à droite avec une fenêtre de taille $w = 4$

Entrées : Deux polynômes $a, b \in \mathbb{F}_2[z]/(P_m)$
Résultat : $c(z) = a(z) \times b(z)$

```
1 Pour tout polynôme u de degré inférieur à 4, calculer T[û] = b(z) × u(z)
2 R ← 0
3 pour i allant de 7 à 0 faire
4     pour j allant de 0 à ⌈m/32⌉ − 1 faire
5         Soit û = u3u2u1u0 où uk est le k-ème bit de Aj,i. // A est la
              représentation binaire du polynôme a
6         Ajouter T[û] à C{j}
7     si i ≠ 0 alors
8         c(z) ← c(z) × z^4
9 retourner R
```

En sortie de l'algorithme 3, le produit c est de degré au plus $2 \times m$ car l'algorithme n'effectue pas la réduction dans le corps $\mathbb{F}_2[z]/(P_m)$; cette dernière étant effectuée à la suite de la multiplication.
Pour tout polynôme $a \in \mathbb{F}_2[z]/(P_m)$, nous associons la représentation binaire $A = A_{\lceil m/32 \rceil - 1} \dots A_0$ où A_j est le j-ème mot de 32-bit de A. Nous notons $A_{j,i}$ le i-ème bloc de quatre coefficients dans A_j. De même, nous associons au polynôme c la représentation binaire $C = C_{\lceil 2m/32 \rceil - 1} \dots C_0$, où C_j est le j-ème mot de 32 bits de C et nous posons $C_{\{j\}} = C_{\lceil 2m/32 \rceil - 1} \dots C_j$. La méthode utilisée pour la multiplication consiste à effectuer toutes les multiplications sur les i-ème blocs de quatre bits dans chaque j-ème mot de 32 bits, soit $0 \le i < 8$. Ces multiplications sont ajoutées au résultat puis décalées au fur et à mesure des itérations sur i.

La multiplication se déroule en deux étapes :
1re étape : pré-calculs. Nous commençons par pré-calculer le produit $u(z) \times b(z)$ pour tout polynôme u de degré inférieur à 4, soit 2^4 éléments stockés dans une table T. En définissant $\hat{u}$ comme étant la valeur entière de la représentation binaire du

coefficient du polynôme u i.e

$$\begin{aligned} u(z) = 0 \leftrightarrow b(u) = (0000)_2 \leftrightarrow \hat{u} = 0, \\ u(z) = 1 \leftrightarrow b(u) = (0001)_2 \leftrightarrow \hat{u} = 1, \\ u(z) = z \leftrightarrow b(u) = (0010)_2 \leftrightarrow \hat{u} = 2, \\ \vdots \\ u(z) = z^3 + z^2 + z + 1 \leftrightarrow b(u) = (1111)_2 \leftrightarrow \hat{u} = 15, \end{aligned}$$

le polynôme $b(z) \times u(z)$ est stocké à l'emplacement de le Tableau $T[\hat{u}]$.

2e étape : technique de la fenêtre glissante. À chaque bloc binaire $A_{j,i}$, nous lui associons sa valeur entière $\hat{u}$. Le polynôme $T[\hat{u}]$ nous donne alors la multiplication d'une partie du polynôme a avec le polynôme b. Ainsi, lors de chaque itération $0 \leq j < \lceil m/32 \rceil$, nous ajoutons l'élément $T[\hat{u}]$ à la troncature $C_{\{j\}}$ du résultat C. Si i est non nul, nous multiplions le polynôme c par z^4, ce qui permet d'obtenir à la fin de l'algorithme les degrés corrects pour chaque coefficient. Cette dernière opération revient à une multiplication par 2^4, donc à un décalage de mots de 32 bits.

Réduction modulaire : dans ROLLO-I, nous sommes amenés à effectuer plusieurs réductions modulaires de polynômes creux. Pour cette opération, nous avons utilisé la méthode proposée dans [HMV04, Sec. 2.3.5] qui consiste à utiliser un algorithme de réduction spécifique à polynôme donné, car plus rapide qu'un algorithme de réduction sous forme générique. Nous l'avons donc adapté aux polynômes de ROLLO-I. L'Algorithme 4 explicite la méthode pour ROLLO-I-128.

Algorithme 4 : Réduction modulo $P_m(z) = z^{79} + z^9 + 1$

Entrées : Un polynôme c de degré au plus 156
Résultat : $c(z) \mod P_m(z)$

/* $C_i = (c_{31+32\times i} \cdots c_{32\times i})$ le i-ème bloc de 32 bits du polynôme d'entrée c */
1. $C_2 \leftarrow C_2 \oplus (C_4 \gg 6) \oplus (C_4 \gg 15)$
2. $C_1 \leftarrow C_1 \oplus (C_4 \ll 17) \oplus (C_4 \ll 26) \oplus (C_3 \gg 6) \oplus (C_3 \gg 15)$
3. $C_0 \leftarrow C_0 \oplus (C_3 \ll 17) \oplus (C_3 \ll 26)$
4. $T \leftarrow C_2$ & 0xFFFF8000
5. $C_0 \leftarrow C_0 \oplus (T \gg 15) \oplus (T \gg 6)$
6. $C_2 \leftarrow C_2$ & 0x7FFF
7. $C_3, C_4 \leftarrow 0$
8. **retourner C**

Considérons le polynôme $c(z) = c_0 + \ldots + c_{156} z^{156}$. Nous lui associons la représentation binaire $C = C_0 \ldots C_4$ où C_i est un mot de 32 bits. La réduction modulaire est réalisée pour chaque mot C_i, avec $0 \leq i \leq 4$.

Pour détailler la méthode, prenons l'exemple de la réduction du quatrième mot de C, c'est-à-dire le polynôme $c_{96} z^{96} + c_{97} z^{97} + \cdots + c_{127} z^{127}$ modulo $P_m(z) = z^{79} + z^9 + 1$.

Nous avons

$$\begin{aligned} z^{96} &\equiv z^{17} + z^{26} \mod P_m(z), \\ z^{97} &\equiv z^{18} + z^{27} \mod P_m(z), \\ &\vdots \\ z^{127} &\equiv z^{48} + z^{57} \mod P_m(z). \end{aligned}$$

D'après ces congruences, nous remarquons que la réduction du mot C_3 va être reportée à l'aide d'additions sur les mots C_0 et C_1. Dans un premier temps, deux mots vont être construits à partir de C_3 en le décalant vers la gauche respectivement de 26 et 17 bits. Ces deux mots sont alors ajoutés à C_0. De même, deux autres mots seront construits à partir de C_3 en le décalant vers la droite respectivement de 6 et 15 bits. Ces deux mots sont alors ajoutés à C_1. Au final, la réduction de $C_3 = (c_{127} \dots c_{96})_2$ peut se réaliser par quatre décalages et quatre XOR. La Figure 2.2 schématise cette réduction.

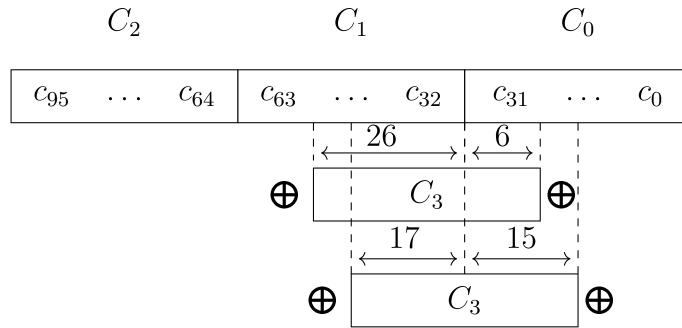


FIGURE 2.2 – Réduction du quatrième mot de 32 bits du polynôme $c(z)$ modulo $P_m(z)$

2.1.2 Opérations dans $\mathbb{F}_{2^m}[Z]/(P_n)$

Pour l'implantation des opérations dans $\mathbb{F}_{2^m}[Z]/(P_n)$, nous avons dû prendre en compte leur coût mémoire. En effet, la taille des éléments étant plus importante dans l'extension de corps, il était nécessaire de pré-calculer l'espace mémoire requis pour que chaque algorithme puisse s'exécuter sur la cible choisie. Dans cette section, nous détaillons les différents algorithmes et opérations implantés.
Pour la suite, m_o représente la longueur en octets des coefficients dans $\mathbb{F}_{2^m}^n$ pour une architecture 32 bits. Comme précédemment énoncé, cela correspond à 12 octets par coefficient pour ROLLO-I-128 et ROLLO-I-192 et 16 octets par coefficient pour ROLLO-I-256.

Génération du support pour un rang donné r : la génération du support consiste à choisir aléatoirement r éléments de $\mathbb{F}_{2^m}$ et vérifier si le rang est maximal (soit r) en utilisant la méthode du pivot de Gauss.

Génération d'élément de $\mathbb{F}_{2^m}^n$ à partir d'un support de rang r : un élément de $\mathbb{F}_{2^m}$ est formé de n combinaisons linéaires aléatoires d'éléments du support dans $\mathbb{F}_{2^m}$.

Addition sur $\mathbb{F}_{2^m}^n$: cela consiste à effectuer n opérations binaires XOR entre éléments dans $\mathbb{F}_{2^m}$.

Multiplication dans $\mathbb{F}_{2^m}[Z]/(P_n)$: elle est l'une des opérations la plus utilisée dans le cryptosystème. En effet, elle est impliquée dans le calcul de la clé publique, du chiffré et du syndrome.
Une optimisation de la multiplication peut ainsi faire gagner un coût considérable en temps d'exécution. Par exemple, lorsque nous multiplions deux polynômes $P(Z) = p_0 + p_1 Z$ et $Q(Z) = q_0 + q_1 Z$ de degré 1 dans un anneau $K[Z]$, nous obtenons

$$P(Z) \times Q(Z) = (p_0 \times q_0) + (p_0 \times q_1 + p_1 \times q_0)Z + (p_1 \times q_1)Z^2.$$

Naïvement, nous comptons quatre multiplications et une addition sur les coefficients. Ainsi, avec la méthode élémentaire, une multiplication dans $\mathbb{F}_{2^m}[Z]/(P_n)$ nécessiterait n^2 multiplications dans $\mathbb{F}_{2^m}$. Contrairement à la méthode ci-dessus, la méthode de Karatsuba [KO62] se base sur l'égalité suivante

$$(p_0 \times q_1) + (p_1 \times q_0) = (p_0 + p_1)(q_0 + q_1) - (p_0 \times q_0) - (p_1 \times q_1).$$

Ainsi, $P(Z) \times Q(Z)$ requiert trois multiplications et quatre additions sur les coefficients, les multiplications $(p_0 \times q_0)$ et $(p_1 \times q_1)$ n'étant comptées qu'une seule fois. Les multiplications dans $\mathbb{F}_{2^m}$ étant plus coûteuses que des additions, le but est de réduire leur nombre. Nous avons donc implanté la méthode de Karatsuba combinée à la multiplication élémentaire, comme décrit dans l'Algorithme 5.

Algorithme 5 : Multiplication de Karatsuba

Entrées : Deux polynômes $\mathbf{f}, \mathbf{g} \in \mathbb{F}_{2^m}[Z]/(P_n)$ et N le nombre de coefficients de $\mathbf{f}$ et $\mathbf{g}$

Résultat : $\mathbf{R}(Z) = \mathbf{f}(Z) \cdot \mathbf{g}(Z)$

1 **si** N *impaire* **alors**
2     $\mathbf{R} \leftarrow$ schoolbook$(\mathbf{f}, \mathbf{g}, N)$
3   **retourner R**
4 $N' \leftarrow N/2$
5 Soit $\mathbf{f}(Z) = \mathbf{f}_0(Z) + \mathbf{f}_1(Z)Z^{N'}$
6 Soit $\mathbf{g}(Z) = \mathbf{g}_0(Z) + \mathbf{g}_1(Z)Z^{N'}$
7 $R_1 \leftarrow$ Karatsuba$(\mathbf{f}_0, \mathbf{g}_0, N')$ // Calcul récursif de $\mathbf{f}_0\mathbf{g}_0$
8 $R_2 \leftarrow$ Karatsuba$(\mathbf{f}_1, \mathbf{g}_1, N')$ // Calcul récursif de $\mathbf{f}_1\mathbf{g}_1$
9 $R_3 \leftarrow \mathbf{f}_0 + \mathbf{f}_1$
10 $R_4 \leftarrow \mathbf{g}_0 + \mathbf{g}_1$
11 $R_5 \leftarrow$ Karatsuba(R_3, R_4, N') // Calcul récursif de R_3R_4
12 $R_6 \leftarrow R_5 - R_1 - R_2$
13 **retourner** $R_1 + R_6 Z^{N'} + R_2 Z^{2N}$

Remplissage des polynômes d'entrée
Ligne 4 de l'Algorithme 5, nous divisons la longueur du polynôme N par 2. Par conséquent, afin que la méthode de Karatsuba soit efficace, la longueur des polynômes en entrée de l'algorithme doit être paire, voire un facteur de puissance de 2. Pour garantir cela, nous utilisons un remplissage des polynômes avec des coefficients nuls. Afin de choisir le nombre de coefficients nuls à ajouter qui réduira le coût de la multiplication dans $\mathbb{F}_{2^m}[Z]/(P_n)$, nous comparons dans la Figure 2.3 le nombre de cycles requis en utilisant la méthode de Karatsuba en fonction de différentes longueurs de polynômes.
En fonction de la mémoire disponible pour une multiplication dans $\mathbb{F}_{2^m}[Z]/(P_n)$, nous pouvons ajouter plus ou moins de remplissage. Par exemple, pour ROLLO-I-128 avec $n = 47$, nous avons 47 coefficients, nous notons que l'ajout d'un coefficient nul nous permettrait d'obtenir plusieurs divisions successives par 2 et donc réduire considérablement le nombre de cycles. Pour ROLLO-I-192, avec $n = 53$, comme nous pouvons voir dans la Figure 2.3, nous avons deux possibilités :

— ajouter trois coefficients nuls aux polynômes. Ainsi, la longueur serait de 56, ce qui requiert un coût mémoire supplémentaire de $3 \times \lceil 89/32 \rceil \times 4 = 36$ octets par polynôme ;
— ajouter onze coefficients nuls aux polynômes. Ainsi, la longueur serait de 64. Ce choix est 10% plus rapide que le premier choix mais requiert $11 \times \lceil 89/32 \rceil \times 4 = 132$ octets supplémentaires par polynôme.

La première possibilité a été retenue car elle représente un meilleur rapport coût mémoire/temps d'exécution.

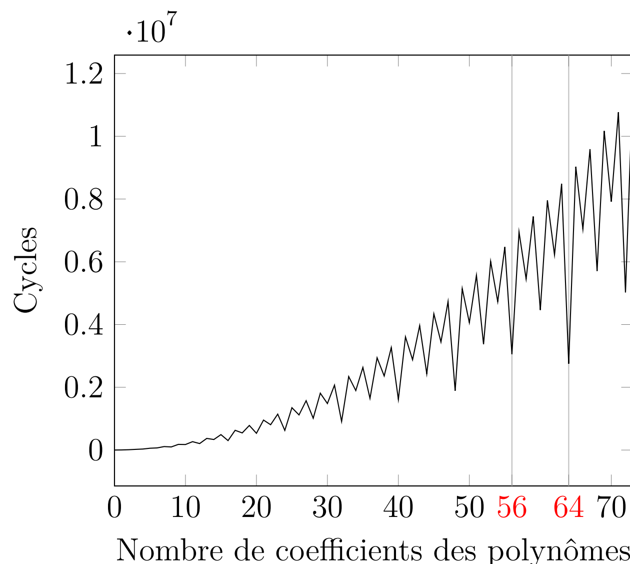


FIGURE 2.3 – Nombre de cycles requis sur CORTEX-M3 par la méthode Karatsuba combinée avec la multiplication élémentaire en fonction du nombre de coefficients des polynômes donnés en entrée.

Inversion dans $\mathbb{F}_{2^m}[Z]/(P_n)$: elle apparaît seulement pour l'étape de la génération des clés. Nous avons alors favorisé le coût mémoire pour l'implantation en

ajustant l'algorithme d'Euclide étendu sur $\mathbb{F}_2[z]/(P_m)$ à l'anneau $\mathbb{F}_{2^m}[Z]/(P_n)$, comme décrit dans l'Algorithme 6. Dans ce dernier, nous notons que plusieurs divisions euclidiennes successives sont effectuées, posant ainsi des contraintes sur l'espace mémoire requis. En effet, durant l'exécution de l'Algorithme 6, nous devons stocker dans la mémoire :

— le polynôme à inverser Q
— une copie du polynôme Q (afin de le conserver en mémoire)
— le dividende, soit le polynôme P_n
— les deux coefficients de Bézout
— trois *buffers* utilisés pour effectuer des opérations intermédiaires (échanges entre polynômes, stockage produits...).

Afin d'exécuter l'inversion, une première solution serait d'allouer l'espace mémoire maximal nécessaire pour tous les éléments. Cependant, un élément dans $\mathbb{F}_{2^m}[Z]/(P_n)$ étant composé d'au plus de n coefficients dans $\mathbb{F}_{2^m}$, le calcul de l'inverse nécessiterait $8 \times n \times m_o$ octets pour le stockage des éléments. En considérant les paramètres de ROLLO-I-128, ROLLO-I-192 et ROLLO-I-256, le coût mémoire serait donc respectivement de 4512, 5088 et 8576 octets, excédant ainsi la taille mémoire disponible sur la carte choisie.

Algorithme 6 : Inversion dans $\mathbb{F}_{2^m}[Z]/(P_n)$

Entrées : Q un polynôme dans $\mathbb{F}_{2^m}[Z]/(P_n)$
Résultat : $Q^{-1} \mod P_n$

1 $U \leftarrow Q$, $V \leftarrow P_n$
2 $G_1 \leftarrow 1$, $G_2 \leftarrow 0$
3 **tant que** $U \neq 1$ **faire**
4 $\quad j \longleftarrow deg(U) - deg(V)$
5 $\quad$ **si** $j < 0$ **alors**
6 $\quad\quad U \leftrightarrow V$, $G_1 \leftrightarrow G_2$, $j \leftarrow -j$
7 $\quad cd_V \leftarrow V_{deg(V)-1}$; // cd_V correspond au coefficient dominant de V
8 $\quad U \leftarrow U + Z^j.(cd_V)^{-1}.V$; // $(cd_V)^{-1}$ est calculé en utilisant l'algorithme d'inversion sur $\mathbb{F}_{2^m}$
9 $\quad cd_G_2 \leftarrow {G_2}_{deg(G_2)-1}$; // cd_G_2 correspond au coefficient dominant de G_2
10 $\quad G_1 \leftarrow G_1 + Z^j.(cd_G_2)^{-1}.G_2$; // $(cd_G_2)^{-1}$ est calculé en utilisant l'algorithme d'inversion sur $\mathbb{F}_{2^m}$
11 **retourner** G_1

Une seconde solution consiste à allouer l'espace exact pour chaque élément et à modifier les allocations mémoires à chaque étape du procédé d'inversion. En effet, au début du procédé d'inversion, le degré du polynôme Q est au plus $n-1$ et le degré de P_n vaut n. Ces degrés sont réduits de 1 suite à chaque division. Par ailleurs, les degrés des deux coefficients de Bézout valent 0 au début du procédé et augmentent de 1 suite à chaque division. Nous en déduisons alors que l'espace mémoire à allouer pour ces éléments se compensent. Le coût mémoire est ainsi de 2590, 2904 et 4864 respectivement pour ROLLO-I-128, ROLLO-I-192 et ROLLO-I-256.

Algorithme de décodage du support du syndrome (*Rank Support Recover (RSR)*) : l'algorithme RSR, dont la version initiale est décrite dans l'Algorithme 2, est une étape importante de la décapsulation car il assure la recherche du support du syndrome. Cependant, plusieurs éléments étant pré-calculés au début de l'algorithme, il est nécessaire d'évaluer le coût mémoire de son implantation.

Dans un premier temps, nous pré-calculons pour $1 \leq i \leq d$, les produits $S_i = f_i^{-1}.S$, où les f_i sont des éléments du support F et S le support du syndrome précédemment calculé. Le syndrome étant de rang $r \times d$, chaque élément S_i est ainsi composé de $r \times d$ coefficients dans $\mathbb{F}_{2^m}$. Nous pouvons en déduire que le pré-calcul des S_i requiert $r \times d \times d \times m_o$ octets.

Remarque 17. Les f_i^{-1} sont calculés avec l'algorithme d'inversion sur $\mathbb{F}_{2^m}$.

Dans un second temps, nous pré-calculons les intersections $S_{i,i+1} = S_i \cap S_{i+1}$ pour $1 \leq i \leq d-1$. Pour ce calcul, nous appliquons l'algorithme de Zassenhaus [LRW97]. Considérons deux sous-espaces vectoriels $U = \langle u_0, u_1, \cdots, u_{n-1} \rangle$ et $V = \langle v_0, v_1, \cdots, v_{n-1} \rangle$ et les vecteurs $\mathbf{u} = (u_0, u_1, \cdots, u_{n-1})$ et $\mathbf{v} = (v_0, v_1, \cdots, v_{n-1})$ de $\mathbb{F}_{2^m}^n$. L'intersection $\mathcal{I}_{U,V} = U \cap V$ est calculée suivant la méthode décrite ci-dessous :

1. créer une matrice par bloc $\mathcal{Z}_{\mathbf{U},\mathbf{V}} = \begin{pmatrix} M(\mathbf{u}) & M(\mathbf{u}) \\ M(\mathbf{v}) & 0 \end{pmatrix}$
2. appliquer le pivot de Gauss sur $\mathcal{Z}_{\mathbf{U},\mathbf{V}}$ afin d'obtenir une matrice sous forme échelonnée
3. la matrice finale est de la forme : $\begin{pmatrix} M(\mathbf{w_1}) & * \\ 0 & M(\mathbf{w_2}) \\ 0 & 0 \end{pmatrix}$,

   avec $\mathbf{w_1}, \mathbf{w_2} \in \mathbb{F}_{2^m}^n$, respectivement la base de $U + V$ et $U \cap V$.

L'algorithme de Zassenhaus nécessite l'équivalent de l'écriture dans la mémoire de quatre S_i, autrement dit $4 \times r \times d \times m_o$ octets. De plus, ces pré-calculs requièrent en mémoire le support de la clé privée F, composée de d coefficients dans $\mathbb{F}_{2^m}$ et le support du syndrome S, composé de $r \times d$ coefficients dans $\mathbb{F}_{2^m}$. À l'issue de ces calculs, chaque intersection est composée de r éléments de $\mathbb{F}_{2^m}$. Ainsi, les $d-1$ intersections pré-calculées requièrent $(d-1) \times r \times m_o$ octets. Nous remarquons également qu'il est nécessaire de calculer $S_{i,i+2}$ pour la somme directe effectuée ligne 5 de l'Algorithme 2, c'est pourquoi lors du calcul d'une intersection $S_{i,i+2}$, le coût mémoire moyen est de

$$\text{Mémoire}_{\textit{pré-calculs}} = (r \times d \times (d+6) - r + d) \times m_o.$$

Avec cette formule, nous pouvons donc prédire que le stockage des pré-calculs de ROLLO-I-128 sera de 4332 octets, ce qui est trop conséquent pour la carte choisie.

Cependant, dans l'algorithme 2, nous remarquons que chaque S_i est utilisé deux fois lors du calcul des intersections, à savoir $S_{i,i+1}$ et $S_{i,i+2}$. Afin de réduire le coût mémoire des pré-calculs, nous avons donc proposé de stocker au plus trois S_i et de calculer directement les deux intersections associées comme présenté dans l'Algorithme 7.
Lors du calcul des intersections, nous aurons en mémoire trois S_i, ce qui correspond à $3 \times r \times d \times m_o$ octets. Lors du calcul de la dernière intersection, nous avons donc en mémoire trois S_i, $(2 \times d - 4)$ intersections, le support du syndrome S et le support de la clé privée F, soit $3 \times r \times d \times m_o + (2 \times d - 4) \times r \times m_o + r \times d \times m_o + d \times m_o$ octets. Ainsi, en prenant en compte le coût mémoire de l'algorithme de Zassenhaus, soit $4 \times r \times d \times m_o$ octets, le coût total des pré-calculs est de

$$\text{Mémoire}_{\textit{pré-calculs}} = (10 \times r \times d - 4 \times r + d) \times m_o.$$

Dans cette méthode, le pré-calcul des $S_{i,i+2}$ correspond à $(d-3)$ intersections, soit $(d-3) \times r \times m_o$ octets. Toutefois, elle nous permet de sauvegarder $(d-3) \times r \times d \times m_o$ octets. Ce qui nous amène à un gain total de $(d-3) \times r \times (d-1) \times m_o$ octets.

Algorithme 7 : RSR (Rank Support Recover)

Entrées : $F = \langle f_1, \cdots, f_d \rangle$ un $\mathbb{F}_q$-sous-espace de $\mathbb{F}_{2^m}$, $s = (s_1, \cdots, s_n) \in \mathbb{F}_{2^m}^n$ le syndrome d'une erreur e et r le rang de e
Résultat : Le sous-espace vectoriel E

```
1  Calculer S = ⟨s_1, ···, s_n⟩_{F_q}
   // On rappelle que S_i = f_i^{-1} S et S_{i,j} = S_i ∩ S_j
2    tmp_1 ← S_1
     tmp_2 ← S_2
     tmp_3 ← S_3
     Calculer S_{1,2} = tmp_1 ∩ tmp_2
     pour i allant de 1 à d − 2 faire
        Calculer S_{i+1,i+2} = tmp_{i+1} ∩ tmp_{i+2}
        Calculer S_{i,i+2} = tmp_i ∩ tmp_{i+2}
        tmp_{(i−1)%3+1} ← S_{i+3}
3  pour i allant de 1 à d − 2 faire
      // Somme directe entre espaces vectoriels
4     tmp ← S + F · (S_{i,i+1} + S_{i+1,i+2} + S_{i,i+2})
5     si dim(tmp) ≤ rd alors
6        S ← tmp ;
7  E ← ⋂_{1≤i≤d} f_i^{-1} · S
8  retourner E
```

Le gain mémoire pour chaque niveau de sécurité de ROLLO-I est donné dans le Tableau 2.2.

coût mémoire (en octet)	sans optimisation	avec optimisation	gain (en %)
ROLLO-I-128	4332	3432	>20%
ROLLO-I-192	6564	4836	>26%
ROLLO-I-256	12560	8640	>31%

Table 2.2 – Gains mémoire avec la version modifiée de RSR

2.1.3 Évaluation des performances

Pour les mesures de performances, nous avons utilisé le compilateur C/C++ de l'environnement de développement IAR Embedded Workbench pour ARM [2], avec une optimisation en vitesse élevée. Nous pouvons ainsi compter le nombre de cycles d'horloge avec la fonctionnalité de débugueur d'IAR. Pour s'assurer d'obtenir des temps d'exécution conformes, les nombres de cycles donnés par le débugueur d'IAR ont été comparés avec ceux donnés par les *timers* internes de la carte et nous avons pu constater que les résultats étaient équivalents.

Toutes les opérations dans $\mathbb{F}_2[z]/(P_m)$ peuvent être accélérées par l'utilisation du coprocesseur cryptographique en $GF(2^m)$, permettant aux implantations ROLLO-I-128 et ROLLO-I-192 d'être plus rapides que leur version totalement logicielle. À l'exception de ROLLO-I-256 qui est trop lourd pour la carte choisie, pour rappel seuls 4 ko sont disponibles pour les opérations cryptographiques. Les performances, avec une carte tournant à 50 MHz, pour les différents niveaux de sécurité de ROLLO-I sont résumées dans le Tableau 2.3.

		Sur SC300 totalement logicielle			Sur SC300 avec coprocesseur		
Instance		GénClé	Encap	Décap	GénClé	Encap	Décap
ROLLO-I-128	cycles ($\times 10^6$)	15,47	1,99	4,31	8,68	0,55	3,75
	ms	309	40,8	86,3	173,6	11	75
ROLLO-I-192	cycles ($\times 10^6$)	21,31	3,38	7,8	11,11	0,8	6,63
	ms	426	67,6	156	222,2	16	132,6
ROLLO-I-256	cycles ($\times 10^6$)	39,92	6,62	15,54	-	-	-
	ms	798,5	132,5	310,8	-	-	-

Table 2.3 – Temps d'exécution du cryptosystème ROLLO-I

Un élément de $\mathbb{F}_{2^m}^n$ est représenté par $n \times \lceil m/32 \rceil \times 4$ octets. Pour ROLLO-I-128, $m = 79$ et pour ROLLO-I-192, $m = 89$, nous obtenons alors $\lceil 79/32 \rceil = \lceil 89/32 \rceil = 3$ mots de 32 bits pour les deux niveaux de sécurité. Nous en déduisons que les

2. https://www.iar.com/knowledge/learn/debugging/how-to-measure-execution-time-with-cyclecounter/

coûts mémoire de ROLLO-I-128 et ROLLO-I-192 diffèrent seulement en fonction du paramètre n. Pour ROLLO-I-256, un élément est représenté par $\lceil 113/32 \rceil = 4$ mots de 32 bits, ce qui explique, dans le Tableau 2.4, la différence significative du coût mémoire avec les deux niveaux de sécurité inférieurs.

Dans le Tableau 2.4, le coût mémoire fait référence à la RAM requise pour exécuter le cryptosystème. Une fois générées, les clés peuvent être stockées dans la mémoire FLASH puis chargées en RAM pour l'exécution de l'Encapsulation et de la Décapsulation. Nous donnons ci-dessous, le coût mémoire de l'implantation avec la clé publique pour l'Encapsulation et le coût mémoire avec la clé privée pour la Décapsulation et pour un message encaspulé de 64 octets. Nous remarquons que ROLLO-I-256 pourrait être exécuté avec 8 ko de RAM, ce qui reste raisonnable.

Instance \ Algo.	GénClé	Encap	Décap
ROLLO-I-128	3520	2940	3496
ROLLO-I-192	4120	3432	4900
ROLLO-I-256	7440	5872	8704

Table 2.4 – Coût mémoire pour ROLLO-I (en octet)

Afin de voir si ROLLO-I peut être une alternative réaliste aux échanges de clés actuels, nous avons comparé sa version logicielle avec celle du protocole d'échange de clés anonyme Diffie-Hellman dans les courbes elliptiques (ECDH) [Cer09]. Pour rappel, dans ECDH, deux entités calculent deux multiplications scalaires sur $E(\mathbb{F}_q)$ pour établir un secret partagé. Nous observons dans le Tableau 2.5 que les deux implantations sont du même ordre de grandeur mais ROLLO-I présente de meilleures performances et pourrait ainsi être une bonne alternative.

Sécurité	Protocole	cycles ($\times 10^6$)
128	ROLLO-I-128	21,8
	ECDH courbe 256	32,4

Table 2.5 – Comparaison des performances entre ROLLO-I et ECDH pour deux niveaux de sécurité différents.

Remarque 18. Les travaux de cette section sont antérieurs à l'attaque algébrique, sur les codes LRPC, trouvée en 2020 [BBB+20].

2.1.4 Nouvelle version de ROLLO

Dans [BBB+20], les auteurs ont trouvé une amélioration des attaques algébriques existantes sur le décodage du syndrome en métrique rang, réduisant ainsi le niveau

de sécurité des paramètres de ROLLO. En Avril 2020, les auteurs de ROLLO ont donc proposé une nouvelle version avec des paramètres modifiés afin d'atteindre les niveaux de sécurité requis par le NIST. Cette nouvelle version est composée du mécanisme d'encapsulation de clé ROLLO-I et du chiffrement à clé publique ROLLO-II. Le Tableau 2.1 spécifie ces nouveaux paramètres. Pour les trois niveaux de sécurité, q est fixé à 2. Comme dans la section précédente, les paramètres d et r correspondent respectivement au rang de la clé privée F et au rang du support de l'erreur E. Les paramètres m et n sont les degrés respectifs des polynômes P_m et P_n.

Instance	d	r	P_m	P_n
ROLLO-I-128	8	7	$z^{67} + z^5 + z^2 + z + 1$	$Z^{83} + Z^7 + Z^4 + Z^2 + 1$
ROLLO-I-192	8	8	$z^{79} + z^9 + 1$	$Z^{97} + Z^6 + 1$
ROLLO-I-256	9	9	$z^{97} + z^6 + 1$	$Z^{113} + Z^9 + 1$

Table 2.6 – Les paramètres de ROLLO pour les trois niveaux de sécurité

Outre les paramètres, les auteurs de ROLLO ont également modifié l'algorithme de décodage du syndrome comme présenté dans l'Algorithme 1 (Chapitre 1). Ils se basent sur l'algorithme de décodage présenté initialement dans [GMRZ13b] sans considérer la partie recouvrement des coordonnées de l'erreur.

Pour cette version, nous avons intégré les nouveaux paramètres de ROLLO-I-128 à notre précédente implantation (totalement logicielle) et modifié les fonctions dépendantes des polynômes telles que le calcul de l'inverse sur $\mathbb{F}_{2^m}$. Nous avons également modifié l'implantation de l'algorithme de décodage afin de correspondre à celui présenté dans l'Algorithme 1. De plus, le pivot de Gauss a été implanté en temps constant suivant la soumission. Ce dernier est présenté dans les prochaines sections. Le Tableau 2.7 donne l'impact sur les performances et le coût mémoire induit par les nouveaux paramètres de ROLLO-I-128. Nous supposons les clés stockées en mémoire FLASH et chargées en RAM quand ces dernières sont utilisées. La taille des paramètres ayant augmenté, nous notons un impact conséquent sur le coût mémoire. Un élément dans $\mathbb{F}_{2^m}^n$ étant représenté par $n \times \lceil m/32 \rceil \times 4$ octets, pour ROLLO-I-128, il est composé de $83 \times \lceil 67/32 \rceil = 249$ mots de 32 bits. Avec $n = 83$, nous avons ajouté un coefficient aux polynômes pour exécuter Karatsuba afin de ne pas excessivement augmenter le coût mémoire. Cependant, 84 est divisible seulement par 2^2, ce qui contraint l'algorithme à exécuter la méthode Schoolbook sur des polynômes de degré 20. Cela impacte également les temps d'exécution.

Malgré le changement de paramètres, ROLLO-I-128 conserve un coût mémoire assez intéressant pour une implantation dans un système embarqué. Nous notons cependant que de nouvelles optimisations doivent être prises en compte afin d'obtenir de meilleures performances, notamment concernant la multiplication dans le corps $\mathbb{F}_{q^m}[Z]/(P_n)$, ainsi que le calcul de l'inverse de la clé privée dans la génération des clés.

	Nombre de cycles ($\times 10^6$)	coût mémoire (RAM)
Génclé	44,4	6152
Encap	8,89	5112
Décap	13,6	6284

Table 2.7 – Performances et coût mémoire de ROLLO-I-128 avec les nouveaux paramètres

2.2 Attaque par canaux auxiliaires

Suite à l'étude de l'implantation optimisée en mémoire du candidat ROLLO, la question de la sécurité matérielle de l'implantation est alors essentielle.
Dans les cryptosystèmes ROLLO-I et ROLLO-II, plusieurs données secrètes doivent être protégées de l'adversaire : la clé privée, le secret partagé dans ROLLO-I et le message qui a été chiffré dans ROLLO-II.
Dans cette section, nous nous concentrons sur l'analyse des vulnérabilités émanant de l'exécution du pivot de Gauss. Ce dernier est utilisé plusieurs fois dans les cryptosystèmes ROLLO-I et ROLLO-II, à savoir pour calculer :

— le support de l'erreur $(\mathbf{e}_1, \mathbf{e}_2)$, qui nous permet d'obtenir le secret partagé dans le cas de ROLLO-I ou de chiffrer et déchiffrer un message dans le cas de ROLLO-II.

— les intersections dans l'algorithme de décodage du syndrome (Algorithme 2 - RSR). Ces intersections détermineront le support E de l'erreur :

$$E \leftarrow \bigcap_{1 \leq i \leq d} f_i^{-1} \cdot S.$$

— le support S du syndrome $\mathbf{s}$.

Le recouvrement de ces différents éléments nous conduisent au recouvrement des différentes données secrètes. Dans la suite, nous expliquons comment retrouver la matrice syndrome à partir de l'analyse de la consommation du pivot de Gauss. Cette même analyse peut être réalisée afin de retrouver le support de l'erreur.

Comme nous pouvons le voir dans la Figure 2.4, lors du procédé de décapsulation ou de déchiffrement, le pivot de Gauss est appliqué sur la matrice syndrome $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ afin de calculer son support. Retrouver le syndrome nous permettrait alors de déterminer la clé privée du cryptosystème.

Nous savons que le syndrome est dans un premier temps calculé comme :

$$\mathbf{s}(Z) = \mathbf{x}(Z) \cdot \mathbf{c}(Z) \mod P_n,$$

avec $\mathbf{x}, \mathbf{c}, \mathbf{s} \in \mathbb{F}_{q^m}[Z]/(P_n)$. Ainsi, avec la connaissance du syndrome $\mathbf{s}$ et du chiffré $\mathbf{c}$, nous pouvons retrouver la première partie de la clé privée :

$$\mathbf{x}(Z) = \mathbf{s}(Z) \times \mathbf{c}(Z)^{-1} \mod P_n.$$

Une fois l'élément $\mathbf{x}$ retrouvé, nous pouvons déterminer $(\mathbf{y}, F)$, en calculant

$$\mathbf{y}(Z) = \mathbf{pk}(Z) \times \mathbf{x}(Z) \mod P_n \text{ et } F = Supp(\mathbf{x}, \mathbf{y}).$$

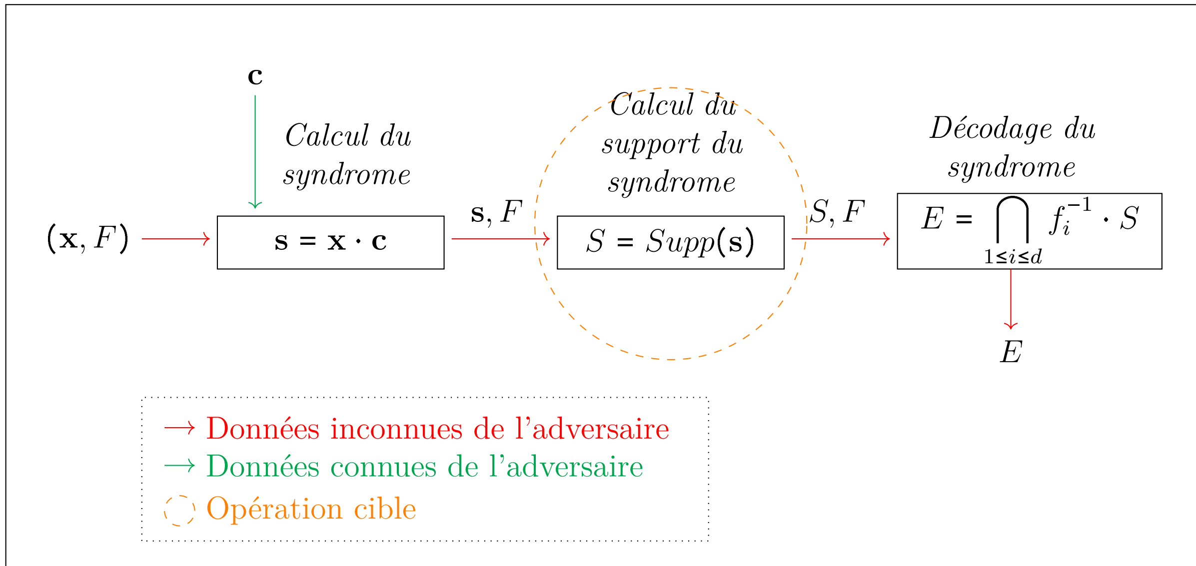


FIGURE 2.4 – Opérations lors de la décapsulation et du déchiffrement

Par l'analyse de la consommation de courant, nous avons pu mettre en évidence que différentes implantations proposées pour le pivot de Gauss sont sujettes à des attaques par analyse simple de la consommation (SPA) et des attaques temporelles. Dans la suite, nous nous penchons sur l'étude de trois implantations :

— Une implantation standard du pivot de Gauss pour les matrices binaires, similaire à celle de la soumission officielle du second tour. Nous la référençons en tant que "implantation standard".

— L'implantation, en temps constant de la nouvelle version mais non prise en compte par le NIST (disponible sur le site officiel de ROLLO [AMAB+19]). Cette implantation utilise la bibliothèque rbc [3] (appelée *rbc_library*) qui fournit plusieurs fonctions implantées pour la métrique rang. Nous nommons cette implantation "implantation de référence en temps constant".

— L'implantation en temps constant publiée sur GitHub et expliquée dans un ePrint disponible sur IACR [AMAB+20b]. Nous la référençons en tant que "implantation en temps constant sur GitHub".

Notations. Nous utilisons les opérateurs $\otimes$ qui définissent la multiplication entre un scalaire et une ligne de la matrice, $\oplus$ le OU EXCLUSIF bit à bit entre deux coefficients ou deux lignes de la matrice. L'opérateur ET est représenté par $\wedge$ et l'opérateur NON par $\neg$.

Remarque 19. Dans la suite, le terme *masque* ne fait pas référence à un masquage booléen mais correspond à une variable permettant d'obtenir les additions sur les lignes suivant les valeurs des coefficients de la colonne en cours de traitement.

3. https://rbc-lib.org/index.html

2.2.1 Fuites d'information de l'implantation standard

L'implantation de référence pour le second tour n'était pas en temps constant, les auteurs ont ainsi appliqué la méthode du pivot de Gauss standard, comme présenté dans l'Algorithme 8, sur la matrice syndrome

$$\mathbf{S} = n \downarrow \overset{\xrightarrow{m}}{\begin{pmatrix} s_{0,0} & s_{0,1} & \cdots & s_{0,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix}}, \tag{2.1}$$

avec $s_{i,j} \in \mathbb{F}_2$ pour $(i,j) \in [\![0, n-1]\!] \times [\![0, m-1]\!]$.

Algorithme 8 : Méthode du pivot de Gauss

Entrées : La matrice syndrome $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : La matrice syndrome $\mathbf{S}$ sous forme échelonnée et le *Rang* de la matrice

```
1  Rang ← 0
2  pour i allant de 0 à m − 1 faire
       // Recherche de la ligne pivot
3      pour j allant de i à n − 1 faire
4          si s_{j,i} = 1 alors
5              // S_j est la ligne pivot
6              s_i ↔ s_j
7              Rang ← Rang +1
8              break
       // Simplification des autres lignes de la matrice
9      pour k allant de ligne i + 1 à n faire
10         si s_{k,i} = 1 alors
               // Annulation des autres coefficients à 1 dans la colonne pivot
11             s_k ← s_k ⊕ s_i
12 retourner (S, Rang)
```

Après application de l'algorithme, nous obtenons une matrice sous forme échelonnée, autrement dit :

$$\mathbf{S} = \begin{pmatrix} 1 & * & * & * & * & * \\ 0 & 1 & * & * & * & * \\ \vdots & \vdots & \ddots & * & * & * \\ 0 & 0 & \cdots & 1 & * & * \end{pmatrix}, \tag{2.2}$$

$\underleftrightarrow{\quad r \times d \quad}$

où $r \times d$ est le rang du syndrome et les motifs $*$ correspondent aux coefficients de la matrice échelonnée.
La première boucle intérieure **pour** de l'Algorithme 8 (ligne 3) permet d'identifier et fixer le pivot dans la colonne en cours de traitement. À partir de la ligne pivot de référence, la première ligne avec un coefficient dominant à 1 devient la ligne pivot. Un échange entre les deux lignes est alors effectué. La deuxième boucle **pour** (ligne 11) permet d'annuler les coefficients à 1 restants dans la colonne pivot. Deux traitements différents sont appliqués en fonction des coefficients dominants des lignes en cours de traitement :

1. Si le coefficient dominant de la ligne traitée est nul, aucune opération n'est effectuée.
2. Si le coefficient vaut 1, alors une addition dans $\mathbb{F}_2[x]/(P_m)$ est effectuée entre la ligne pivot et la ligne traitée.

Dans cet algorithme, nous observons deux sources de fuites d'information. La première consiste en la recherche du pivot. Le temps requis pour déterminer le pivot nous indique le nombre de coefficients traités et nous permet ainsi de retrouver la ligne pivot. La deuxième source provient de la simplification des lignes qui entraîne une différence de traitement en fonction du coefficient dominant de chaque ligne. Cela nous permet ainsi de déterminer les valeurs des coefficients dans la colonne pivot.
Supposons que nous ayons retrouvé la ligne pivot ainsi que les valeurs des coefficients de la première colonne, que nous notons $(\delta_{0,0}, \ldots, \delta_{n-1,0})$. Nous obtenons la matrice syndrome

$$\mathbf{S} = \begin{pmatrix} \boldsymbol{\delta_{0,0}} & * & * & * & * & * \\ \boldsymbol{\delta_{1,0}} & * & * & * & * & * \\ \vdots & \vdots & \ddots & * & * & * \\ \boldsymbol{\delta_{n-1,0}} & * & \cdots & * & * & * \end{pmatrix},$$

où les éléments $*$ nous sont inconnus.

Nous cherchons alors à retrouver les autres colonnes. Suite au traitement de la première colonne, nous pouvons déduire les opérations effectuées sur les différentes lignes. À chaque itération sur les colonnes, nous devons donc considérer les opérations précédemment effectuées. Comme nous pouvons le voir à l'équation 2.2, la matrice binaire obtenue à la fin du procédé du pivot de Gauss n'est pas sous forme réduite, ce qui signifie que les coefficients de la partie supérieure de la matrice n'ont pas été traités durant la réduction. Par exemple, après la deuxième itération sur les colonnes, nous allons retrouver les coefficients $(\delta_{1,1}, \ldots, \delta_{n-1,1})$ où $\delta_{i,1} = (\delta_{0,0} \times \delta_{i,0} \times s_{0,1}) \oplus s_{i,1}$, avec $s_{i,j}$ les coefficients initiaux de la matrice syndrome $\mathbf{S}$, pour $0 \leq i \leq n-1$. Afin de retrouver les coefficients $s_{i,1}$, il nous suffit de résoudre un système d'équations linéaires dans $\mathbb{F}_2$. Cependant, le coefficient $s_{0,1}$ n'ayant pas été traité, il nous est inconnu. Pour la troisième colonne, nous aurons deux coefficients inconnus pour la quatrième trois et ainsi de suite. Étant donné qu'il s'agit de

systèmes d'équations dans $\mathbb{F}_2$, il sera donc de plus en plus compliqué de résoudre les équations de chaque colonne.
Afin de simplifier l'attaque, une solution consiste à traiter chaque colonne de la matrice comme première colonne. Cela revient donc à effectuer m rotations de la matrice **S**. Pour ce faire, nous utilisons le chiffré, connu et donné en entrée du procédé de décapsulation, que nous multiplions par z^i dans $\mathbb{F}_{2^m}[Z]/(P_n)$, avec $1 \le i < m$.
Prenons l'exemple des paramètres de ROLLO-I-128, donnés dans le Tableau 2.1, nous donnons 78 chiffrés en entrée de la décapsulation :

$$\mathbf{c}_1 = \mathbf{c}.z \mod P_n = c_0 \cdot z + c_1 Z \cdot z + \cdots + c_{46} Z^{46} \cdot z \mod P_n$$

$$\mathbf{c}_2 = \mathbf{c} \cdot z^2 \mod P_n = c_0 \cdot z + c_1 Z \cdot z^2 + \cdots + c_{46} Z^{46} \cdot z^2 \mod P_n$$

$$\vdots$$

$$\mathbf{c}_{78} = \mathbf{c} \cdot z^{78} \mod P_n = c_0.z^{78} + c_1 Z \cdot z^{78} + \cdots + c_{46} Z^{46} \cdot z^{78} \mod P_n,$$

avec $c_i \cdot z = c_{i,0} \cdot z + \cdots + c_{i,78} z^{78} \cdot z \mod P_m$.

Comme nous pouvons le voir, la réduction modulo P_m doit être prise en compte lors de la récupération des colonnes de la matrice syndrome. En effet, en donnant en entrée le chiffré $\mathbf{c}_1$, nous obtenons la rotation modulaire de la matrice suivante

$$\begin{pmatrix} 0 & s_{0,0} & s_{0,1} & \cdots & s_{0,8} & \cdots & s_{0,77} \\ 0 & s_{1,0} & s_{1,1} & \cdots & s_{1,8} & \cdots & s_{1,77} \\ \vdots & \vdots & \vdots & & \vdots & & \vdots \\ 0 & s_{46,0} & s_{46,1} & \cdots & s_{46,8} & \cdots & s_{46,77} \end{pmatrix} \begin{bmatrix} s_{0,78} \\ s_{1,78} \\ \vdots \\ s_{46,78} \end{bmatrix}$$

(la colonne $s_{\cdot,78}$ est renvoyée par $\oplus$ vers la première colonne et vers la colonne 8)

Suite à la rotation, la première colonne correspond à la colonne 78 de la matrice initiale **S** et peut être retrouvée suivant les différences de consommation en fonction des coefficients traités, comme expliqué précédemment.
Une fois la colonne 78 retrouvée, nous devrons la prendre en compte lorsque nous chercherons à retrouver la colonne 8. En effet, lorsque nous multiplions le chiffré $\mathbf{c_1}$ par z^{69}, nous retrouvons les coefficients de la première colonne de la matrice syndrome

$$\begin{pmatrix} \delta_{0,8} = s_{0,8} \oplus s_{0,78} \\ \delta_{1,8} = s_{1,8} \oplus s_{1,78} \\ \vdots \\ \delta_{46,8} = s_{46,8} \oplus s_{46,78} \end{pmatrix}$$

Ainsi, afin d'obtenir les coefficients de la colonne 8, nous devons simplifier les coefficients retrouvés en les XORant avec les coefficients de la colonne 78.

Pour ROLLO-I-128 et ROLLO-I-256, cette simplification concernera les colonnes 1 à 8 vu que les modulo respectifs sont $P_m(z) = z^{79} + z^9 + 1$ et $P_m(z) = z^{113} + z^9 + 1$. Pour ROLLO-I-192, dont le modulo est $P_m(z) = z^{89} + z^{38} + 1$ les colonnes 1 à 38 sont concernées.
Nous pouvons ainsi retrouver toute la matrice syndrome par cette méthode.

2.2.2 Fuites d'information de l'implantation de référence en temps constant

Dans la version soumise à la fin du second tour, les auteurs ont proposé une implantation du pivot de Gauss en temps constant, présentée dans l'Algorithme 9 qui a été introduit dans [BCS13].

Algorithme 9 : Pivot de Gauss en temps constant

Entrées : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ sous forme systématique
$rang = min(dimension, n)$

```
1  dimension = 0
2  pour j = 0, ..., m − 1 faire
3      ligne_pivot = min(dimension, n − 1)
4      pour i = 0, ..., n − 1 faire
5          masque = s_{ligne_pivot,j} ⊕ s_{i,j}
6          tmp = masque ⊗ s_i
7          si i > ligne_pivot alors
8              s_{ligne_pivot} = s_{ligne_pivot} ⊕ tmp
9          sinon
10             dummy = s_{ligne_pivot} ⊕ tmp
11     pour i = 0, ..., n − 1 faire
12         si i ≠ j alors
13             masque = s_{i,j}
14             tmp = masque ⊗ s_{ligne_pivot}
15             si dimension < n alors
16                 s_i = s_i ⊕ tmp
17             sinon
18                 dummy = s_i ⊕ tmp
19     dimension = dimension + s_{ligne_pivot,i}
```

De même que dans la section précédente, la matrice est composée de n lignes et m colonnes. Cependant, l'algorithme retourne cette fois-ci la matrice sous une forme systématique, c'est-à-dire sous la forme de l'Équation 2.3.

$$\mathbf{S} = \underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 & * & * & * \\ 0 & 1 & \cdots & 0 & * & * & * \\ \vdots & \vdots & \ddots & 0 & * & * & * \\ 0 & 0 & \cdots & 1 & * & * & * \end{pmatrix}}_{r \times d}, \tag{2.3}$$

avec $*$ des éléments de $\mathbb{F}_2$.
La première boucle intérieure **pour** (ligne 4 de Algorithme 9) fixe les 1 dans la diagonale correspondant aux pivots. La seconde boucle intérieure **pour** (ligne 13 de Algorithme 9) permet de simplifier les autres 1 de la colonne pivot. Dans ces deux boucles, *masque*, élément de $\mathbb{F}_2$, est déterminé puis multiplié avec des lignes spécifiques de la matrice syndrome. Or, la multiplication entre un mot de 32 bits $(u_0, \ldots, u_{31})_2$ et 0 ou 1 a une consommation de courant différente, menant à la récupération des masques calculés durant le procédé. Dans la suite, nous allons montrer comment à partir des valeurs des masques, nous pouvons retrouver la matrice syndrome initiale.
Comme pour la première attaque, nous cherchons à retrouver la matrice syndrome $\mathbf{S}$. La notation $\mathbf{S}_j$ représentera la matrice obtenue après le traitement de la j-ème colonne de $\mathbf{S}$ et $\mathbf{S}[j]$, la colonne j.

Déroulement de l'attaque : à partir des valeurs des masques, nous allons pouvoir définir un système d'équations linéaires pour chaque colonne en suivant les deux étapes ci-dessous.

Suite à la première boucle intérieure "pour" dans l'Algorithme 9 :
Nous retrouvons les valeurs de *masque*, soit $s_{ligne_pivot,j} \oplus s_{i,j}$. Si $masque = 0$, alors la ligne pivot reste inchangée. Sinon, la i-ème ligne est ajoutée à la ligne pivot. Ainsi, la première boucle nous permet de déterminer quelles lignes sont ajoutées à la ligne pivot. Durant le traitement de la colonne j, nous définissons

$$\delta_j = (\delta_{0,j}, \delta_{1,j}, \ldots, \delta_{n-1,j}), \quad \text{où } \delta_{i,j} = \begin{cases} 0 & \text{si } masque = 0 \\ 1 & \text{si } masque = 1 \end{cases},$$

le vecteur contenant toutes les valeurs de *masque* retrouvées suite au traitement de la première boucle **pour**. Nous définissons également la matrice

$$J_k = \begin{pmatrix} 1 & 0 & \ldots & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ \delta_{0,k} & \delta_{1,k} & \ldots & 1 & \ldots & \delta_{n-1,k} \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & 0 & \ldots & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \longleftarrow k\text{-ème ligne} \\ \\ \\ \end{matrix},$$

(flèche sous la matrice : k-ème colonne)

utilisée pour la construction du système d'équations linéaires. Par exemple, considérons la ligne pivot d'indice 0. Suite à la première boucle intérieure **pour**, la matrice syndrome donnée en Équation 2.1 est sous la forme suivante :

$$\begin{pmatrix} \sum_{i=0}^{n-1} \delta_{i,0} s_{i,0} & \sum_{i=0}^{n-1} \delta_{i,0} s_{i,1} & \cdots & \sum_{i=0}^{n-1} \delta_{i,0} s_{i,m-1} \\ s_{1,0} & s_{1,1} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ s_{n-1,0} & s_{n-1,1} & \cdots & s_{n-1,m-1} \end{pmatrix}.$$

Autrement dit, nous pouvons la calculer comme

$$J_0 \times \mathbf{S} = \left(\begin{array}{c|c} 1 & \delta_{1,0} \cdots \delta_{n-1,0} \\ \hline \mathbf{0}_{n-1,1} & I_{n-1} \end{array}\right) \times \mathbf{S}.$$

Nous remarquons que dans l'Algorithme 9, seules les lignes pour lesquelles l'indice est plus grand que celui de la ligne pivot sont ajoutées à la ligne pivot (lignes 7 et 8). Ainsi, à la suite du traitement de la colonne j, nous posons $\delta_{i,j} = 0$ pour $i < ligne_pivot$.

Suite à la seconde boucle "pour" intérieure dans l'Algorithme 9 :

Les valeurs de *masque* retrouvées correspondent aux coefficients $s_{i,j}$ de la matrice obtenue suite à la première boucle intérieure **pour**, soit $J_0 \times \mathbf{S}$.
Nous notons $\delta'_j = (\delta'_{0,j}, \ldots, \delta'_{j-1,j}, *, \delta'_{j+1,j}, \ldots, \delta'_{n-1,j})$, le vecteur contenant les valeurs de *masque* et l'élément $*$ représentant le pivot non traité dans la seconde boucle. Pour l'attaque, ce dernier est remplacé par 1.
Dans un premier temps, à la suite du traitement de la j-ème colonne, δ'_j complète le système d'équations linéaires. Par exemple, supposons que nous souhaitions retrouver la colonne 0, nous résolvons le système d'équations

$$J_0 \times \mathbf{S}[0] = (\delta'_0)^t.$$

Dans un second temps, le vecteur δ'_j nous permet de retrouver les opérations effectuées sur les lignes. Ces additions doivent être prises en compte lorsque nous résolvons le système d'équations linéaires de la $(j+1)$-ème colonne. Pour cela, nous définissons la matrice J'_k ainsi

$$J'_k = \begin{pmatrix} 1 & 0 & \ldots & \delta'_{0,k} & \ldots & 0 \\ 0 & 1 & \ldots & \delta'_{1,k} & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & 1 & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & \delta'_{n-1,k} & \ldots & 1 \end{pmatrix} \begin{matrix} \\ \\ \\ \longleftarrow k\text{ème ligne} \\ \\ \\ \end{matrix} .$$

($\uparrow$ $k$ème colonne)

Par exemple, pour le traitement de la colonne 1, nous considérons la matrice

$$\mathbf{S}_0 = \underbrace{\left(\begin{array}{c|c} (\delta'_0)^t & \begin{array}{c} \mathbf{0}_{1,m-1} \\ \hline I_{n-1} \end{array} \end{array} \right)}_{=J'_0} \times J_0 \times \mathbf{S}.$$

De manière plus générale, au cours du traitement de la colonne j, pour $n > j \geq 1$, nous considérons la matrice

$$\mathbf{S}_{j-1} = \left(\prod_{k=j-1,\ldots,0} J'_k \times J_k \right) \times \mathbf{S}.$$

Finalement, pour retrouver la colonne j, nous résolvons le système d'équations linéaires

$$J_j \times \mathbf{S}_{j-1}[j] = (\delta'_j)^t.$$

NB : dans le cas où il n'y a pas de pivot dans la colonne k, toutes les valeurs des masques valent 0, ainsi $J'_k \times J_k = I_n$ et donc la matrice syndrome, suite au traitement de la colonne k, reste inchangée.

Attaque sur un petit exemple

Pour illustrer l'attaque, prenons le cas de petits paramètres avec $q = 2$, $m = 5$ et $n = 7$.

Supposons que nous voulions retrouver la matrice suivante

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

correspondante au syndrome $\mathbf{s} \in \mathbb{F}_{2^5}^7$.

La matrice recherchée est définie par

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

Après l'exécution du pivot de Gauss, nous devinons à partir de l'analyse de consommation de courant, les valeurs des masques des deux boucles intérieures **pour** :

1. masques de la première boucle :
$(*, 1, 1, 1, 0, 0, 0), (1, *, 1, 0, 1, 1, 0), (1, 0, *, 0, 1, 0, 1), (1, 1, 1, *, 0, 1, 1), (1, 1, 1, 0, *, 1, 0);$

2. masques de la deuxième boucle :
$(*, 0, 0, 0, 1, 1, 1), (1, *, 1, 1, 0, 0, 1), (0, 1, *, 1, 0, 1, 0), (1, 1, 1, *, 0, 1, 0), (1, 1, 1, 0, *, 1, 1),$

où $*$ représente le masque du pivot. Comme expliqué dans la section précédente, les étoiles sont remplacées par 1.

Focalisons-nous sur la récupération des deux premières colonnes de la matrice syndrome. Le vecteur contenant les valeurs des masques de la première boucle $\delta_0 = (1, 1, 1, 1, 0, 0, 0)$ nous donne les additions sur la ligne pivot 0 :

$$J_0 \times \mathbf{S} = \left(\begin{array}{c|cccccc} 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ \hline \mathbf{0} & & & I_6 & & & \end{array}\right) \times \mathbf{S} = \begin{pmatrix} s_0 + s_1 + s_2 + s_3 \\ s_1 \\ s_2 \\ s_3 \\ s_4 \\ s_5 \\ s_6 \end{pmatrix}.$$

Le vecteur des masques de la deuxième boucle $\delta_0' = (1, 0, 0, 0, 1, 1, 1)$ complète ainsi le système linéaire où $s_{i,0}$ sont les inconnues. Ainsi, en appliquant un solveur linéaire, par exemple *SageMath* sur le système

$$J_0 \times \mathbf{S}_{[0]} = (1 \ 0 \ 0 \ 0 \ 1 \ 1 \ 1)^t,$$

nous trouvons le vecteur solution $(1, 0, 0, 0, 1, 1, 1)$, qui correspond bien à la première colonne de la matrice syndrome initiale.
À la fin du procédé de la première colonne, nous obtenons la matrice

$$\mathbf{S}_0 = \left(\begin{array}{c|c} (\delta_0')^t & \begin{array}{c} \mathbf{0} \\ \hline I_6 \end{array} \end{array} \right) \times J_0 \times \mathbf{S}.$$

Pour la seconde colonne, le vecteur des valeurs de *masque* de la deuxième boucle **pour** est $(1, 0, 1, 0, \ 1, 1, 0)$. Cependant, comme expliqué précédemment, seules les lignes pour lesquelles l'indice est plus grand que celui de la ligne pivot sont ajoutées à la ligne pivot. Ainsi, dans le vecteur des valeurs *masque*, nous remplaçons pour $i < 1$ tous les 1 par des 0. Ce qui nous donne le vecteur $\delta_1 = (0, 0, 1, 0, 1, 1, 0)$. De plus, le vecteur des valeurs de *masque* de la deuxième boucle est $\delta_1' = (1, 1, 1, 1, 0, 0, 1)$. Ainsi, cela nous ammène à résoudre le système d'équations

$$\underbrace{\left(\begin{array}{c|cccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ \hline \mathbf{0} & & & I_5 & & & \end{array}\right)}_{J_1} \times \mathbf{S}_0[1] = (1 \ 1 \ 1 \ 1 \ 0 \ 0 \ 1)^t.$$

Le résultat du système correspond au vecteur $(1,0,1,1,1,1,0)$. Suite au traitement de la colonne 1, nous obtenons la matrice

$$\mathbf{S}_1 = \left(\begin{array}{cc|c} 1 & (\delta_1')^t & \mathbf{0} \\ \mathbf{0} & & I_5 \end{array}\right) \times J_1 \times \mathbf{S}_0.$$

Nous effectuons ce même procédé pour les colonnes restantes.

2.2.3 Fuites d'information de l'implantation en temps constant sur *GitHub*

Dans cette partie, nous notons $\mathbf{1} = \underbrace{(11\ldots11)}_{m}$ et $\mathbf{0} = \underbrace{(00\ldots00)}_{m}$.

Suite à la version soumise à la fin du second tour, de nouveaux algorithmes pour ROLLO-I-128 ont été introduits [AMAB+20b] dont une implantation est accessible sur GitHub. Dans cette nouvelle version, les auteurs de [AMAB+20b] définissent une réduction de matrice sous forme échelonnée en temps constant décrite dans l'Algorithme 10.

Algorithme 10 : Réduction de matrice sous forme échelonnée en temps constant

Entrées : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ sous forme échelonnée et son $rang = pivot$

```
1  pivot = 0 ;                                   // indice de la ligne pivot
2  pour j = 0, ···, m − 1 faire
3      pour i = 0, ···, n − 1 faire
4          si s_{pivot,j} == 0 alors
5              masque1 = 1
6          sinon
7              masque1 = 0
8          si s_{i,j} == 1 alors
9              masque2 = 1
10         sinon
11             masque2 = 0
12         si i ≥ pivot alors
13             masque3 = 1
14         sinon
15             masque3 = 0
16         s_pivot ← s_pivot ⊕ (s_i ∧ (masque1 ∧ (masque2 ∧ masque3)))
17         s_i ← s_i ⊕ (s_pivot ∧ (masque2 ∧ masque3))
18     si s_{pivot,j} = 1 et pivot < n alors
19         pivot = pivot + 1
20 retourner S et pivot
```

Cette réduction est alors appliquée à la matrice syndrome lors du procédé de décapsulation. À la sortie de l'Algorithme 10, nous obtenons, comme pour la version de la standardisation, une forme échelonnée (voir Équation 2.2). Pour cela, trois masques sont d'abord calculés suivant les coefficients et les pivots traités. Chaque masque prend la valeur **1** ou **0**. Les trois masques déterminent alors les opérations sur les lignes, comme présenté dans la Figure 2.5. Nous remarquons que deux chemins (en rouge) mènent à des XOR binaires sur les lignes de la matrice. Lorsque $masque1 = masque2 = masque3 = \mathbf{1}$, le coefficient pivot est fixé à 1. Cela se produit au plus une fois au cours du traitement de la colonne j. De plus, lorsque $masque2 = masque3 = \mathbf{1}$, indépendamment de $masque1$, les autres 1 de la colonne j en cours de traitement sont alors simplifiés.

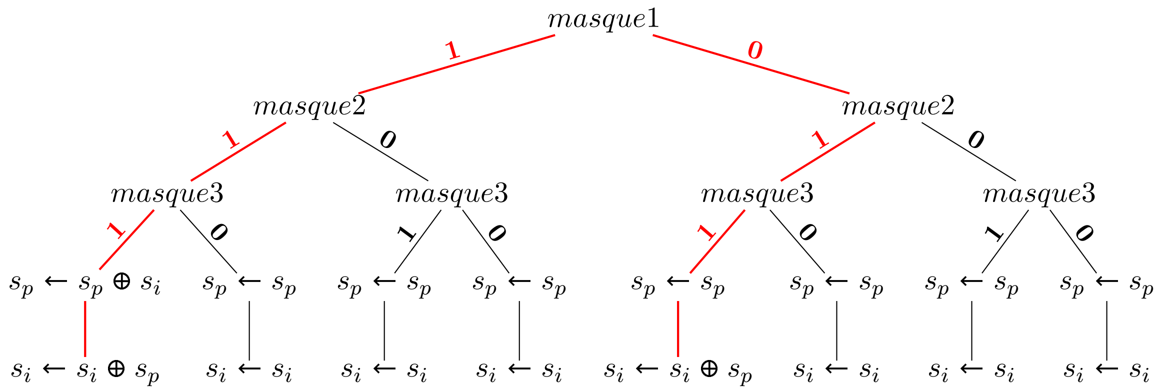


FIGURE 2.5 – Opérations sur les lignes de la matrice en fonction des valeurs des masques. En rouge, les chemins menant à des XOR sur les lignes où s_p dénote la ligne pivot et s_i la ligne traitée.

Dans l'Algorithme 10, nous observons deux sources de fuites. La première concerne le calcul des $masque1$, $masque2$ et $masque3$. Dans l'implantation *GitHub*, considérons la fonction *bf_compute_mask*, donnée dans Figure 2.6, utilisée pour le calcul de $masque2$.

```
1 int bf_compute_mask(bf_element_t *mask, bf_element_t *a, uint8_t bit_position)
2 {
3  // Determine the processed bit
4    uint8_t pos = bit_position / 64u;
5    // Determine the bit position
6    uint8_t bit =  ((uint64_t) (pos * (a->high >> (bit_position - 64u))) ^ (1u-pos)
     *(a->low >> bit_position)) & 0x1u;
7    mask->low = -((uint64_t) bit);
8    mask->high = (uint64_t) -((uint8_t) bit) & ROLLO_I_BF_MASK_HIGH;
9 }
```

FIGURE 2.6 – Fonction tirée de l'implantation en temps constant sur *GitHub*

Nous remarquons que si le coefficient traité, défini par "bit" à la ligne 4 vaut 1, alors tous les bits de $masque2$ sont mis à 1. Sinon, tous les bits sont mis à 0 (lignes 5 et 6). Le même genre d'opérations est observé pour le calcul de $masque1$ et $masque3$. Or, passer tous les bits d'un mot à 1 fuite différemment que lorsque tous les bits

d'un mot sont passés à 0. Ainsi, nous déduisons qu'il serait possible de retrouver toutes les valeurs des masques.

La seconde source de fuites provient des opérations binaires AND et XOR appliquées sur les lignes. En effet, comme décrit dans l'Algorithme 10 (lignes 19 et 20), en fonction des valeurs des masques, les lignes sont XORées avec des lignes nulles ou non-nulles, cette différence peut également être détectable lors de l'analyse de la consommation de courant. Cette source de fuites n'a pas été exploitée ici car elle est équivalente à ce que l'on observe pour retrouver les valeurs des masques. Cependant, elle reste intéressante pour une attaque par canaux auxiliaires et doit être prise en compte lorsque nous cherchons à protéger l'implantation du pivot de Gauss.

Comme précédemment, les valeurs des masques vont nous permettre de construire un système d'équations linéaires. Pour cela, nous définissons trois vecteurs contenant respectivement les valeurs de $masque1$, $masque2$ et $masque2 \wedge masque3$ suite à chaque itération sur j : $\delta_{masque1,j} = (\delta_{0,j}, \ldots, \delta_{n-1,j})$, $\delta_{masque2,j} = (\delta'_{0,j}, \ldots, \delta'_{n-1,j})$ et $\delta_{masque2 \wedge masque3,j} = (\delta''_{0,j}, \ldots, \delta''_{n-1,j})$, avec $\delta_{i,j}$,$\delta'_{i,j}$, $\delta''_{i,j} = 0$ ou 1 si $masque1$, $masque2$, $masque2 \wedge masque3 = \mathbf{0}$ ou $\mathbf{1}$. Comme nous pouvons le voir dans la Figure 2.5, lorsque $masque1 = \mathbf{1}$, seul un chemin mène à des opérations sur les lignes. De plus, une fois que nous obtenons $masque1 = masque2 = masque3 = \mathbf{1}$, alors la variable $masque1$ ne prendra plus la valeur **1** jusqu'à la fin du traitement de la colonne. Ainsi, pour trouver l'indice de la ligne ajoutée à la ligne pivot dans la colonne j, trois cas se présentent :

- Le vecteur $\delta_{masque1,j}$ contient seulement des 0. Dans ce cas, le coefficient dominant de la ligne pivot vaut déjà 1.
- Le vecteur $\delta_{masque1,j}$ contient seulement des 1. Dans ce cas, soit la colonne ne contient pas de pivot, soit le pivot est sur la dernière ligne. Nous devons alors vérifier les valeurs de $masque2$ et $masque3$.
- Le vecteur $\delta_{masque1,j}$ contient des 0 et des 1, alors, la position du dernier 1 correspond à l'indice de la ligne qui a été ajoutée à la ligne pivot dans la colonne j.

Pour déterminer le système d'équations linéaires, nous définissons deux matrices J_k et J'_k dépendantes respectivement de $masque1$ et $masque2 \wedge masque3$:

$$J_k = \begin{pmatrix} 1 & 0 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\ 0 & 1 & \ldots & 0 & \ldots & 0 & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & 1 & \ldots & 1 & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & \underset{\substack{\uparrow \\ \text{indice du pivot}}}{0} & \ldots & 0 & \ldots & 1 \end{pmatrix} \leftarrow k, \quad J'_k = \begin{pmatrix} 1 & 0 & \ldots & \delta''_{0,k} & \ldots & 0 \\ 0 & 1 & \ldots & \delta''_{1,k} & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & 1 & \ldots & 0 \\ \vdots & \vdots & & \vdots & & \vdots \\ 0 & 0 & \ldots & \underset{\substack{\uparrow \\ k}}{\delta''_{n-1,k}} & \ldots & 1 \end{pmatrix} \leftarrow k \quad .$$

La matrice J_k nous donne la position de la ligne pivot et la matrice J'_k les positions

des autres 1 dans la colonne pivot k.

Le vecteur $\delta_{masque2,j}$ dépend des coefficients traités dans la colonne j, ainsi pour la colonne 0, $\delta_{masque2,0}$ car aucune opération sur les lignes n'a préalablement été effectuée.
Suite à la première itération sur les colonnes, nous devons considérer les opérations qui ont été réalisées sur les lignes de la matrice durant le traitement de la colonne 0. Les positions des XOR réalisés sont données par la matrice $J_0' \times J_0$. Ainsi, pour la colonne 1, nous pouvons résoudre le système d'équations linéaires :

$$J_0' \times J_0 \times \mathbf{S}[0] = (\delta_{masque2,1})^t.$$

De manière plus générale, pour retrouver la colonne $j \geq 1$, nous devons résoudre le système d'équations linéaires :

$$\left(\prod_{k=j-1,\ldots,1} J_k' \times J_k\right) \times \mathbf{S}[j] = (\delta_{masque2,j})^t.$$

Petit exemple pour l'attaque sur la version GitHub
Reprenons l'exemple de la section précédente, la matrice syndrome avant échelonnement correspond à

$$\mathbf{S} = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix} \times \mathbf{S}$$

Suite à l'exécution de la réduction du syndrome décrit dans l'Algorithme 10, nous pouvons obtenir, par une analyse de consommation de courant, les valeurs des masques pour chaque colonne :

1. valeurs de $masque1$:
$(0,0,0,0,0,0,0),(1,1,1,0,0,0,0),(1,1,1,1,1,1,0),(0,0,0,0,0,0,0),(1,1,1,1,1,1,0)$

2. valeurs de $masque2$:
$(1,0,0,0,1,1,1),(1,0,1,1,0,0,1),(1,0,0,0,0,1,1),(0,0,0,1,1,1,1),(0,0,1,1,0,1,0)$

3. valeurs de $masque2 \wedge masque3$:
$(0,0,0,0,1,1,1),(0,0,1,1,0,0,1),(0,0,0,0,0,1,1),(0,0,0,0,1,1,1),(0,0,0,0,0,1,0)$.

Nous cherchons à retrouver les deux premières colonnes de la matrice. Comme nous pouvons le voir dans l'Algorithme 10, les valeurs de $masque2$ correspondent aux coefficients de la colonne. Ainsi, le premier vecteur de valeurs de $masque2$ est directement le vecteur solution :

$$(s_{0,0}\ \ s_{1,0}\ \ s_{2,0}\ \ s_{3,0}\ \ s_{4,0}\ \ s_{5,0}\ \ s_{6,0})^t = (1\ \ 0\ \ 0\ \ 0\ \ 1\ \ 1\ \ 1)^t.$$

Considérons à présent la deuxième colonne. Après l'exécution de la colonne 0, des opérations ont été effectuées sur les lignes. Ces opérations nous sont données par les valeurs de $masque1$ et $masque2 \wedge masque3$ de la colonne 0. Toutes les valeurs du vecteur $\delta_{masque1,0} = (0,0,0,0,0,0,0)$ sont nulles, cela signifie que le pivot se trouvait déjà à la ligne pivot 0. Les valeurs du vecteur $\delta_{masque2 \wedge masque3,0} = (0,0,0,0,1,1,1)$ nous donnent directement les positions des autres 1 dans la colonne, ainsi que les lignes qui ont été simplifiées. Dans ce cas, nous avons

$$J_1 = I_7, \quad J_1' = \left(\begin{array}{cc|c} 1 & & \mathbf{0} \\ & (\delta^t_{masque2 \wedge masque3,0})^t & \\ \mathbf{0} & & I_5 \end{array} \right).$$

Pour la deuxième colonne, il nous suffit alors de résoudre le système d'équations

$$J_1' \times J_1 \times S[1] = (1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 0\ \ 1)^t.$$

Ce qui nous donne bien le vecteur solution $(1,0,1,1,1,1,0)$. Nous procédons de même pour les colonnes de la matrice restante.

2.2.4 Résultats expérimentaux des attaques par canaux auxiliaires

Les traces de consommation ont été acquises à partir de l'installation présentée dans la Section 1.2.2. Pour les trois implantations, nous avons placé un trigger avant l'exécution du pivot de Gauss. La première partie de la clé privée $\mathbf{x}$ et le chiffré $\mathbf{c}$ impliqués dans le calcul du syndrome ont préalablement été générés lors des procédés de génération des clés et d'encapsulation de la clé.
Le code source pour les attaques sur les implantations en temps constant est donné en Annexes B.1 et B.2.

Implantation du pivot de Gauss standard
Pour les résultats expérimentaux, nous avons considéré l'implantation de ROLLO-I-128, avec les paramètres $m = 79$ et $n = 47$, utilisant le coprocesseur cryptographique de la carte (voir Section 1.2.2). La matrice syndrome est ainsi composée de 47 lignes, chacune composée de trois mots de 32 bits.
Nous pouvons ainsi observer dans la Figure 2.7 la trace de consommation du traitement de la première colonne. Nous notons alors la différence de motifs lorsqu'un XOR est effectué, dans le cas où le coefficient vaut 1, et lorsqu'il n'y a aucune opération réalisée dans le cas où le coefficient vaut 0. Cette différence de traitement nous permet ainsi de retrouver toute la première colonne de la matrice syndrome

$$(11111100010000011010111111110101000100000100000).$$

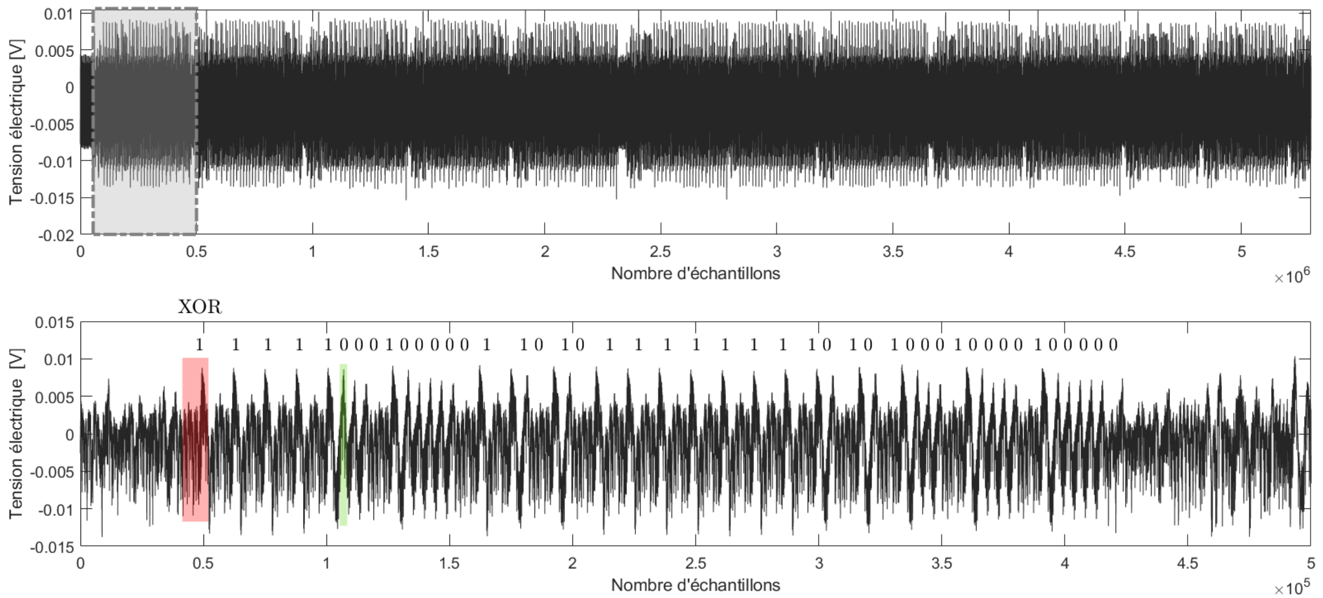


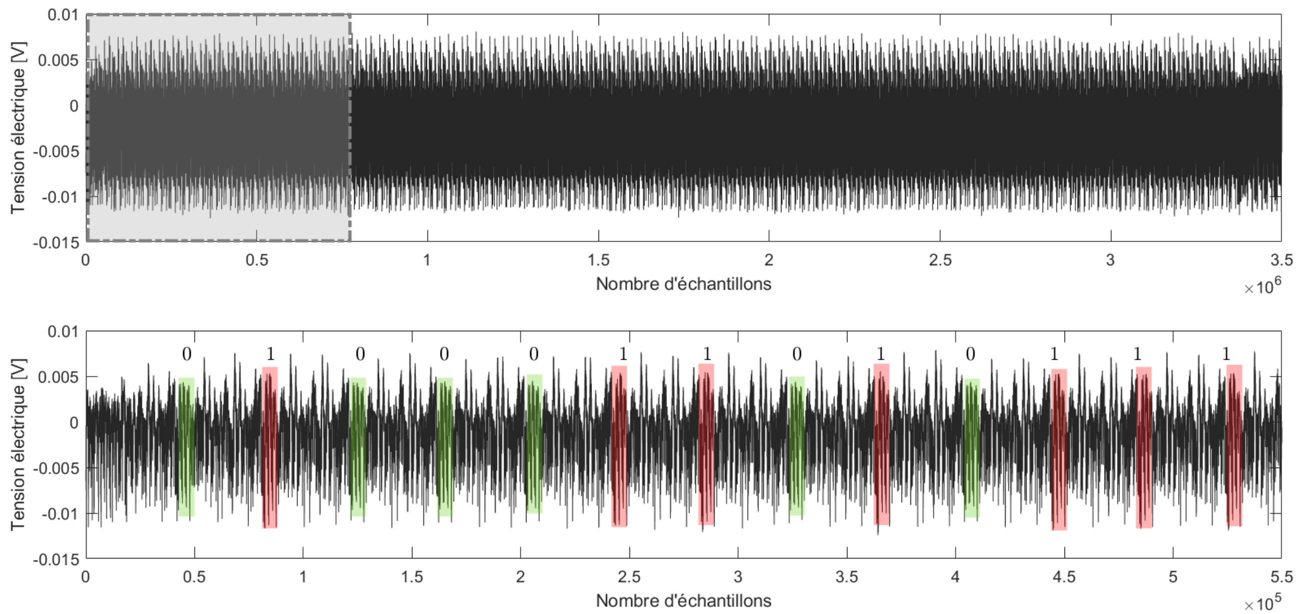
FIGURE 2.7 – Partie de la trace de consommation de l'implantation du pivot de Gauss standard avec un zoom sur la première colonne - en rouge le motif représentant un XOR entre deux lignes (le coefficient est à 1), en vert le motif lorsqu'il n'y pas d'opération (le coefficient est à 0).

Implantation de référence en temps constant
Notre deuxième implantation est basée sur l'implantation de référence en temps constant. Nous avons considéré les paramètres de ROLLO-I-128, la matrice syndrome est composée de 83 lignes et 67 colonnes, ce qui correspond à trois mots de 32 bits. Dans ce cas, $\otimes$ représente la multiplication entre un scalaire et des mots de 32 bits.
La trace de consommation de la première boucle intérieure **pour** (ligne 4 - Algorithme 9) est donnée dans la Figure 2.8a et la trace de la deuxième boucle intérieure **pour** (ligne 13 - Algorithme 9) est donnée dans la Figure 2.8b. Nous observons une différence de consommation quand les mots de 32 bits sont multipliés soit par 0 soit par 1. La différence de motifs nous permet de retrouver les valeurs des masques des deux boucles intérieures **pour**. Par exemple, nous observons dans la Figure 2.8a le début du traitement de la colonne 0 :

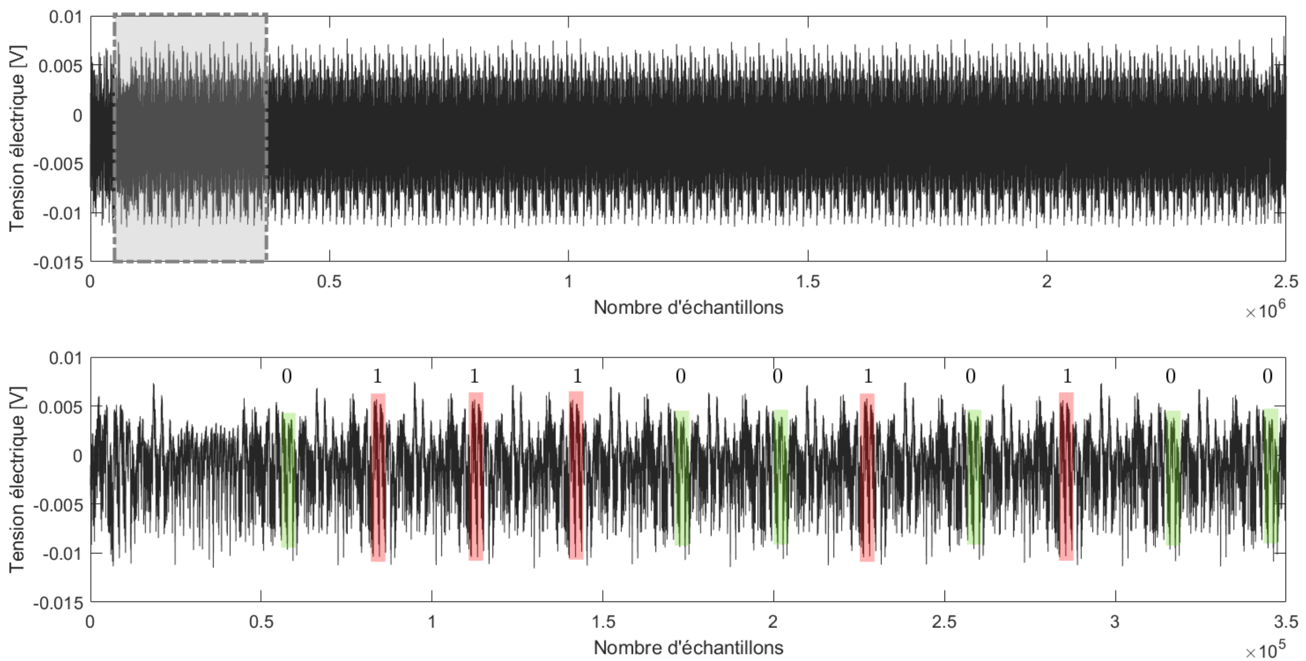
$$\delta_0 = (0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, \cdots).$$

La Figure 2.8b nous permet de retrouver les coefficients des masques de la deuxième boucle intérieure **pour**, pour la colonne 0, nous avons

$$\delta_0' = (*, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, \cdots).$$



(a) Trace complète de la première boucle **pour** et un zoom sur le traitement des premiers coefficients



(b) Trace complète de la deuxième boucle **pour** et un zoom sur le traitement des premiers coefficients

FIGURE 2.8 – Trace des deux premières boucles intérieures **pour** - en rouge le motif représentant le traitement d'une multiplication avec la valeur de masque à 1, et en vert celui d'un masque à 0.

Exemple de trace sur Cortex-M4

Pour la standardisation, la cible officielle pour des implantations embarquées est le processeur ARM Cortex-M4 qui est assez répandu. Nous avons donc décidé de vérifier si l'attaque était également réalisable sur Cortex-M4. La Figure 2.9 donne l'exemple d'une trace du traitement d'une colonne par la première boucle intérieure **pour** et la Figure 2.10 présente la trace de consommation de traitement de cette même colonne, suite à l'exécution de la seconde boucle intérieure **pour** sur un STM32F4. Pour réaliser l'attaque, les traces ont été acquises [4] à partir d'un oscilloscope RTO2000 avec une bande passante de 3 GHz. Bien que la consommation de courant soit différente de celle du Cortex-M3, il est également possible de distinguer une multiplication par 0 ou par 1. Lorsque nous réalisons une multiplication par 0, une chute dans la consommation de courant apparaît dans le motif, ce qui n'est pas le cas lorsqu'une multiplication par 1 est réalisée.

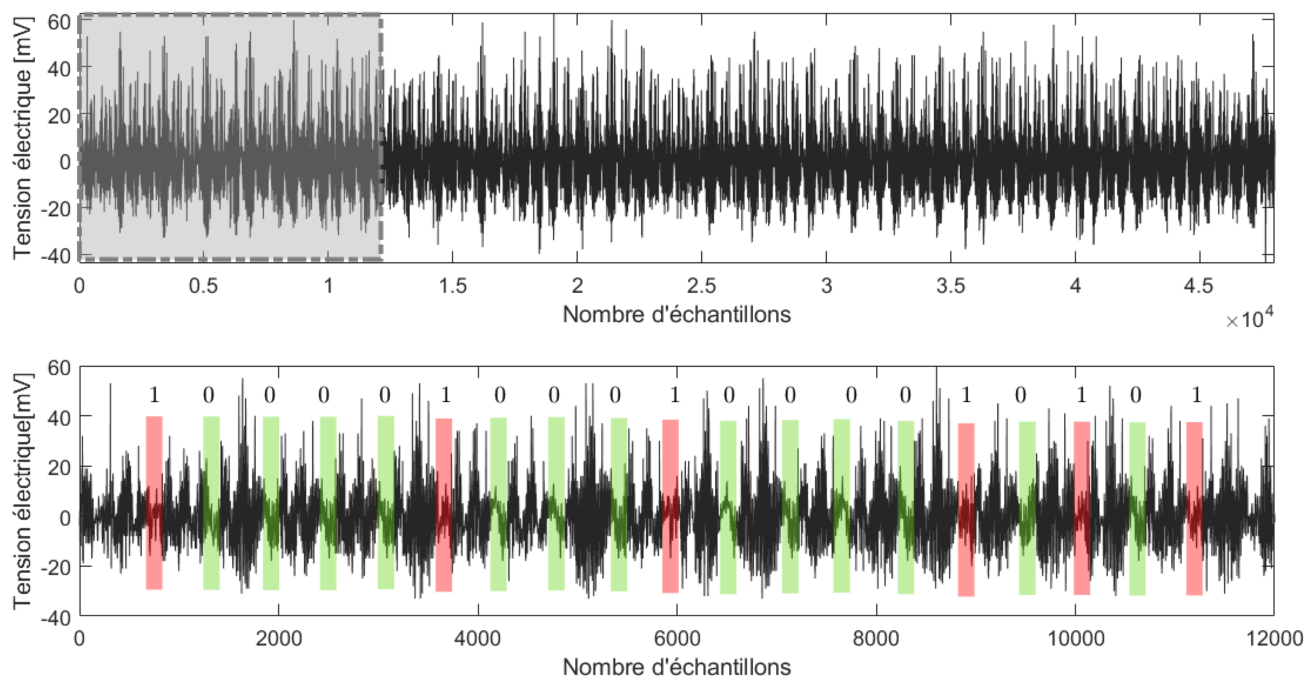


FIGURE 2.9 – Trace d'une première boucle intérieure **pour** - en rouge le motif représentant le traitement d'une multiplication lorsque $masque = 1$, et en vert lorsque $masque = 0$.

4. L'acquisition des traces sur CORTEX-M4 a été réalisée par Agathe Cherière.

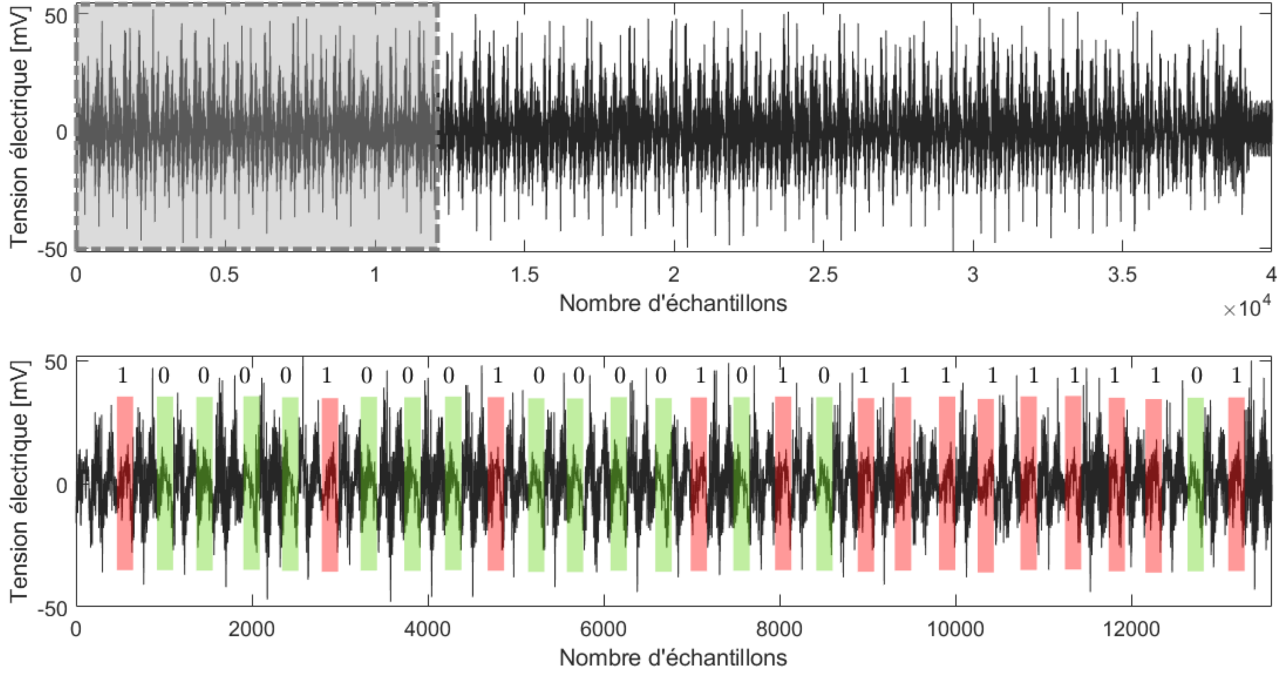


FIGURE 2.10 – Trace de la deuxième boucle intérieure **pour** - en rouge le motif représentant le traitement d'une multiplication lorsque $masque = 1$, et en vert lorsque $masque = 0$.

Implantation sur GitHub en temps constant

Pour la troisième implantation, nous avons intégré le pivot de Gauss correspondant à la version sur *GitHub*. Les mesures pour l'implantation de la version GitHub sont données dans la Figure 2.11. Sur la trace, le premier rectangle correspond au calcul de $masque1$, le deuxième rectangle correspond au calcul de $masque2$ et le troisième rectangle correspond à celui de $masque3$. Nous observons alors une différence d'amplitude lorsque tous les bits des masques sont mis à 1 ou à 0, nous permettant ainsi de définir leurs valeurs. Par exemple, nous pouvons observer sur le zoom de la trace du traitement d'une colonne dans la Figure 2.11, les premières valeurs des trois masques

$$\delta_{masque1,1} = (1,1,1,0,0,0,0,\dots)$$

$$\left.\begin{array}{l}\delta_{masque2,1} = (1,0,1,0,0,1,0,\dots)\\ \delta_{masque3,1} = (0,0,1,1,1,1,1,\dots)\end{array}\right\} \quad \delta_{masque2 \wedge masque3,1} = (0,0,1,0,0,1,0,\dots).$$

À partir des valeurs retrouvées, nous sommes alors en mesure d'effectuer l'attaque précédemment détaillée.

Bien que d'autres variations puissent être observées dans la trace, elles ne sont pas exploitées ici puisqu'elles sont équivalentes à ce nous pouvons récupérer à partir des autres valeurs des masques.

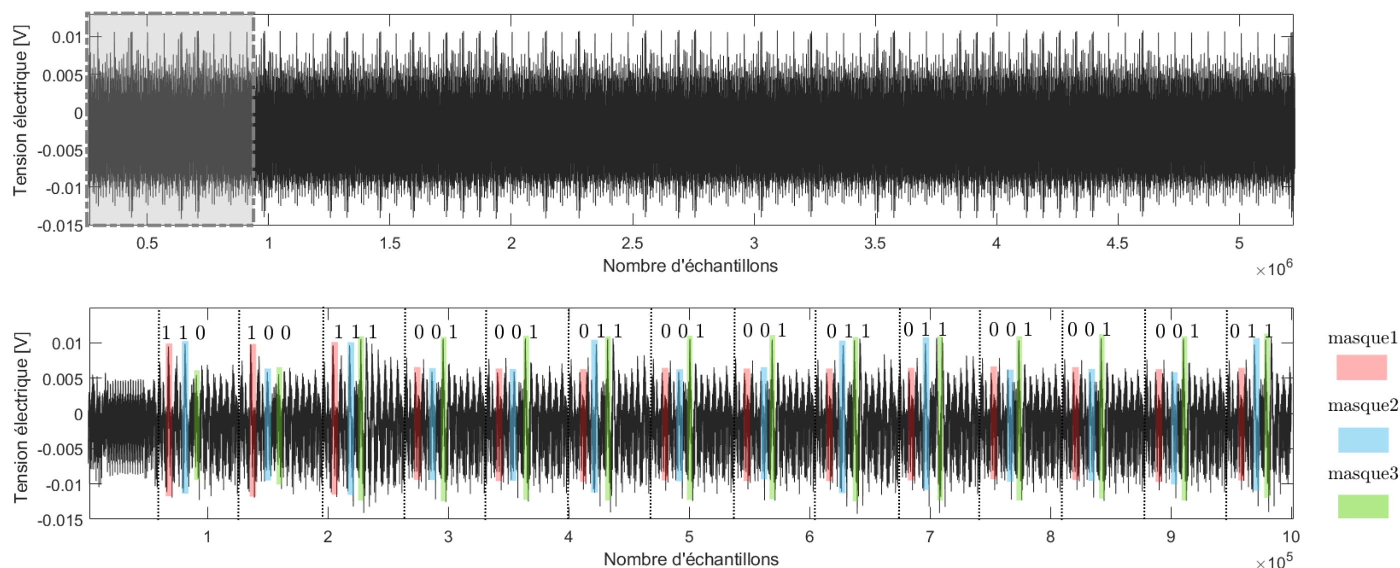


FIGURE 2.11 – Partie de la trace de consommation de l'exécution de la réduction de la matrice sous forme échelonnée avec un zoom sur le traitement des premières lignes de la colonne 1.

Une automatisation de l'attaque est réalisable pour la reconnaissance des différents motifs dans une trace. Cependant, cette analyse qui est encore en cours sera traitée dans la thèse d'Agathe Cherière.

2.3 Contre-mesures

Dans cette partie, nous proposons deux contre-mesures pour protéger les futures implantations de ces attaques. Concernant les deux dernières implantations, il est important de noter que les contre-mesures n'impactent pas le temps constant.

2.3.1 Pour l'implantation de la standardisation

Dans cette partie, nous discutons des solutions pour sécuriser le cryptosystème contre l'attaque expliquée dans la sous-section 2.2.1. Différentes contre-mesures peuvent être appliquées à la protection du pivot de Gauss présenté dans l'Algorithme 8. Ces solutions consistent à :

— Implanter un pivot de Gauss en temps constant dans lequel les additions sur $\mathbb{F}_2[x]/(P_m)$ sont indépendantes des coefficients traités, comme présenté dans l'Algorithme 9. Ainsi, le temps d'exécution ne dépend plus des données traitées. Cette contre-mesure permet de protéger contre les attaques temporelles.

— Ajouter des opérations factices lorsque le coefficient traité vaut 0. Cette solution requiert un élément de $\mathbb{F}_{2^m}$ supplémentaire pour le stockage des calculs intermédiaires. Elle permet d'avoir une amplitude de consommation de courant similaire en fonction du coefficient traité.

— Randomiser le traitement des coefficients dans chaque colonne. Un adversaire n'est alors plus en mesure de retrouver les indices des lignes traitées. En considérant la première colonne de la matrice, un adversaire aura n possibilités pour le pivot et $(n-1)!$ possibilités pour le traitement des autres lignes de la colonne. Ainsi, la complexité de l'attaque suite à l'analyse de la consommation de courant serait de $(n!)^m$. Par exemple, avec ROLLO-I-128, la complexité serait de $(47!)^{79}$, ce qui correspond environ à 2^{15591} possibilités.

Afin d'assurer une meilleure protection, il est possible de combiner les deux dernières contre-mesures, comme présenté dans l'Algorithme 11.

Algorithme 11 : Pivot de Gauss avec l'ajout d'opérations factices

Entrées : Une matrice $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : Une matrice $\mathbf{S}$ sous forme échelonnée et le rang de la matrice

```
1  Rang, dummy ← 0
2  pour i allant de 0 à m − 1 faire
3      pour j allant de i à n − 1 faire
4          j_rand = (j + random()) mod (n − i) ;                 // randomisation
5          si s_{j_rand,i} = 1 alors
6              s_i ↔ s_{j_rand} ;                  // la ligne j_rand est la ligne pivot
7              Rang ← Rang +1
8              break
9      pour k allant de ligne i + 1 à n faire
10         k_rand = (k + random()) mod (n − k) ;                 // randomisation
11         s_k ↔ s_{k_rand} ;                      // s_{k_rand} devient la ligne traitée
12         si s_{k,i} = 1 alors
13             s_k ← s_k + s_i
14         sinon
15             dummy ← s_k + s_i ;                 // si le coefficient dominant est nul
16 retourner (M, Rang)
```

2.3.2 Pour les implantations en temps constant

Les contre-mesures pour les implantations en temps constant vont consister à protéger principalement des attaques par analyse simple de la consommation.
Concernant l'implantation de référence en temps constant, une solution consiste à amplifier le bruit afin de réduire la différence de consommation entre la multiplication d'un mot par 0 ou par 1. Pour cela, nous pouvons masquer les coefficients traités. Dans la première boucle intérieure **pour**, nous partageons la ligne pivot en deux parties. Ainsi, pour chaque itération, nous calculons

$$s_{ligne_pivot} = s_{1ligne_pivot} \oplus s_{2ligne_pivot},$$

avec $s_{1ligne_pivot}, s_{2ligne_pivot} \in \mathbb{F}_{2^m}$. En définissant $masque\mathsf{I} = \neg(masque - 1)$, la variable tmp, à la ligne 6 de l'Algorithme 9, est alors calculée comme suit

$$tmp_{masque} = (masque\mathsf{I} \wedge (s_i \oplus s_{2ligne_pivot})) \vee (\neg masque\mathsf{I} \wedge s_{2ligne_pivot}).$$

L'opération réalisée sur la ligne pivot devient ainsi

$$s_{ligne_pivot} = s_{1ligne_pivot} \oplus tmp.$$

Dans le cas où $i \leq ligne_pivot$, nous avons

$$dummy = s_{1ligne_pivot} \oplus tmp_{masque}.$$

Les mêmes opérations sont effectuées dans la seconde boucle **pour** en remplaçant la ligne pivot par la ligne en cours de traitement s_i.

Équivalence entre les deux schémas :

$$\begin{aligned}
s_{ligne_pivot} &= s_{1ligne_pivot} \oplus tmp_{masque} \\
&= s_{1ligne_pivot} \oplus (masque\mathsf{I} \wedge (s_i \oplus s_{2ligne_pivot})) \vee (\neg masque\mathsf{I} \wedge s_{2ligne_pivot}) \\
&= s_{1ligne_pivot} \oplus \begin{cases} s_i \oplus s_{2ligne_pivot} & \text{si } masque = 1 \\ s_{2ligne_pivot} & \text{sinon} \end{cases} \\
&= \begin{cases} s_{ligne_pivot} \oplus s_i & \text{si } masque = 1 \\ s_{ligne_pivot} & \text{sinon} \end{cases} \\
&= s_{ligne_pivot} \oplus tmp,
\end{aligned}$$

avec $tmp = masque \otimes s_i$.

Avec cette contre-mesure, nous ne pourrons plus différencier des motifs lorsque le *masque* vaudra 0 ou 1. En effet, nous réalisons toujours les mêmes opérations, c'est-à-dire deux opérations binaires AND entre un mot non nul et un mot nul.

La seconde contre-mesure est basée sur un traitement aléatoire des colonnes, semblable à celle proposée dans l'Algorithme 11. Cependant, afin de conserver le temps constant, nous pré-mélangeons les indices des coefficients de la matrice contenus dans une liste. Ce mélange peut se réaliser en utilisant un algorithme qui génère une permutation aléatoire sur un ensemble fini telle que la méthode de Fisher-Yates décrite dans l'Algorithme 18. Pour l'implantation de référence en temps constant, une liste contient les indices des coefficients de la matrice mélangés avant chaque boucle intérieure **pour**. Puis, à chaque itération la ligne pivot est choisie de manière aléatoire et les coefficients de la colonne sont traités suivant l'ordre des indices de la liste mélangés. Cette contre-mesure est présentée dans l'Algorithme 12. Les indices étant mélangés avant chaque boucle intérieure **pour**, nous évitons ainsi toute corrélation entre les valeurs de *masque* de la première boucle **pour** (ligne 8) et celle de la deuxième **pour** (ligne 19).

Algorithme 12 : Pivot de Gauss en temps constant avec randomisation

Entrées : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ sous forme systématique

```
1  masque = dimension = 0
2  L ← tableau contenant tous les indices 0, ..., n − 1
3  pour j allant de 0 à m − 1 faire
4      ligne_pivot = min(dimension, n − 1)
5      randpivot = random(ligne_pivot + 1, n − 1)
6      échanger s_ligne_pivot et s_randpivot ;   // choix aléatoire de la ligne pivot
7      L = FisherYates(L)
8      pour i = 0, ···, n − 1 faire
9          randindice = L[i]
10         masque = s_{ligne_pivot,j} ⊕ s_{randindice,j}
11         tmp = masque ⊗ s_randindice
12         si randindice > ligne_pivot alors
13             s_ligne_pivot = s_ligne_pivot ⊕ tmp
14         sinon
15             dummy = s_ligne_pivot ⊕ tmp
16     L = FisherYates(L)
17     pour i = 0, ···, n − 1 faire
18         randindice = L[i]
19         si randindice ≠ j alors
20             masque = s_{randindice,j}
21             tmp = masque ⊗ s_ligne_pivot
22             si dimension < n alors
23                 s_randindice = s_randindice ⊕ tmp
24             sinon
25                 dummy = s_randindice ⊕ tmp
26     dimension = dimension + s_{ligne_pivot,i}
```

Pour l'implantation de la version GitHub, nous pouvons appliquer le même principe, comme présenté dans l'Algorithme 13. Avant chaque itération sur les lignes, la ligne pivot est choisie de manière aléatoire et les indices des coefficients, contenus dans une liste, sont mélangés avant chaque début de traitement sur les colonnes. Ce mélange peut également se faire en utilisant la méthode de Fisher-Yates (Algorithme 18) sous condition d'une bonne implantation de ce dernier afin d'éviter les sources de biais connues. Pour les deux implantations en temps constant, cette contre-mesure nécessite le stockage en mémoire d'un tableau contenant les indices des lignes de la matrice, soit n octets.
Etant donnée une liste de n éléments à mélanger, la méthode de Fisher-Yates consiste à générer à chaque itération un nombre aléatoire j tel que $1 \leq j \leq n - k$, où k est le nombre d'éléments déjà traités. Puis, à chaque itération, deux éléments de la liste sont échangés dans un ordre décroissant, le dernier élément de la liste est échangé avec le j^{e} élément de la liste, puis l'avant-dernier sera échangé et ainsi de suite

jusqu'à atteindre le premier élément de la liste. La complexité de l'algorithme de Fisher-Yates est de $\mathcal{O}(n)$.

Algorithme 13 : Réduction de la matrice sous forme échelonnée avec randomisation

Entrées : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$
Résultat : $\mathbf{S} \in \mathcal{M}_{n,m}(\mathbb{F}_2)$ sous forme échelonnée et son $rang = pivot$

```
1  pivot = 0
2  L ← tableau contenant les indices 0, ..., n − 1
3  pour j = 0, ⋯, m − 1 faire
4      randpivot = random(pivot + 1, n − 1)
5      échanger s_pivot et s_randpivot
6      L = FisherYates(L)
7      pour i = 0, ⋯, n − 1 faire
8          randindice = L[i]
9          si s_{pivot,j} == 0 alors
10             masque1 = 1
11         sinon
12             masque1 = 0
13         si s_{randindice,j} == 1 alors
14             masque2 = 1
15         sinon
16             masque2 = 0
17         si randindice ≥ pivot alors
18             masque3 = 1
19         sinon
20             masque3 = 0
21         s_pivot ← s_pivot ⊕ s_i ∧ (masque1 ∧ (masque2 ∧ masque3))
22         s_randindice ← s_randindice ⊕ s_pivot ∧ (masque2 ∧ masque3)
23     si s_{pivot,j} = 1 et pivot < n alors
24         pivot = pivot + 1
```

De même que pour la contre-mesure du traitement aléatoire des colonnes proposées dans l'Algorithme 11, nous pourrons toujours différencier le traitement lorsque les masques sont nuls ou non nuls. Cependant, une fois les valeurs retrouvées, nous aurons également par colonne $n!$ combinaisons possibles pour trouver l'ordre des éléments. Ce qui conduit à $(n!)^m$ combinaisons pour toute la matrice. Ainsi, avec les paramètres de la nouvelle version, ROLLO-I-128, nous aurons $(83!)^{67}$ possibilités, soit une complexité d'environ 2^{27731}.

2.3.3 Résultats expérimentaux

Pour les mesures de consommation de courant, nous avons conservé la même installation que pour les attaques.

Pour l'implantation du pivot de Gauss standard

La figure 2.12 présente la trace du pivot de Gauss de l'implantation pour la standardisation avec les opérations factices. L'opération binaire XOR étant effectuée dans le cas où les coefficients valent 1 ou 0, nous observons une uniformisation de la trace.

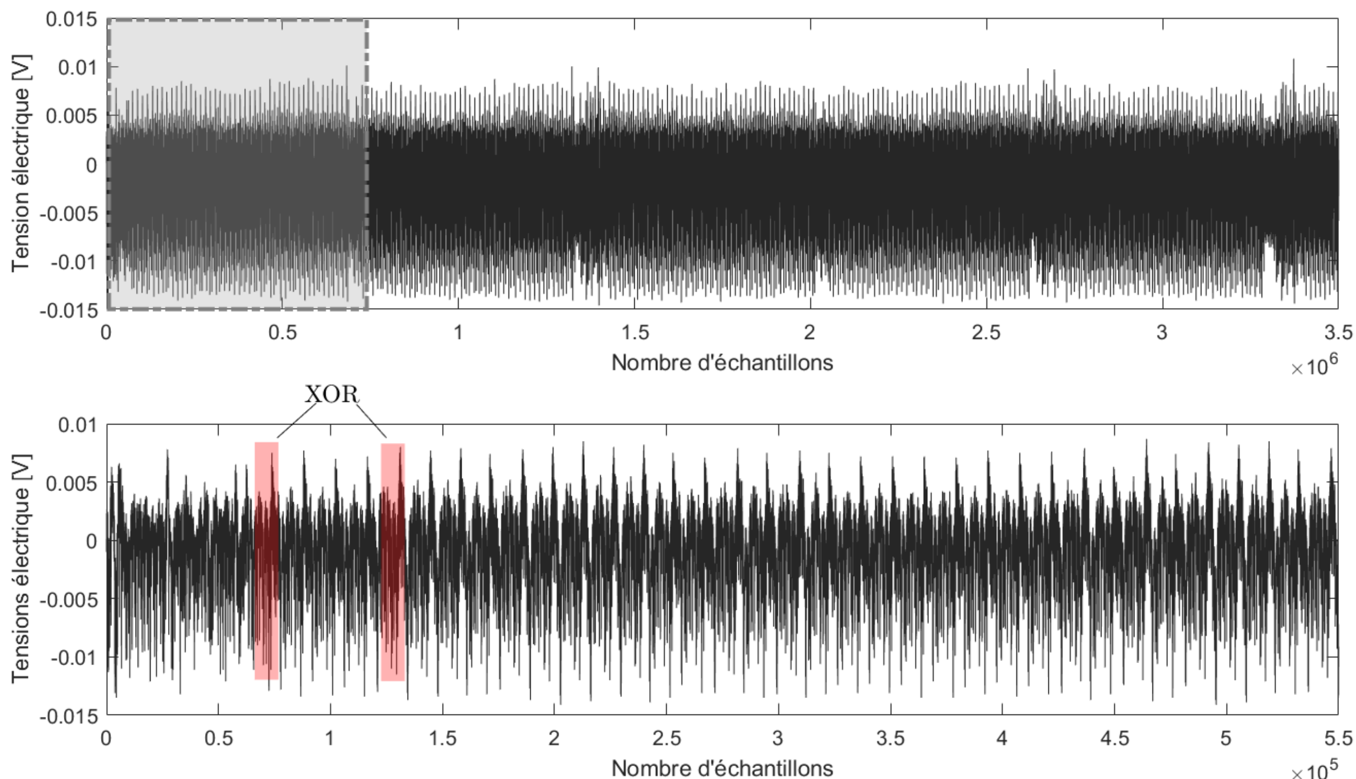


Figure 2.12 – Partie de la trace de consommation de l'implantation du pivot de Gauss avec l'ajout d'opérations factices pour la standardisation avec un zoom sur la première colonne

Le Tableau 2.8 présente l'impact [5] des contre-mesures sur le temps d'exécution du procédé de décapsulation. L'ajout d'opérations factices impacte le temps d'exécution d'environ 40%. Le traitement aléatoire des colonnes a un impact plus significatif, l'augmentant d'environ 50%. Cela peut s'expliquer par l'échange de lignes à chaque itération. Si l'espace mémoire le permet, la contre-mesure utilisant la méthode de Fisher-Yates, présentée précédemment, sera préférée.

5. dépendant de la carte et du TRNG utilisés

Sécurité		Décapsulation		
		randomisation	opérations factices	sans contre-mesure
128	cycles ($\times 10^6$)	8,09	5,84	4,31
	ms	161,8	116,6	86,3
192	cycles ($\times 10^6$)	17,01	11,23	7,8
	ms	340,2	224,6	156
256	cycles ($\times 10^6$)	32,45	21,62	15,54
	ms	649	432,4	310,8

TABLE 2.8 – Temps d'exécution de la décapsulation avec et sans contre-mesure

Pour les implantations en temps constant
La Figure 2.13 présente la trace de consommation de la première boucle intérieure **pour** (ligne 4 - Algorithme 9) avec le masquage présentée dans la sous-section 2.3.2. Comme nous pouvons l'observer sur la trace de consommation, nous ne sommes plus en mesure de différencier des motifs lorsque nous calculons la valeur de *tmp* en fonction de *masque*. En effet, si *masque* = 0 ou *masque* = 1, le buffer *tmp* sera toujours calculé à partir des mêmes opérations.

Le Tableau 2.9 présente l'impact [6] des contre-mesures sur le temps d'exécution sur les méthodes du pivot de Gauss en temps constant. La contre-mesure basée sur le masquage de la multiplication a un impact plus important sur le temps d'exécution que la contre-mesure avec le traitement aléatoire des colonnes, ce qui est notamment dû au partage des lignes à chaque itération.

	Référence			GitHub	
	masquage	randomisation	sans contre-mesure	randomisation	sans contre-mesure
cycles ($\times 10^6$)	3,15	2,5	1,82	2,9	2,22
ms	63	49,58	36,46	58,84	44,57

TABLE 2.9 – Temps d'exécution des implantations du pivot de Gauss en temps constant pour les paramètres de ROLLO-I-128

6. dépendant de la carte et du TRNG utilisés

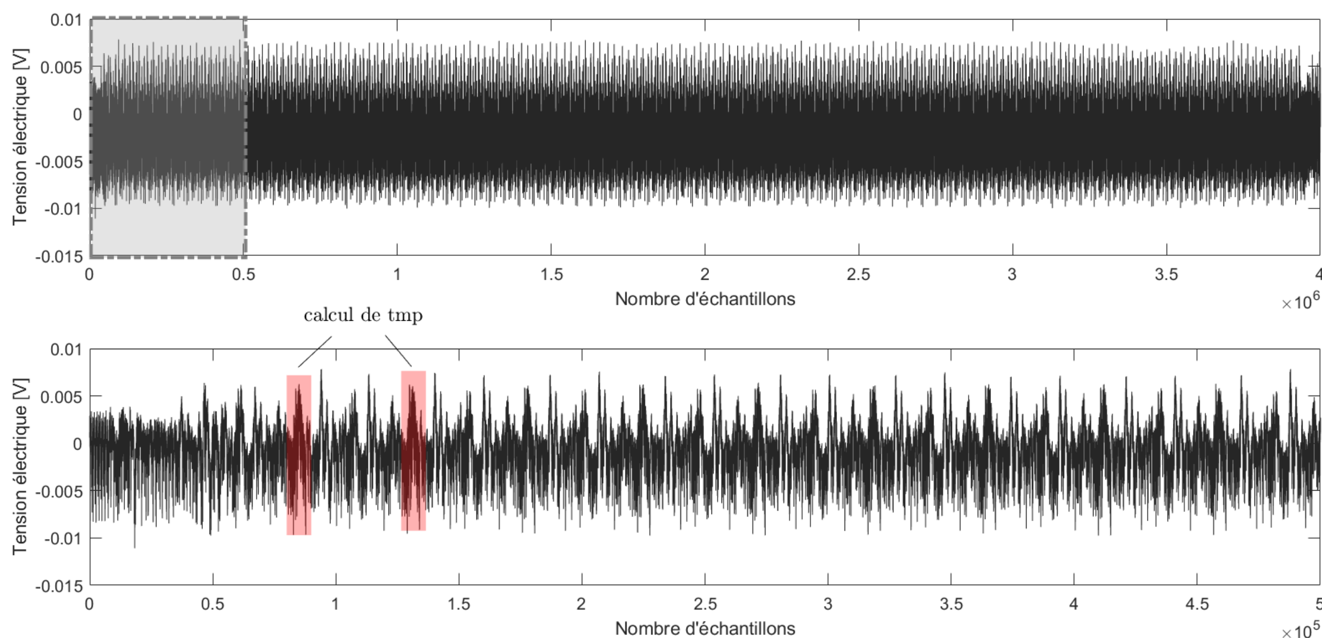


FIGURE 2.13 – Partie de la trace de consommation de l'implantation du pivot de Gauss en temps constant avec masquage de la multiplication et du XOR

2.4 Conclusion

Nous avons montré qu'il était possible d'intégrer le cryptosystème ROLLO-I dans un microcontrôleur à ressources restreintes. Il est également possible de bénéficier d'une accélération des opérations sur $\mathbb{F}_{2^m}$ en utilisant un coprocesseur cryptographique existant. Cela permet ainsi de déduire qu'il serait envisageable d'intégrer le cryptosystème dans des produits actuellement déployés et que les modifications en termes d'espace mémoire et de temps d'exécution restent raisonnables pour un cryptosystème post-quantique. Nous avons également montré comment, par des analyses de la consommation de courant, il était possible de retrouver le syndrome, nous permettant ainsi de retrouver la clé privée utilisée ou le secret partagé dans le cas des implantations en temps constant. Afin de sécuriser les implantations, nous avons proposé différentes contre-mesures, certaines permettant d'ajouter du bruit telles que le masquage ou l'ajout d'opérations factices, d'autres permettant de modifier l'ordre des coefficients dans la matrice, par un traitement aléatoire des colonnes.
Pour la suite, nous souhaiterions réaliser l'analyse de la fonction Karatsuba utilisée dans l'implantation de ROLLO. Comme montré dans la Figure 2.4, la multiplication dans le corps $\mathbb{F}_{2^m}[Z]/(P_n)$ fait interagir une partie de la clé privée $\mathbf{x}$ et le chiffré $\mathbf{c}$, connu de l'adversaire, nous pouvons ainsi envisager une attaque par canaux auxiliaires sur cette opération afin de retrouver la partie secrète. De même, il serait intéressant de poursuivre l'optimisation des opérations sur $\mathbb{F}_{2^m}[Z]/(P_n)$ telle que l'inversion lors de la génération des clés.

Deuxième partie

Réseaux euclidiens

Chapitre 3

Étude de NTRU

Sommaire

3.1 Attaque par canaux auxiliaires sur NTRU 86
3.1.1 Attaques existantes 86
3.1.2 Opération ciblée 87
3.1.3 Attaques profilées 88
3.2 Proposition de contre-mesures 95
3.3 Conclusion 97

Dans ce chapitre, nous étudions la sécurité du cryptosystème NTRU face aux attaques profilées. Le candidat NTRU semblait dès le début de la standardisation un candidat prometteur grâce à son ancienneté. Il était donc intéressant d'avoir une analyse de ce candidat dans le cas d'une future implantation. Pour cette étude, nous nous concentrons sur le protocole de chiffrement NTRU, détaillé dans la Figure 3.1 (suivant les notations données dans la Figure 1), soumis au second tour de la standardisation, pour lequel les auteurs ont recommandé deux ensembles de paramètres identifiés par NTRU-HPS (Hofftein, Pipher et Silverman) et NTRU-HRSS (Hülsing, Rijneveld, Schanck et Schwabe), et donnés dans le Tableau 3.1.

	NTRU-HPS		NTRU-HRSS
n	509	821	701
q	2048	4096	8192
p	3	3	3
Clé privée	935	1592	1452
Clé publique	699	1230	1138
Chiffré	699	1230	1138

Table 3.1 – Paramètres pour la soumission NTRU

Ces paramètres sont conservés pour le troisième tour de la standardisation.

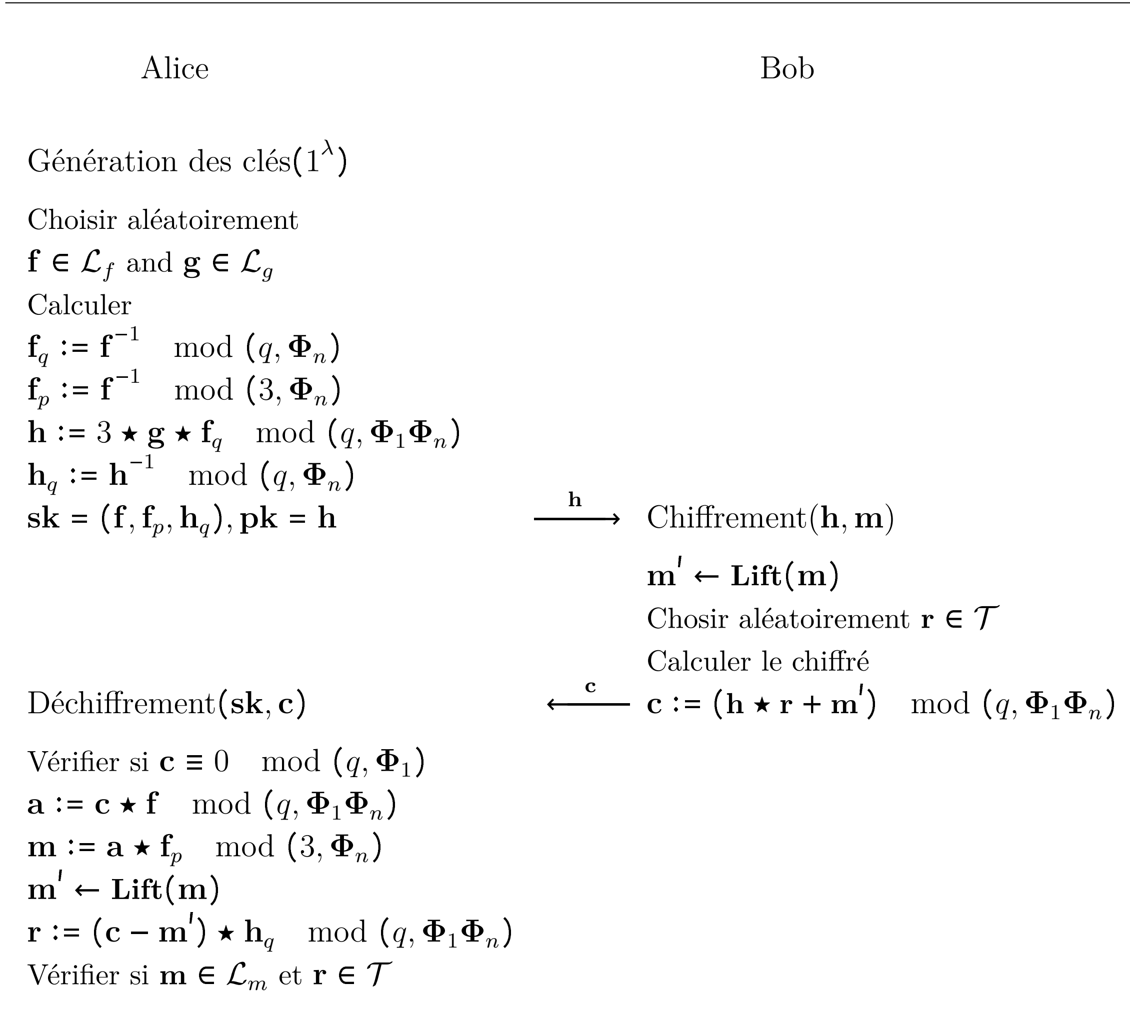


Figure 3.1 – Chiffrement NTRU

Pourquoi nous retrouvons bien le message clair lors du déchiffrement

$$
\begin{aligned}
\mathbf{a} &\equiv \mathbf{c} \star \mathbf{f} \mod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\
&\equiv (\mathbf{h} \star \mathbf{r} + \mathbf{m}') \star \mathbf{f} \mod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\
&\equiv 3 \star \mathbf{g} \star \mathbf{F}_q \star \mathbf{r} \star \mathbf{f} + \mathbf{m}' \star \mathbf{f} \mod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n) \\
&\equiv 3 \star \mathbf{g} \star \mathbf{r} + \mathbf{m}' \star \mathbf{f} \\
\mathbf{a} \star \mathbf{F}_p &\equiv (3 \star \mathbf{g} \star \mathbf{r} + \mathbf{m}' \star \mathbf{f}) \star \mathbf{F}_p \mod (3, \mathbf{\Phi}_n) \\
&\equiv \mathbf{m} \mod (3, \mathbf{\Phi}_n).
\end{aligned}
$$

Le Tableau 3.2 présente la définition des ensembles d'appartenance des clés en fonction des paramètres utilisés dans les cryptosystèmes de la soumission NTRU. Notons

que $\mathcal{T}$ est l'ensemble des polynômes ternaires non nuls dont les degrés sont au plus $n-2$. Un polynôme ternaire est un polynôme dont les coefficients sont dans $\{-1, 0, 1\}$.

	NTRU-HPS	NTRU-HRSS
$\mathcal{L}_f$	$\mathcal{T}$	$\mathcal{T}_+$
$\mathcal{L}_g$	$\mathcal{T}(q/8-2)$	$\{\mathbf{\Phi}_1.\mathbf{v}, \mathbf{v} \in \mathcal{T}_+\}$
$\mathcal{L}_m$	$\mathcal{T}(q/8-2)$	$\mathcal{T}$
Lift	$\mathbf{m} \to \mathbf{m}$	$\mathbf{m} \to \mathbf{\Phi}_1.\mathcal{S}_3(\mathbf{m}/\mathbf{\Phi}_1)$

Table 3.2 – Ensembles et fonction utilisés suivant les paramètres

Ainsi, nous pouvons voir que la clé privée $\mathbf{f}$ est un polynôme ternaire. D'un point de vue implantation, les coefficients de $\mathbf{f}$ s'écrivent sur 16 bits. En effet, considérons les paramètres de NTRU-HRSS, $\mathbf{f} \in \mathcal{R}_q$, alors $-1 \mod q = 8191$. Donc les coefficients de la clé correspondent à `0x1FFF`, `0x0000` et `0x0001`. Dans la suite, afin de simplifier les notations, nous notons ces coefficients $-1, 0$ et 1.

3.1 Attaque par canaux auxiliaires sur NTRU

3.1.1 Attaques existantes

Le cryptosystème NTRU étant étudié depuis de nombreuses années, plusieurs implantations embarquées ont déjà été réalisées, menant ainsi à l'analyse de la sécurité de ces implantations :

- 2007 : Silverman et Whyte [SW07] réalisent des attaques temporelles basées sur le nombre d'appels de la fonction de hachage durant le procédé de déchiffrement de NTRUEncrypt ;
- 2010 : Lee et al. [LESCH10] démontrent la vulnérabilité d'une implantation naïve de la multiplication utilisée dans le schéma NTRU en réalisant une attaque simple de la consommation (SPA) et une attaque par corrélation (CPA) incluant une attaque de second ordre. Ils proposent alors trois contre-mesures contre leur attaque SPA consistant en un masquage du chiffré $\mathbf{c} = \mathbf{c} + \mathbf{r}$, avec $\mathbf{r} \in \mathcal{R}_q$, la randomisation de la clé privée (en mélangeant les positions) et l'initialisation de buffers temporaires avec des valeurs aléatoires ;
- 2013 : Zheng et al. [ZWW13] trouvent une attaque par collision de premier ordre sur l'implantation de la même précédente multiplication et cassent les trois contre-mesures proposées dans [LESCH10]. Les auteurs proposent alors de nouvelles contre-mesures résistantes à leur attaque telles que l'ajout d'opérations factices, l'ajout de délai aléatoire et une randomisation des éléments : multiplication de la clé privée $\mathbf{f}$ par x^i avec i un nombre aléatoire et multiplication du chiffré $\mathbf{c}$ par x^{n-i}. ;
- 2017 : Mahmoud et al. [MNY17] proposent une nouvelle contre-mesure pour une implantation sur FPGA qui consiste à séparer le chiffré en deux parties

et exécuter la multiplication en parallèle entre chaque partie du chiffré et la clé privée.

- 2018 : An et al. [AKJ+18] analysent deux implantations : la soumission du premier tour NTRUEncrypt [CHWZ17] et une implantation open-source de NTRU. Ils retrouvent toute la clé privée durant le procédé de déchiffrement avec l'analyse d'une seule trace (STA). Ils font l'observation que les multiplications par -1 forment des pics plus élevés sur la trace et que les multiplications par 0 sont représentées par les pics les moins élevés sur la trace. Ce qui est dû au fait que le poids de Hamming de -1 est supérieur aux autres.
- 2019 : Huang et al. [HCY19] réalisent une attaque contre la multiplication présente dans NTRU Prime (soumission du second tour de la standardisation) et adaptent les contre-mesures proposées dans [LESCH10] à NTRU Prime : accès randomisé aux coefficients des polynômes, initialisation des buffers ou encore masquage d'ordre 1 en ajoutant un polynôme aléatoire à la clé privée et au chiffré. La deuxième contre-mesure revient à initialiser avec une valeur aléatoire le polynôme $\mathbf{a}$ dans l'Algorithme 14 ligne 2 et à soustraire ces valeurs à la fin de l'algorithme ;
- 2020 : Sim et al. [SKL+20] attaquent l'encodage des messages dans le procédé d'encapsulation de plusieurs cryptosystèmes dont NTRU, permettant ainsi d'obtenir le secret partagé.

3.1.2 Opération ciblée

Comme nous pouvons le voir dans la Figure 3.1, durant le déchiffrement, nous opérons une multiplication polynomiale entre une partie de la clé privée $\mathbf{f}$ et le chiffré $\mathbf{c}$. Il est donc normal que cette opération ait pu être la cible de différentes attaques par canaux auxiliaires. Cependant, plusieurs attaques ciblent des précédentes implantations de NTRU, antérieures à la standardisation lancée par le NIST. Seules les attaques proposées dans [AKJ+18] et [SKL+20] concernent des implantations soumises à la standardisation.

Algorithme 14 : Produit de convolution

Entrées : n un entier positif, la clé secrète $\mathbf{f} = (f_0, \ldots, f_{n-1})$, le chiffré $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathcal{R}_q$

Output : $\mathbf{a} = \mathbf{f} \star \mathbf{c}$ in $\mathcal{R}_q$

```
1 pour i allant de 0 à n − 1 faire
2     a_i ← 0
3     pour j allant de 1 à n − i − 1 faire
4         a_i ← a_i + f_{i+j} × c_{n−j}
5     pour j allant de 0 à i faire
6         a_i ← a_i + f_{i−j} × c_j
7     a_i ← a_i mod q
8 retourner a
```

Dans la soumission NTRU, la multiplication est implantée suivant le produit de

convolution décrit dans l'Algorithme 14. Nous avons décidé d'analyser les attaques profilées sur l'implantation de cette opération. En effet, la clé privée $\mathbf{f}$ a ses coefficients dans l'ensemble $\{-1, 0, 1\}$. Ainsi, il est envisageable de créer un modèle de consommation pour ces trois valeurs possibles.

3.1.3 Attaques profilées

Nous avons décidé de réaliser une première étude sur des traces simulées, cela nous a notamment permis de tester les attaques à base de réseaux de neurones. Puis, nous avons étudié une attaque template avec des traces réelles car cela demandait moins de ressources que les attaques par apprentissage profond.

3.1.3.1 Simulation

Pour l'attaque, nous avons généré la clé privée et le chiffré à partir de l'implantation des paramètres de NTRU-HRSS de la soumission (voir Table 3.1) et avons implanté le produit de convolution dans la carte présentée dans la section 1.2.2.
L'attaque par apprentissage profond s'est porté sur les traces acquises à partir d'un simulateur de coeur CORTEX-M3 basé sur le modèle de la distance de Hamming et développé à l'entreprise. Le choix du simulateur est très pratique dans l'analyse de sécurité d'une implantation car il permet :

- l'alignement parfait des traces et retire ainsi le besoin de prétraitement du signal ;
- l'absence de bruit, rendant ainsi toute fuite de consommation parfaitement visible ;
- la détection du secret si ce dernier apparaît en clair dans un registre à un instant t car le modèle de fuites prend, à chaque cycle, les changements dans les registres (en distance de Hamming).

Nous avons alors fait le choix de tester une attaque à base de réseaux de neurones qui sont des attaques profilées puissantes et de plus en plus utilisées. Cependant, ces attaques demandent souvent un nombre conséquent de ressources, aussi bien en termes de capacité de calculs qu'en termes de nombre de traces composant la base de données pour la phase d'entraînement. Le fait d'utiliser des traces simulées permet alors de réduire grandement ces ressources et d'avoir une première simulation d'une attaque profilée.

Apprentissage profond [GBC16]. L'apprentissage profond est une branche de l'apprentissage automatique qui utilise des réseaux de neurones artificiels (ANN pour *Artificial Neural Network*) pour la classification de tâches. En d'autres termes, durant la phase d'apprentissage, un réseau de neurones est entraîné avec des vecteurs d'entrées $X_k = (x_{k,1}, \dots, x_{k,n}) \in \mathbb{R}^n$ et des étiquettes correspondantes $Y_k = (y_{k,1}, \dots, y_{k,m}) \in \mathbb{R}^m$. À la suite de cette phase, étant donnée une nouvelle entrée X', le réseau de neurones entraîné doit être capable d'assigner la bonne étiquette Y'.
Nous basons notre étude sur un type spécifique de réseaux de neurones : le réseau de neurones convolutifs (CNN pour *Convolutional Neural Network*), présenté

dans la Figure 3.2, ce dernier étant très utilisé pour les attaques par canaux auxiliaires [MPP16, CDP17]. Durant la phase d'apprentissage, des traces de consommation sont données en entrée du CNN, chaque trace est associée à une clé privée. Cette valeur de clé devient alors l'étiquette assignée à l'entrée. Ainsi, dans la phase d'attaque, en donnant une nouvelle trace au réseau de neurones entraîné, un adversaire sera en mesure de retrouver la valeur de la clé privée correspondante.

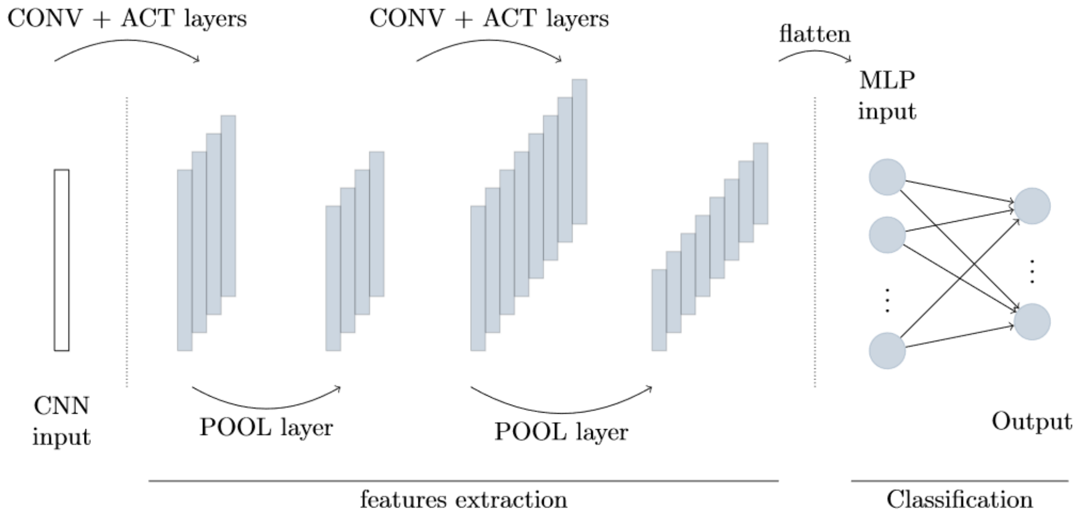


FIGURE 3.2 – Architecture d'un réseau de neurones convolutifs. La couche de sortie utilise la fonction d'activation Softmax (définie dans l'Équation 3.2).

Architecture d'un CNN
Un CNN est formé de plusieurs blocs de traitement qui permettent l'extraction de caractéristiques (*features*) discriminant la classe d'appartenance des traces. Chaque bloc est composé de différentes couches :

— les couches de convolution (CONV) composées de filtres caractérisés par leur nombre et leur taille. Ces couches réalisent un produit de convolution entre les entrées et ces filtres. Ces derniers sont modifiés automatiquement durant la phase d'entraînement afin d'extraire les informations les plus pertinentes ;
— les couches d'activation (ACT) qui appliquent une fonction non-linéaire sur les données de sortie de la couche de convolution permettent ainsi l'extraction de caractéristiques plus complexes. Dans notre étude, nous avons utilisé la fonction d'activation SELU (*Sacled Exponential Linear Unit*)

$$SELU(x) = \lambda \begin{cases} x & \text{if } x > 0 \\ \alpha e^x - \alpha & \text{if } x \leq 0 \end{cases} ;$$

— les couches de pooling (POOL) permettent de réduire la dimension de l'entrée tout en conservant une représentation approximativement invariante. Par exemple, si nous appliquons l'opération de la moyenne dans cette couche (*average pooling*), cela revient à calculer la moyenne des valeurs d'entrée dans une fenêtre de taille donnée.

Plusieurs blocs de traitement peuvent se succéder jusqu'à atteindre un nombre de valeurs en sortie raisonnable, contenant ainsi le plus d'information. À la suite de l'extraction des caractéristiques, l'étape de la classification des données se réalise par le biais du bloc de classification constitué également de plusieurs couches consistant en un perceptron multicouche (MLP pour *Multilayer Percreptron*) :

— les couches entièrement connectées (FC) qui sont constituées de nœuds appelés neurones ; chaque neurone de la couche i est connecté à tous les neurones de la couche suivante $i+1$. Chaque neurone a son propre poids $w_i i, i+1$ qui est modifié durant la phase d'entraînement par la technique de rétro propagation du gradient. Cette technique permet l'ajustement automatique des poids en fonction de la valeur du gradient de la fonction de perte.

— la couche de perte (loss) permet de calculer l'erreur entre la prévision du réseau et la valeur réelle de l'étiquette. Pour notre étude, nous considérons l'entropie croisée (*Cross-entropy*) définie par

$$H_{Y_k}(\hat{Y}_k) = -\sum_{i=1}^{m} y_{k,i} \cdot log(\hat{y}_{k,i}).$$

D'un point de vue théorique, l'utilisation du CNN dans l'analyse des traces de consommation permet d'éviter de réaliser du prétraitement sur les traces tel que la synchronisation des traces ou la recherche de points d'intérêt (comme pour les attaques par template). En pratique, faire apprendre la trace de consommation entière de millions de poids engendrerait l'utilisation d'un réseau de neurones complexes et le temps de calculs serait également conséquent. Afin de pallier ce problème, nous pouvons réaliser une recherche de POIs et entraîner le réseau de neurones avec une zone de points contenant des POIs.

Attaque du cryptosystème NTRU. L'attaque consiste à retrouver récursivement les coefficients de la clé privée $\mathbf{f} = (f_0, \ldots, f_{n-1})$ avec $f_i \in \mathcal{K} = \{-1, 0, 1\}$. Pour l'explication de l'attaque, supposons que nous attaquions le coefficient f_1.

Phase d'apprentissage

Pour modéliser les fuites d'information, nous définissons les deux ensembles requis pour entraîner le CNN :

— une base de données d'entrée X composée de N traces de consommation T_{i,f_1} avec $0 \leq i < N$ et $f_1 \in \mathcal{K}$.
Ainsi,

$$X = \bigcup_{i=1}^{N} (T_{i,-1} \cup T_{i,0} \cup T_{i,1});$$

— une base de sortie Y composée d'étiquettes correspondant à l'ensemble des clés $f_1 \in \mathcal{K}$ associées à chaque trace T_{i,f_1} pour $0 \leq i < N$. Nous définissons les étiquettes par $Y_{i,-1} = (1, 0, 0)$ lorsque $f_1 = -1$, $Y_{i,0} = (0, 1, 0)$ quand $f_1 = 0$ et $Y_{i,1} = (0, 0, 1)$ quand $f_1 = 1$. Suivant ces notations, l'ensemble des étiquettes pour la phase d'entraînement correspond à

$$Y = \bigcup_{i=1}^{N} (Y_{i,-1} \cup Y_{i,0} \cup Y_{i,1}).$$

Phase d'attaque
Nous réalisons l'acquisition de nouvelles traces dont les clés privées sont inconnues et donnons ces traces en entrée du CNN. Le réseau de neurones prédit alors un score $\hat{Y}_{f_1}$ pour chaque trace de consommation. Le score le plus élevé correspond alors à la clé correcte.

Résultats
Afin de réaliser l'attaque, nous avons décidé d'utiliser la bibliothèque open-source Keras [Ker] qui permet une construction rapide du CNN. Cependant, comme signalé précédemment, les traces étant souvent composées de millions de points, les ressources requises par le CNN pour apprendre toute la trace seraient vraiment conséquentes, en termes de mémoire ou de puissance de calcul. Ainsi, nous avons décidé de nous focaliser sur une zone de points contenant des informations relatives au coefficient de clé recherché. Pour trouver cette zone spécifique, nous cherchons les POIs en utilisant la méthode de la somme des différences (SOSD). Avec cette méthode, nous pouvons trouver des points pertinents en choisissant ceux qui maximisent la somme suivante

$$\sum_{k,l \in \mathcal{K}} (T_k - T_l) \text{ pour } k \geq l, \tag{3.1}$$

où T_k est la moyenne des traces lorsque la variable secrète appartient à la classe $k \in \mathcal{K}$. Puisque nous utilisons des traces simulées, nous avons seulement acquis 5 traces par classe et appliqué la méthode SOSD à ces trois ensembles de traces. Le résultat est montré dans la Figure 3.3.

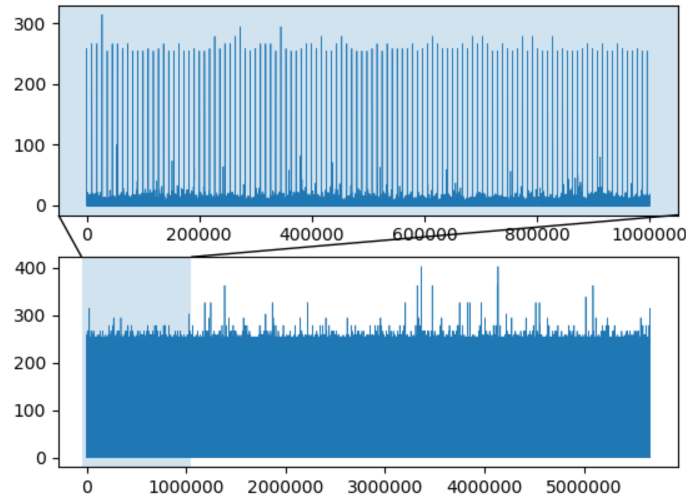


FIGURE 3.3 – Méthode SOSD appliquée sur des traces dont un coefficient a été modifié (en abscisse échantillonnage de points et en ordonnée, le résultat de la somme présentée à l'équation 3.1).

Nous cherchons donc les points maximisant la somme de l'équation 3.1. Si nous fixons un seuil à 200 dans la Figure 3.3, nous observons qu'un large échantillon de

points peut être sélectionné. À la suite d'un zoom sur une zone, nous remarquons que les pics sont séparés de manière régulière. Ces pics correspondent aux itérations de la première boucle interne **pour** dans le produit de convolution (Algorithme 14 - ligne 4). En effet, dans l'algorithme, pour $j = 1$, $(n-1)$ coefficients de la clé privée sont multipliés par $(n-2)$ coefficients du texte chiffré, le produit est reporté dans la même variable a_0. Les classes étant composées des mêmes clés privées avec seulement le premier coefficient qui diffère, la modification du premier coefficient impacte donc les résultats suivants. Cependant, nous ciblons la première multiplication. Pour cela, nous choisissons une zone de 60 points contenant au moins le premier pic.

Pour la phase d'apprentissage, nous avons acquis 1000 traces simulées du produit de convolution entre des chiffrés $\mathbf{c} \in \mathcal{R}_q$ vérifiant la condition $\mathbf{c} \equiv 0 \mod (q, \mathbf{\Phi}_1)$ et une clé fixe $\mathbf{f}$. Nous avons répété ce procédé deux autres fois en faisant varier le coefficient de la clé ciblée avec les deux autres valeurs possibles. Nous avons ensuite extrait sur chaque trace une zone de 60 contenant au moins le premier pic. Chaque trace est ainsi réduite à 60 points et nous notons

$$T_{i,f_1} = (p_{i,1}, \cdots, p_{i,40}),$$

avec $1 \leq i \leq 3000$ et $p_{i,j}$ correspondant au point j dans la zone sélectionnée de la trace i.
Nous obtenons finalement l'ensemble

$$X = \bigcup_{i=1}^{1000} (T_{i,-1} \cup T_{i,0} \cup T_{i,1}),$$

Et l'ensemble des étiquettes pour la phase d'entraînement correspond à

$$Y = \bigcup_{i=1}^{1000} (Y_{i,-1} \cup Y_{i,0} \cup Y_{i,1}).$$

Le CNN est entraîné avec les deux ensembles précédemment définis. Il est constitué d'une couche de convolution avec 8 filtres de taille 8 suivie d'une couche d'activation SELU. À la suite de ce bloc, une couche de normalisation est ajoutée pour accélérer la phase d'apprentissage [IS15]. Puis, une couche de *average pooling*, dont la sortie est la moyenne des entrées, avec des filtres de taille 2×2. Une couche de dropout est appliquée au résultat de la couche de pooling afin de réduire le sur-apprentissage. Finalement, le vecteur de sortie est lissé et donné en entrée d'une couche de trois neurones. La fonction Softmax définie par

$$S(y)_i = \frac{e^{y_i}}{\sum_j e^{y_j}}, 1 \leq i < m, \tag{3.2}$$

est appliquée à la couche de sortie afin de calculer la probabilité de chaque étiquette. Le CNN est entraîné avec la fonction de perte *binary cross-entropy* et 10% de l'ensemble de traces pour l'entraînement servent à la phase de validation.

L'évaluation de la fonction de perte est résumée dans la Figure 3.4. La Figure 3.5 présente le taux de succès de l'attaque sur plusieurs traces avec des clés différentes en fonction du nombre de traces utilisées durant la phase d'apprentissage.

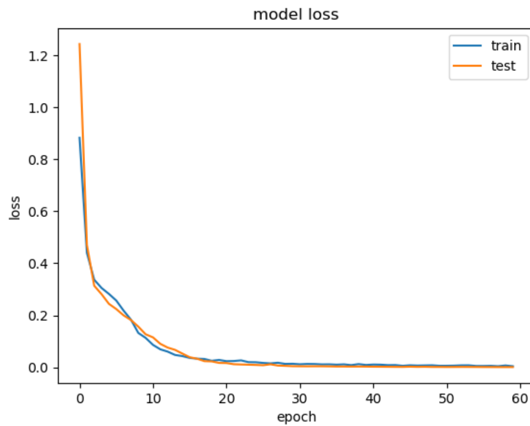


FIGURE 3.4 – Évaluation de la fonction de perte durant la phase d'entraînement

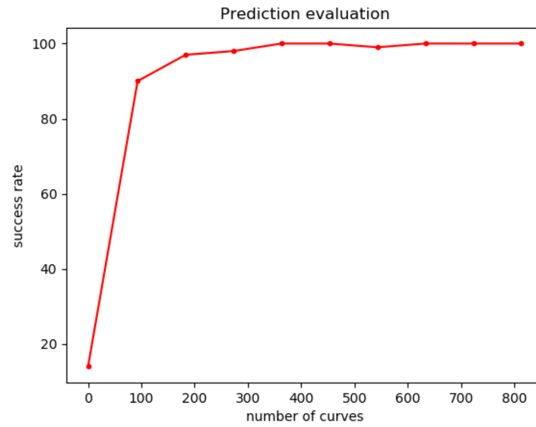


FIGURE 3.5 – Taux de succès de l'attaque suivant le nombre de traces apprises

Remarque 20. Avec des traces simulées, l'attaque fonctionne également en apprenant les coefficients de la clé privée deux par deux, voire trois par trois. Cependant, la phase d'apprentissage est plus longue car l'architecture du réseau de neurones est plus complexe suivant le nombre de points donnés en entrée.

3.1.3.2 Expérimentation

En pratique, nous avons préféré l'attaque template qui est tout aussi efficace lorsque les POIs sont bien sélectionnés et les traces alignées.
L'objectif est donc de créer une probabilité multivariée décrivant les traces acquises pour chacune des trois valeurs possibles du coefficient de la clé.

Dans un premier temps, nous réalisons l'acquisition de nombreuses traces avec des chiffrés et clés privées différentes, avec le banc d'attaque décrit dans la section 1.2.2. Un trigger a été placé juste avant le début du produit de convolution.
Les traces acquises sont catégorisées en trois groupes suivant la valeur du coefficient ciblé.
Dans un second temps, nous effectuons la moyenne des traces dans chacune des classes et nous appliquons la méthode SOST décrite dans l'équation 1.4. La Figure 3.6 présente une partie de la première boucle interne **pour** avec une itération encadrée, ainsi que la méthode SOST appliquée sur 100 traces par classe.

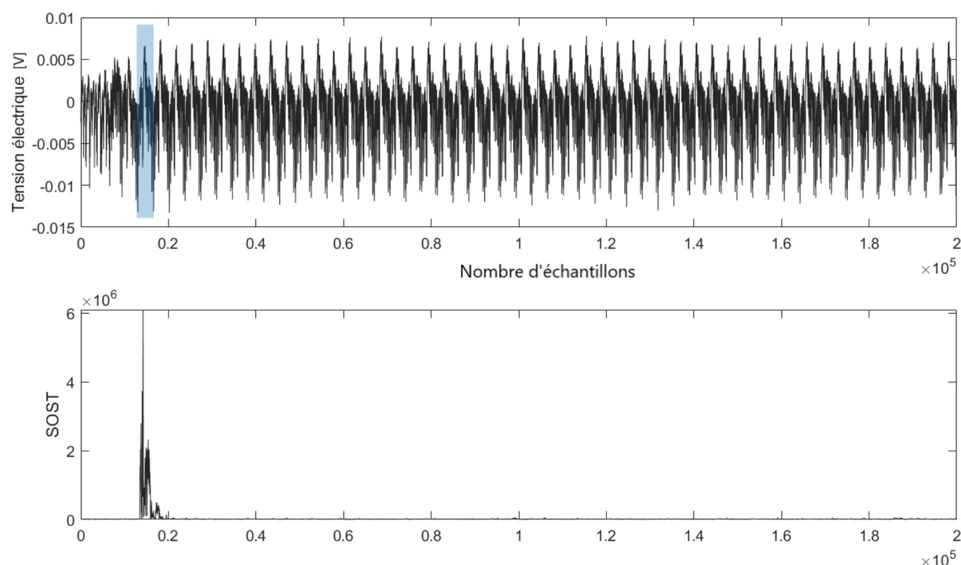


FIGURE 3.6 – Partie d'une trace de la première boucle interne **pour** de l'Algorithme 14 et méthode SOST appliquée sur les traces dont le coefficient f_1 a été modifié

Nous avons alors sélectionné cinq POIs et appliqué la méthodologie de la section 1.2.1.1 (*Les attaques profilées*). Nous avons acquis plus de 1000 traces par classe pour créer le template et effectué l'attaque sur 100 traces avec des clés différentes. Le Tableau 3.3 résume le taux de réussite de l'attaque suivant la valeur du coefficient traité.

Valeur du coefficient	Taux de succès
0	100%
1	95%
-1	93%

TABLE 3.3 – Taux de réussite de l'attaque sur une seule trace en fonction de la valeur du coefficient

Nous avons pu observer qu'il était assez évident de distinguer le coefficient 0 des coefficients $-1/1$. Cependant, nous obtenons quelques erreurs lorsqu'il s'agit de distinguer entre les coefficients -1 et 1.

Remarque 21. En effectuant la méthode SOST sur les couples de classes $(0, 1)$, $(0, -1)$ et $(1, -1)$, nous avons pu observer que, suivant les classes, les POIs n'étaient pas dans la même zone temporelle et que les valeurs étaient moins élevées, notamment entre les couples de classes $(0, 1)$ et $(0, -1)$ d'un côté et $(1, -1)$ de l'autre. Puisque nous choisissons les POIs dont les valeurs sont les plus élevées, il se peut que les POIs discriminant le couple de classes $(1, -1)$ ne soient pas assez représentatifs. Ce qui explique les erreurs occurrentes lorsqu'il s'agit de distinguer les valeurs

1 et -1. Une étude est en cours afin de considérer ces POIs, ce qui permettrait d'améliorer les taux de succès de l'attaque.

3.2 Proposition de contre-mesures

Dans cette partie, nous nous focalisons sur la protection du produit de convolution. Cependant, il est intéressant de considérer les contre-mesures proposées afin de voir si certaines pourraient être adaptées à cette version. Nous proposons notamment deux solutions.

Une première solution consiste à rendre aléatoire les couples de coefficients (f_{i+j}, c_{n-j}) et (f_{i-j}, c_j) dans les deux boucles internes **pour** de l'Algorithme 14 (ligne 3 et 5). Comme présenté dans [HCY19], nous pouvons traiter les coefficients de la clé dans un ordre aléatoire par l'utilisation d'un algorithme de permutation d'une liste contenant les indices $\{0, \ldots, n-1\}$. Le produit de convolution peut alors être implanté suivant l'Algorithme 15, en utilisant, comme permutation aléatoire, l'algorithme de Fisher-Yates (Algorithme 18).

Algorithme 15 : Produit de convolution randomisé

Entrées : n un entier positif, la clé secrète $\mathbf{f} = (f_0, \ldots, f_{n-1})$, le chiffré $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathcal{R}_q$

Output : $\mathbf{a} = \mathbf{f} \star \mathbf{c}$ in $\mathcal{R}_q$

```
   // Création de la liste contenant les indices
1  L ← {0, ..., n − 1}
2  pour i allant de 0 à n − 1 faire
3  |   a_i ← 0
   |   // Permutation de Fisher-Yates
4  |   L_perm := FisherYates(L)
5  |   pour j allant de 0 à n − 1 faire
6  |   |   k := (i + L_perm[j]) mod n
7  |   |   l := (n − L_perm[j]) mod n
8  |   |   a_i := a_i + f_k × c_l
9  |   a_i := a_i mod q
10 retourner a
```

Cette contre-mesure nécessite alors le stockage d'une liste de n coefficients contenant les indices 0 à $n-1$. Dans le cas de NTRU-HRSS, cela reviendrait à stocker 701 indices sur des mots de 16 bits, soit 1402 octets.
Avec cette solution, nous pourrons toujours distinguer les valeurs des coefficients de la clé. Cependant, ces derniers étant traités dans un ordre aléatoire, la complexité de l'attaque est beaucoup trop importante pour réaliser une attaque par force brute.

Une dernière solution consiste à masquer les coefficients de la clé. Pour cela, nous proposons de réaliser un partage de la clé privée sur $\mathcal{R}_q$

$$\mathbf{f} = \mathbf{f}_1 + \mathbf{f}_2 \mod (q, \Phi_1\Phi_n),$$

avec $\mathbf{f}_1 = (f_{1,0}, \ldots, f_{1,n-1})$ et $\mathbf{f}_2 = (f_{2,0}, \ldots, f_{2,n-1})$ choisis dans $\mathcal{R}_q$ comme

$$f_{1,i} = (f_i + q) - f_{2,i}, \text{ pour } 0 \leq i \leq n-1,$$

où $f_{2,i} \in \mathbb{Z}_q$ est choisi aléatoirement. Nous obtenons ainsi deux masques uniformes. Ainsi, le message initial $\mathbf{m}$ pourra être retrouvé en calculant

$$\mathbf{m}_1 = \mathbf{f}_1 \star \mathbf{c} \text{ et } \mathbf{m}_2 = \mathbf{f}_2 \star \mathbf{c},$$

puis,

$$\mathbf{m} = \mathbf{m}_1 + \mathbf{m}_2 \mod (q, \mathbf{\Phi}_1\mathbf{\Phi}_n).$$

Cette contre-mesure nécessite que la clé soit partagée et nécessite donc de stocker $2 \times n$ coefficients de $\mathcal{R}_q$ au lieu de N.
Afin d'éviter des recombinaisons de points sur les traces de consommation, nous pouvons ajouter également de l'aléa dans le traitement des coefficients. Ce traitement peut se réaliser par le biais de la permutation suivante :

$$r_1 + j \times r_2, \text{ pour } , 0 \leq j \leq n-2,$$

avec r_1 et r_2 des nombres aléatoires.

L'algorithme 16 présente cette contre-mesure. Nous supposons que la clé privée a été partagée avant d'effectuer le produit de convolution.

Algorithme 16 : Produit de convolution avec masquage et ajout d'aléa

Entrées : n un entier positif, les deux parties de la clé privée $\mathbf{f}_1 = (f_{1,0}, \ldots, f_{1,n-1})$, $\mathbf{f}_2 = (f_{2,0}, \ldots, f_{2,n-1})$ et le chiffré $\mathbf{c} = (c_0, \ldots, c_{n-1}) \in \mathcal{R}_q$

Output : $\mathbf{a} = \mathbf{f} \star \mathbf{c}$ in $\mathcal{R}_q$

```
1  pour i allant de 0 à n − 1 faire
2      a_{1,i} ← 0
3      a_{2,i} ← 0
       // Initialisation des différents aléas
4      rf_{1,1} := random(1, n − 1)
5      rf_{1,2} := random(1, n − 1)
6      rf_{2,1} := random(1, n − 1)
7      rf_{2,2} := random(1, n − 1)
8      pour j allant de 0 à n − 1 faire
           // Calcul de a_1 = f_1 × c
9          k := (rf_{1,1} + j × rf_{1,2}) mod n
10         l := (n − k + i) mod n
11         a_{1,i} := a_{1,i} + f_{1,k} × c_l
           // Calcul de a_2 = f_2 × c
12         k := (rf_{2,1} + j × rf_{2,2}) mod n
13         l := (n − k + i) mod n
14         a_{2,i} := a_{2,i} + f_{2,k} × c_l
15     a_i := (a_{1,i} + a_{2,i}) mod q
16 retourner a
```

3.3 Conclusion

Dans ce chapitre, nous nous sommes focalisés sur l'analyse de la sécurité du chiffrement NTRU face aux attaques profilées et plus particulièrement la sécurité du produit de convolution entre la clé privée et le chiffré durant le procédé de déchiffrement. Les coefficients de la clé privée ne pouvant prendre que trois valeurs possibles, une modélisation de la consommation de courant suivant ces trois valeurs permet alors de retrouver cette clé. Notre étude a, dans un premier temps, porté sur l'analyse d'une attaque à base de réseaux de neurones sur des traces simulées. Cela nous a permis de vérifier qu'une attaque profilée pouvait être réalisable. Dans un second temps, nous avons réalisé une attaque template sur des traces réelles. Finalement, nous avons proposé des contre-mesures pour protéger l'implantation.
Pour la suite, nous devons évaluer l'efficacité et l'impact des contre-mesures sur les performances. De plus, nous étudions également des possibilités d'automatisation de l'attaque afin de retrouver tous les coefficients de la clé plus rapidement. Ces résultats sont en cours de finalisation pour une future soumission.
Dans le cadre d'un projet étudiant, nous avons également effectué l'implantation du produit de convolution sur un STM32F407 et analysé la consommation électromagnétique du microcontrôleur. Pour réaliser l'attaque, 10000 traces par classes ont été acquises. Une moyenne glissante a été réalisée sur le signal, ce dernier étant trop bruité. Le Tableau 3.4 présente les taux de réussite de l'attaque avec 10 traces. Nous remarquons que la valeur 1 reste plus difficilement distinguable des autres.

Valeur du coefficient	Taux de succès
0	80%
1	60%
-1	90%

Table 3.4 – Taux de réussite de l'attaque en fonction de la valeur du coefficient - STM32

Il serait donc également intéressant de poursuivre les travaux sur le microcontrôleur STM32F407 afin d'essayer d'autres techniques de prétraitement sur le signal pour réduire le bruit.

Chapitre 4

Étude de Crystals-Dilithium

Sommaire

4.1 Opérations et coût mémoire des éléments 100
4.1.1 Génération des clés 102
4.1.2 Signature 103
4.1.3 Vérification 104
4.2 Optimisations réalisées 105
4.2.1 Génération des clés 106
4.2.2 Signature 107
4.2.3 Vérification 108
4.3 Résultats expérimentaux 109
4.3.1 État de l'art 110
4.3.2 Résultats obtenus 110
4.4 Conclusion 111

Dans ce chapitre, nous étudions une première estimation du coût mémoire qu'induirait l'intégration dans un microcontrôleur d'un schéma de signature post-quantique. Crystals-Dilithium est le candidat qui nous semblait le plus adapté pour cette étude. Comme nous avons pu le voir dans le Tableau 1.5, il présentait avec FALCON des tailles de paramètres intéressantes. Cependant, contrairement à FALCON, Crystals-Dilithium présente des facilités pour l'implantation et semble plus évident à protéger des attaques par canaux auxiliaires. Ce dernier est basé sur la structure *Fiat-Shamir with aborts* [Lyu09]. Durant le procédé de signature, l'échantillonnage de certaines variables intermédiaires est rejeté par le biais de la vérification de certaines conditions. Cela permet de s'assurer que les signatures générées ne fuitent aucune information sur la distribution de la clé privée. Ainsi, le processus de signature recommence jusqu'à sortir une signature valide. Une version simplifiée de la signature Crystals-Dilithium est présentée dans la Figure 4.1. Nous utilisons les mêmes notations que dans la spécification de la soumission [DKL+21] dans laquelle une version plus détaillée du cryptosystème est présentée. Les opérations réalisées dans les trois procédés composant le schéma de signature sont explicitées dans la section 4.1.

Génération des clés(1^λ)

1: Clé privée : $(\mathbf{s}_1, \mathbf{s}_2) \xleftarrow{\$_{\rho'}} S_\eta^l \times S_\eta^k$
2: Clé publique : $\mathbf{A} \xleftarrow{\$_\rho} \mathcal{R}_q^{k\times l}$ et
$\mathbf{t} = \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$

Signature($(\mathbf{A}, \mathbf{t}), (\mathbf{s}_1, \mathbf{s}_2), M$)

1: $\mathbf{z} := \perp$
2: **tant que** $\mathbf{z} := \perp$ **faire**
3: $\mathbf{y} \xleftarrow{\$_{\rho''}} S_{\gamma_1 - 1}^l$
4: $\mathbf{w} := \mathbf{A} \cdot \mathbf{y}$
5: $\mathbf{w}_1 := \mathtt{HighBits}_q(\mathbf{w}, 2\gamma_2)$
6: $c = H_B(M||\mathbf{w}_1) \in B_\tau$
7: $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$
8: **si** $||\mathbf{z}||_\infty \geq \gamma_1 - \beta$ ou
$||\mathtt{LowBits}_q(\mathbf{w} - c\mathbf{s}_2, 2\gamma_2)||_\infty \geq \gamma_2 - \beta$
alors
9: $\mathbf{z} := \perp$
10: **fin si**
11: **fin tant que**
12: **retourner** $\sigma = (\mathbf{z}, c)$

Vérification($(\mathbf{A}, \mathbf{t}), M, \sigma$)

1: $\mathbf{w}_1' := \mathtt{HighBits}_q(\mathbf{A} \cdot \mathbf{z} - c\mathbf{t}, 2\gamma_2)$
2: **si** $||\mathbf{z}||_\infty < \gamma_1 - \beta$ et
$c = H(M||\mathbf{w}_1')$ **alors**
3: Accepter
4: **fin si**

FIGURE 4.1 – Version simplifiée de la signature Crystals-Dilithium

Suivant les tailles de paramètres, Crystals-Dilithium atteint les trois niveaux de sécurité requis par le NIST, à savoir 128, 192 et 256 bits de sécurité classique. Pour les trois niveaux de sécurité, les opérations sont effectuées sur des polynômes dans l'anneau $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ avec $q = 2^{23} - 2^{13} + 1$ et $n = 256$. Le Tableau 4.1 résume les paramètres soumis pour le troisième tour : les paramètres γ_1,γ_2 β constituent les bornes d'échantillonnage pour créer une signature valide et le paramètre η représente la borne d'échantillonnage pour la clé privée.

Bien que nous ayons considéré, dans un premier temps, les paramètres du second tour de la standardisation, dans ce chapitre, nous nous concentrons sur les paramètres du troisième tour définissant Crystals-Dilithium-II. Dans un premier temps, nous détaillons alors les opérations effectuées pour la génération des clés, la signature et la vérification, ainsi que le coût mémoire requis pour l'exécution de ces dernières. Dans un second temps, nous expliquons les modifications effectuées afin de réduire ce coût mémoire. Nous concluons ce chapitre par une comparaison avec l'état de l'art actuel des implantations de Crystals-Dilithium sur CORTEX-M3/CORTEX-M4.

Paramètres	Sécurité (bits)	(k,l)	γ_1	γ_2	η	β
Crystals-Dilithium-II	128	$(4,4)$	2^{17}	$(q-1)/88$	2	78
Crystals-Dilithium-III	192	$(6,5)$	2^{19}	$(q-1)/32$	4	196
Crystals-Dilithium-V	256	$(8,7)$	2^{19}	$(q-1)/32$	2	120

Table 4.1 – Paramètres de Crystals-Dilithium pour les trois niveaux de sécurité requis par le NIST

4.1 Opérations et coût mémoire des éléments

Dans cette section, nous traitons des opérations réalisées dans les trois procédés composant le schéma de signature ainsi que leur coût mémoire.

De la même façon que dans [DKL+21], nous définissons :

- $r' = r \pmod{^{\pm}\alpha}$, la réduction modulaire centrée, ainsi, pour un entier positif α, r' correspond à l'unique élément compris entre $-\alpha/2$ et $\alpha/2$ lorsque α est pair et $(-\alpha-1)/2$ et $(\alpha-1)/2$ lorsque α est impair ;
- $r' = r \pmod{^{+}\alpha}$ où r' correspond à l'unique élément compris entre 0 et $\alpha-1$;
- pour un élément $w \in \mathbb{Z}_q$, $|w| = |w \pmod{^{\pm}q}|$ et

$$\|w\|_\infty = \max_i |w_i|, \quad \|w\| = \sqrt{|w_0| + \cdots + |w_{n-1}|};$$

- $S_\eta = \{w \in \mathcal{R} | \|w\|_\infty \leq \eta\}$ et $\tilde{S}_\eta = \{w \in \mathcal{R} | w \pmod{^{\pm}2\eta})\}$;
- B_τ désigne l'ensemble des polynômes dans $\mathcal{R}$ qui ont τ coefficients à ± 1 et le reste des coefficients à 0.
- Soit $r \in \mathbb{Z}_q$ et α un diviseur de $q-1$ pair, les fonctions $\texttt{HighBits}_q(r,\alpha)$ et $\texttt{LowBits}_q(r,\alpha)$ extraient respectivement les bits de poids fort et les bits de poids faible de r. Pour ce faire, l'élément r est décomposé de la manière suivante $r = r_1\alpha + r_0$ avec $r_0 = r \mod {^{+}\alpha}$ et $r_1 = (r-r_0)/\alpha$. Dans le cas où $r - r_0 = q-1$, $r_1 = 0$ et $r_0 = r_0 - 1$, cela permet d'ajouter des petits nombres à r sans que cela ne cause de débordement sur les bits de poids fort de r. En effet, la distance entre $q-1$ et 0 étant de 1, l'ajout d'un petit élément à r peut impacter les bits de poids fort de r de plus de 1, il est alors nécessaire de retirer $q-1$ de la liste des possibilités de l'élément αr. Pour $\mathbf{r} = (r_0, r_1, \ldots, r_k)$ avec $r_i \in \mathcal{R}_q$, nous avons $\texttt{HighBits}_q(\mathbf{r},\alpha) = (\texttt{HighBits}_q(r_0,\alpha), \texttt{HighBits}_q(r_1,\alpha), \ldots, \texttt{HighBits}_q(r_k,\alpha))$, de même avec la fonction $\texttt{LowBits}_q$. Ainsi, les deux fonctions vérifient la propriété décrite dans la spécification suivante
  Lemme 1. Si $\|\mathbf{s}\|_\infty \leq \beta$ et $\|\texttt{LowBits}_q(\mathbf{r},\alpha)\|_\infty < \alpha/2 - \beta$ alors

$$\texttt{HighBits}_q(\mathbf{r},\alpha) = \texttt{HighBits}_q(\mathbf{r}+\mathbf{s},\alpha).$$

Transformation `NTT`. Les multiplications de deux éléments de $\mathcal{R}_q$ définies dans l'Équation 1.3 requièrent n^2 multiplications entre coefficients, soit $256^2 = 65536$

multiplications entre coefficients sont nécessaires pour multiplier deux polynômes dans le cryptosystème Crystals-Dilithium. Sachant que la multiplication s'effectue dans les trois procédés composant le schéma de signature : génération des clés, signature et vérification, cette opération représente un coût d'exécution important. Afin de réaliser des multiplications plus performantes sur cet anneau, les auteurs ont utilisé la transformation NTT (*Number Theoric Transform*) qui se base sur le principe suivant. Pour $n = 256$ et $q = 2^{23} - 2^{13} + 1$, nous pouvons décomposer le polynôme $X^n + 1 \in \mathbb{Z}_q[X]$ en produit de 256 polynômes de degré 1 deux à deux différents (donc deux à deux premiers entre eux). Pour cela, il nous suffit de considérer r une racine primitive 512-ème de l'unité [1] et nous obtenons ainsi :

$$X^{256} + 1 = \prod_{i=0}^{255}(X - r^{2i+1}).$$

Soit ϕ l'isomorphisme d'anneaux donné par le théorème des restes chinois :

$$\begin{array}{rccl} \phi : & \mathcal{R}_q & \to & \prod_{i=0}^{255} \mathbb{Z}_q[X]/(X - r^{2i+1}) \\ & a & \mapsto & \hat{a} = (\hat{a}_0, \dots, \hat{a}_{255}) \end{array},$$

où $\hat{a}_k = a(r^{2\times k+1})$. Autrement dit,

$$\hat{a}_k = \sum_{j=0}^{255} a_i r^{j\times(2k+1)}.$$

L'inverse ϕ^{-1} est défini par

$$\begin{array}{rccl} \phi^{-1} : & \prod_{i=0}^{255} \mathbb{Z}_q[X]/(X - r^{2i+1}) & \to & \mathcal{R}_q \\ & \hat{a} = (\hat{a}_0, \dots, \hat{a}_{255}) & \mapsto & \sum_{k=0}^{255} a_k X^k \end{array},$$

où $a_k = \frac{1}{256} \sum_{j=0}^{255} \hat{a}_j r^{-k\times(2j+1)}$.

Puisque ϕ est un isomorphisme, nous avons $\phi(a \cdot b) = \phi(a) \cdot \phi(b)$ qui correspond à un produit coordonnée par coordonnée et qui nécessite donc seulement 256 multiplications entre coefficients. La transformation des polynômes dans la représentation NTT se réalise par le biais de l'algorithme FFT de Cooley-Tukey [CT65] et la transformation inverse, $\texttt{NTT}^{-1}$, se réalise par le biais de l'algorithme FFT de Gentleman-Sande [GS66]. En incluant le coût de la transformation, la complexité de la multiplication est de $\mathcal{O}(n \log n)$ au lieu de $\mathcal{O}(n^2)$.

Pour tout polynôme $w \in \mathcal{R}_q$, nous notons $\texttt{NTT}(w) = \hat{w}$ sa représentation suite à la transformation NTT, pour tout élément $\mathbf{w} \in \mathcal{R}_q^k$, nous avons $\texttt{NTT}(\mathbf{w}) = \hat{\mathbf{w}} = (\hat{w}_0, \dots, \hat{w}_{k-1})$.

1. Le modulo q a été choisi de sorte qu'il existe une racine primitive 512-ème de l'unité dans $\mathbb{Z}_q$.

Plusieurs fonctions supplémentaires sont utilisées dans la soumission mais ne sont pas détaillées dans ce chapitre. L'échantillonnage des éléments utilise comme source d'aléas la sortie de la fonction de hachage SHA-256 ou la sortie du chiffrement AES-256. Des fonctions permettant de réduire la taille des éléments pour leur stockage ont été utilisées telles que `Power2Round`. Ces fonctions sont détaillées dans la spécification de Crystals-Dilithium [DKL+21].

Dans les sections suivantes, nous donnons le coût mémoire réel des différents éléments permettant l'exécution du schéma de signature. Les différentes opérations étant réalisées dans l'anneau de polynôme $\mathcal{R}_q$, chaque coefficient composant les polynômes nécessite $\lceil \log_2(q) \rceil = 23$ bits, soit un mot de 32 bits. Ainsi, pour le calcul du coût mémoire, nous prendrons en compte le nombre de mots de 32 bits requis pour stocker un élément appartenant à $\mathcal{R}_q$.

4.1.1 Génération des clés

Clé privée. Les deux vecteurs $\mathbf{s}_1, \mathbf{s}_2$ sont générés aléatoirement de manière uniforme à partir d'une graine aléatoire $\rho' \in \{0,1\}^{512}$ par le biais de la fonction `ExpandS` définie par

$$\begin{array}{rccc} \texttt{ExpandS}: & \{0,1\}^{512} & \to & S_\eta^l \times S_\eta^k \\ & \rho' & \mapsto & (\mathbf{s}_1, \mathbf{s}_2) \end{array}.$$

Clé publique. L'élément $\mathbf{A}$ est généré aléatoirement de manière uniforme à partir d'une graine aléatoire $\rho \in \{0,1\}^{256}$ et consiste en une matrice $k \times l$ dont les coefficients sont dans $\mathcal{R}_q$. Ce procédé est réalisé par le biais de la fonction `ExpandA` définie par

$$\begin{array}{rccc} \texttt{ExpandA}: & \{0,1\}^{256} & \to & (\mathbb{Z}_q^{256})^{k \times l} \\ & \rho & \mapsto & \hat{\mathbf{A}} \end{array}.$$

Calcul de l'élément $\mathbf{t} = \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$: le résultat est dans R_q^k. De plus, ce calcul nécessite les éléments $\mathbf{A}, \mathbf{s}_1, \mathbf{s}_2$ et est calculé par

$$\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \hat{\mathbf{s}_1}) + \mathbf{s}_2 \in \mathcal{R}_q.$$

Estimation du coût mémoire. Le Tableau 4.2 présente la taille des différents éléments générés durant le procédé de génération des clés. Ainsi, en générant entièrement tous les éléments, le coût mémoire requis pour exécuter la génération des clés, nous obtenons un total de 28672 octets.

Paramètres	# coefficients dans $\mathbb{Z}_q$	coût mémoire (en octets)
$\mathbf{s}_1$	$l \times 256$	4096
$\mathbf{s}_2$	$k \times 256$	4096
$\mathbf{A}$	$k \times l \times 256$	16384
$\mathbf{t}$	$k \times 256$	4096
Total	$(k \times l + 2 \times k + l) \times 256$	**28672**

TABLE 4.2 – Coût mémoire des éléments dans le procédé de génération des clés.

4.1.2 Signature

Le procédé de signature prend en entrée la clé privée $(\mathbf{s}_1, \mathbf{s}_2)$, la clé publique $(\mathbf{A}, \mathbf{t})$ et le message M.
L'élément $\mathbf{y}$ est généré à partir d'une graine aléatoire ρ'' par le biais de la fonction `ExpandMask` définie par

$$\begin{array}{rccc} \texttt{ExpandMask}: & \{0,1\}^{512} & \rightarrow & S^l_{\gamma_1 - 1} \\ & \rho'' & \mapsto & \mathbf{y} \end{array}.$$

Le signataire calcule ensuite $\texttt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \hat{\mathbf{y}})$ puis, en extrait les bits de poids fort formant ainsi l'élément $\mathbf{w}_1$. La troisième étape de la signature consiste à créer le défi (*challenge*) $c \in B_\tau$. Ce calcul se déroule en deux étapes :

- La fonction de hachage SHA-256 est utilisée sur la concaténation du message M et de $\mathbf{w}_1$, la sortie de la fonction de hachage est notée $\tilde{c}$.
- L'élément $\tilde{c}$ sert alors de graine aléatoire à la fonction de hachage extensible XOF (SHAKE-256) dont la sortie est utilisée comme source d'aléas lors de la création d'un tableau de 256 éléments contenant τ ± 1. La création de ce tableau se réalise par le biais d'une version modifiée de l'algorithme de Fisher-Yates décrite dans la Figure 4.2. Pour les étapes 3 et 4, les éléments j et s sont choisis aléatoirement suivant la sortie de la fonction SHAKE-256.

1: Initialisation de $\mathbf{c} = c_0 \dots c_{255} = 00 \dots 0$
2: **pour** i allant de $256 - \tau$ à 255 **faire**
3: $\quad j \leftarrow \{0, 1, \dots, i\}$
4: $\quad s \leftarrow \{0, 1\}$
5: $\quad c_i := c_j$
6: $\quad c_j := (-1)^s$
7: **fin pour**

FIGURE 4.2 – Création d'un polynôme dans l'ensemble B_τ.

La signature potentielle est finalement calculée comme étant

$$\mathbf{z} = \mathbf{y} + c\mathbf{s}_1 \in \mathcal{R}^l_q.$$

Afin qu'une partie de la signature $\mathbf{z}$ ne dépende plus de la clé privée $\mathbf{s}_1$, les auteurs utilisent une méthode de rejet. Pour cela, un coefficient β est choisi comme étant le maximum possible des coefficients $c\mathbf{s}_1$. Puisque le polynôme c est composé de τ coefficients à ± 1, que le reste des coefficients sont à 0 et que les coefficients de $\mathbf{s}_1$, $\mathbf{s}_2$ sont bornés par η, nous avons $\beta \leq \tau \cdot \eta$. Ainsi, par sécurité, si la norme de $\mathbf{z}$ est plus grande que $\gamma_1 - \beta$, la signature est rejetée et nous recommençons le procédé. Une deuxième condition doit être vérifiée pour que la signature soit valide, les bits de poids faible de $\mathbf{A} \cdot \mathbf{y} - c\mathbf{s}_2$ doivent être inférieurs à $\gamma_2 - \beta$, dans le cas contraire la signature est rejetée et nous recommençons le procédé de signature. En complément de l'aspect sécurité, cette deuxième condition permet d'obtenir l'égalité présentée dans l'équation 4.1 du procédé de vérification.

Estimation du coût mémoire. Le Tableau 4.3 présente la taille des différents éléments générés durant le procédé de signature. Ainsi, en générant entièrement tous les éléments requis pour l'exécution de la signature, nous obtenons un coût mémoire total de 33792 octets, auquel nous devons ajouter la taille du message à signer, ainsi que l'espace requis pour le stockage des valeurs intermédiaires.

Paramètres	# coefficients dans $\mathbb{Z}_q$	taille (en octets)
$\mathbf{y}$	$l \times 256$	4096
$\mathbf{A}$	$k \times l \times 256$	16384
c	256	1024
$\mathbf{s}_1$	$l \times 256$	4096
$\mathbf{z}$	$l \times 256$	4096
$\mathbf{s}_2$	$k \times 256$	4096
Total	$(k \times l + 3 \times l + k + 1) \times 256$	**33792**

TABLE 4.3 – Coût mémoire des éléments dans le procédé de signature.

4.1.3 Vérification

Le procédé de vérification requiert en entrée la clé publique $(\mathbf{A}, \mathbf{t})$, le message M et la signature $\sigma = (\mathbf{z}, c)$.
L'élément $\mathbf{w}_1$ est calculé en prenant les bits de poids fort de l'élément

$$\mathtt{NTT}^{-1}(\hat{\mathbf{A}} \cdot \hat{\mathbf{z}}) + \mathtt{NTT}^{-1}(\hat{c}\hat{\mathbf{t}}).$$

La signature peut alors être considérée valide dans le cas où la norme de $\mathbf{z}$ est plus petite que $\gamma_1 - \beta$ et si c correspond bien à la sortie de la fonction de hachage SHA-256 du message M concaténé à $\mathbf{w}_1$. Autrement dit, la condition principale pour que la signature soit valide réside dans l'égalité suivante

$$H(M||\mathbf{w}_1) = H(M||\mathbf{w}_1') \implies \mathtt{HighBits}_q(\mathbf{A} \cdot \mathbf{z} - c\mathbf{t}, 2\gamma_2) = \mathtt{HighBits}_q(\mathbf{A} \cdot \mathbf{y}, 2\gamma_2). \tag{4.1}$$

Or, nous savons que

$$\begin{aligned}\mathbf{A}\cdot\mathbf{z} - c\mathbf{t} &= \mathbf{A}\cdot(\mathbf{y} + c\mathbf{s}_1) - c(\mathbf{A}\cdot\mathbf{s}_1 + \mathbf{s}_2)\\ &= \mathbf{A}\cdot\mathbf{y} + \mathbf{A}\cdot c\mathbf{s}_1 - \mathbf{A}\cdot c\mathbf{s}_1 - c\mathbf{s}_2\\ &= \mathbf{A}\cdot\mathbf{y} - c\mathbf{s}_2.\end{aligned}$$

Une signature valide vérifie la condition $\|\texttt{LowBits}_q(\mathbf{A}\cdot\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty < \gamma_2 - \beta$ et nous savons que la norme de $c\mathbf{s}_2$ est inférieure à β. Ainsi, ajouter $c\mathbf{s}_2$ à $\mathbf{A}\cdot\mathbf{y}$ n'impacte pas les bits de poids fort (Lemme 1), nous pouvons donc en déduire que

$$\texttt{HighBits}_q(\mathbf{A}\cdot\mathbf{y} - c\mathbf{s}_2, 2\gamma_2) = \texttt{HighBits}_q(\mathbf{A}\cdot\mathbf{y}, 2\gamma_2).$$

Ainsi, l'égalité 4.1 est bien vérifiée.

Estimation du coût mémoire : le Tableau 4.4 présente le coût mémoire des différents paramètres requis pour le procédé de vérification de la signature. Ainsi, afin de réaliser le calcul $\mathbf{w}_1$ et de vérifier les conditions pour valider ou non la signature, le total des éléments à prendre en compte correspond à : $(k\times l+2\times l+k+1)\times 256\times 4 =$ 29696 octets, auquel nous devons ajouter la taille du message signé et l'espace requis pour le stockage des valeurs intermédiaires.

Paramètres	# coefficients dans $\mathbb{Z}_q$	taille (en octets)
$\mathbf{A}$	$k \times l \times 256$	16384
$\mathbf{z}$	$l \times 256$	4096
c	256	1024
$\mathbf{t}$	$k \times 256$	4096
$\mathbf{w}_1'$	$l \times 256$	4096
Total	$(k \times l + 2 \times l + k + 1) \times 256$	**29696**

TABLE 4.4 – Coût mémoire des éléments dans le procédé de vérification de signature.

4.2 Optimisations réalisées

Comme nous avons pu le voir dans la section précédente, les différents éléments utilisés dans les procédés de génération des clés, de signature et de vérification, induisent des coûts mémoire trop importants pour une implantation dans des environnements restreints. Dans cette section, nous proposons alors une solution possible d'implantation pour réduire ces coûts.

Nous avons considéré deux solutions pour le stockage de ces éléments :

— nous nous basons sur l'implantation de la soumission dans laquelle les clés publique et privée sont stockées de manière compacte : la graine aléatoire ρ

permettant de régénérer l'élément publique $\mathbf{A}$ est concaténée avec les autres éléments composant les clés. Ces éléments sont stockés, réduits et compressés suivant les fonctions `Power2Round` et le procédé `Bit-packing` décrits dans la spécification de la soumission [DKL+21]. À la suite de ces opérations, la clé privée requiert 1312 octets et la clé publique 2544 octets pour Crystals-Dilithium-II ;

— si l'espace mémoire le permet, une fois les éléments donnés dans le Tableau 4.2 et générés suite au procédé de génération des clés, nous les stockons entièrement en mémoire FLASH afin d'éviter le coût d'exécution de leur régénération dans les procédés de signature et de vérification de la signature. Dans ce cas, nous pouvons envisager environ 29 ko d'espace libre en mémoire FLASH pour le stockage de ces éléments. La carte décrite dans la section 1.2.2 permettrait ce stockage.

L'implantation n'étant pas effectuée sur un microcontrôleur et l'étude portant plus sur une estimation du coût mémoire que sur les performances, nous avons conservé la première option. Cependant, pour un cas d'usage pratique, en fonction de la carte ciblée et de l'espace de la mémoire FLASH disponible, la deuxième option sera amplement préférée pour de meilleures performances.

4.2.1 Génération des clés

La principale opération dans la génération des clés est le calcul de $\mathbf{t} = \mathbf{A} \cdot \mathbf{s}_1 + \mathbf{s}_2$, avec $\mathbf{A} \in \mathcal{R}_q^{k \times l}$, $\mathbf{s}_1 \in \mathcal{R}_q^l$ et $\mathbf{s}_2 \in \mathcal{R}_q^k$. Autrement dit, pour $0 \leq i < k$, nous avons

$$t_i = \mathtt{NTT}^{-1}\left(\sum_{j=0}^{l} \hat{A}_{ij} \cdot \hat{s}_{1,j}\right) + s_{2,i}.$$

D'un point de vue implantation, au lieu de générer entièrement les éléments pour réaliser ce calcul, nous pouvons générer chaque polynôme de $\mathbf{A}$, $\mathbf{s}_1$ et $\mathbf{s}_2$ et réaliser les opérations élément par élément comme expliqué dans le pseudo-code décrit dans la Figure 4.3.

1: **pour** i allant de 0 à k **faire**
2: $\quad t_i := 0$
3: $\quad$ **pour** j allant de 0 à l **faire**
4: $\quad\quad s_{1,j} \xleftarrow{\$_{\rho'_j}} S_\eta$
5: $\quad\quad \hat{s}_{1,i} := \mathtt{NTT}(s_{1,i})$
6: $\quad\quad \hat{A}_{ij} \xleftarrow{\$_{\rho_{ij}}} \mathcal{R}_q$
7: $\quad\quad t_i := t_i + \hat{A}_{ij} \cdot \hat{s}_{1,j}$
8: $\quad$ **fin pour**
9: $\quad t_i = \mathtt{NTT}^{-1}(t_i)$
10: $\quad s_{2,i} \xleftarrow{\$_{\rho'_i}} S_\eta$
11: $\quad t_i := t_i + s_{2,i}$
12: **fin pour**

FIGURE 4.3 – Calcul de $\mathbf{t} = (t_0, \ldots, t_{k-1})$

Remarque 22. Les éléments $\mathbf{s}_1$ et $\mathbf{s}_2$ étant générés à partir d'une graine aléatoire ρ' et d'un nonce respectif, les polynômes constituant ces derniers peuvent être générés indépendamment les uns des autres.

Ainsi, l'espace mémoire alloué pour la génération des clés suit le schéma suivant :
- 1312 octets sont réservés pour le stockage de la clé privée ;
- 2544 octets sont réservés pour le stockage de la clé publique ;
- $3 \times 256 \times 4$ octets sont réservés pour le stockage des polynômes de $\mathbf{A}$, $\mathbf{s}_1$, $\mathbf{s}_2$ et $\mathbf{t}$. En effet, les polynômes $s_{1,j}$ et $s_{2,j}$ n'interagissant pas entre eux lors du calcul, ils peuvent donc être placés à la même adresse mémoire. Une fois le polynôme t_i calculé, ce dernier est réduit par le biais de la fonction `Power2Round` et stocké directement dans l'espace alloué pour les clés privée et publique de manière compressée.

Nous obtenons donc un total de 6928 octets pour exécuter la génération des clés.

4.2.2 Signature

Le procédé de signature est plus complexe à optimiser en mémoire, ce qui est dû notamment à la méthode de rejet qui nous contraint à régénérer certains éléments plusieurs fois ou à en conserver d'autres. La clé privée étant stockée de manière compressée, nous notons $\texttt{unpacks}(\mathbf{s}_i, j)$, la fonction qui retourne le j^e polynôme de l'élément $\mathbf{s}_i$ pour $i \in \{1, 2\}$.
L'implantation des calculs de $\mathbf{w} = \mathbf{A} \cdot \mathbf{y}$ et $\mathbf{z} = \mathbf{y} + c\mathbf{s}_1$ sont respectivement décrits dans les Figures 4.4 et 4.5. Ces deux calculs font partie de la boucle de rejet décrite à partir de l'étape 2 du procédé de signature donné dans la Figure 4.1. Nous remarquons à l'étape 6 du procédé de signature que l'élément $\mathbf{w_1}$ doit être entièrement généré afin de réaliser le calcul du challenge c. À la suite du calcul des polynômes w_i, ces derniers sont donc décomposés en deux parties :
- $w_{1,i}$ constituant les bits de poids fort, est stocké de manière compressée par le biais de la fonction `packw` ;
- $w_{0,i}$ constituant les bits de poids faible (requis lors de la condition à l'étape 8 du procédé de signature), est stocké entièrement.

1: **pour** i allant de 0 à k **faire**
2: **pour** j allant de 0 à l **faire**
3: $y_j \xleftarrow{\$_{\rho''_j}} S_{\gamma_1 - 1}$
4: $\hat{y}_j := \texttt{NTT}(y_j)$
5: $\hat{A}_{ij} \xleftarrow{\$_{\rho_{ij}}} \mathcal{R}_q$
6: $w_i := w_i + \hat{A}_{ij} \cdot \hat{y}_j$
7: **fin pour**
8: $w_{1,i} := \texttt{packw}(\texttt{HighBits}(w_i))$
9: $w_{0,i} := \texttt{LowBits}(w_i)$
10: **fin pour**

FIGURE 4.4 – Calcul de $\mathbf{w}_1 = \texttt{HighBits}(\mathbf{A} \cdot \mathbf{y})$

1: $c := H_B(M || \mathbf{w}_1)$
2: $\hat{c} := \texttt{NTT}(c)$
3: **pour** i allant de 0 à l **faire**
4: $s_{1,i} := \texttt{unpacks}(\mathbf{s}_1, i)$
5: $z_i := z_i + \hat{c} \cdot s_{1,i}$
6: $z_i := \texttt{NTT}^{-1}(z_i)$
7: $y_i \xleftarrow{\$_{\rho''_i}} S_{\gamma_1 - 1}$
8: $z_i := z_i + y_i$
9: **fin pour**

FIGURE 4.5 – Calcul de $\mathbf{z}$

L'espace mémoire alloué pour la vérification de la signature suit donc le schéma suivant :

- 1312 octets sont réservés pour la clé privée ;
- $4 \times 256 \times 4$ octets sont réservés pour la génération des polynômes $\hat{A}_{i,j}$, y_j, w_i et $w_{1,i}$. Les polynômes $s_{2,i}, s_{1,i}$ sont placés à la même adresse que l'élément $A_{i,j}$ car ils n'interagissent pas entre eux lors des calculs. De même, $cs_{2,i}$ se situe à la même adresse que l'élément y_j, c à la même adresse que w_i et les polynômes z_i à la même adresse que les polynômes $w_{1,i}$;
- $k \times 256 \times 4$ octets pour l'élément $\mathbf{w}_0$;
- 2420 octets sont réservés pour le stockage de la signature qui contient les éléments la composant de manière compressée.

Remarque 23. D'autres éléments, non présentés dans la version simplifiée (Figure 4.1), doivent également être pris en compte, cependant, ils n'impactent pas le coût mémoire présenté ci-dessus car ils ont pu être placés aux mêmes adresses que les éléments présentés.

En conclusion, nous obtenons un total de 11924 octets d'espace mémoire requis pour le stockage des différents éléments présents dans le procédé de signature. En pratique, en comptant les buffers supplémentaires, internes au procédé de signature, nous exécutons la signature avec 13200 octets d'espace RAM.

4.2.3 Vérification

Le procédé de vérification de la signature s'assure dans un premier temps que certaines conditions sont satisfaites pour valider ou non la signature avant de poursuivre, telles que la taille du message reçu ou encore $\|\mathbf{z}\|_\infty < \gamma_1 - \beta$. Pour calculer $\mathbf{w}_1'$, il nous faut d'abord calculer $\mathbf{A} \cdot \mathbf{z} - c\mathbf{t}$, ce qui peut être effectué de manière équivalente au pseudo-code décrit dans la Figure 4.3. La signature (composée du challenge c et de l'élément $\mathbf{z} \in \mathcal{R}_q^k$) et la clé publique, toutes deux stockées de manière compressée, nous notons `unpackz(z,` j`)` et `unpackt(t,` i`)` les fonctions qui retournent respectivement le j^{e} polynôme de $\mathbf{z}$ et le i^{e} polynôme de $\mathbf{t}$ dans leur forme non compressée ; autrement dit, chaque coefficient est écrit sur 32 bits. À la fin de chaque itération, les éléments $w_{1,i}'$ sont stockés de manière compressée et concaténés dans un même buffer. Une fois le calcul complet de $\mathbf{w}_1'$ terminé, le buffer contenant $\mathbf{w}_1'$ compressé est alors utilisé pour vérifier la deuxième condition de validation de la clé.

1: $\hat{c} = \texttt{NTT}(c)$
2: **pour** i allant de 0 à k **faire**
3: $tmp_1 := 0$
4: $tmp_2 := 0$
5: **pour** j allant de 0 à l **faire**
6: $z_j \leftarrow \texttt{unpackz}(\mathbf{z}, j)$
7: $\hat{z}_j := \texttt{NTT}(z_j)$
8: $\hat{A}_{ij} \xleftarrow{\$_\rho} \mathcal{R}_q$
9: $tmp_1 := tmp_1 + \hat{A}_{ij} \cdot \hat{z}_j$
10: **fin pour**
11: $t_i \leftarrow \texttt{unpackt}(\mathbf{t}, i)$
12: $t_i := \texttt{NTT}(t_i)$
13: $tmp_2 := \hat{c} \cdot \hat{t}_i$
14: $tmp_1 := tmp_1 + tmp_2$
15: $tmp_1 := \texttt{NTT}^{-1}(tmp_1)$
16: $w'_{1,i} = \texttt{HighBits}_q(tmp1, 2\gamma_2)$
17: **fin pour**

FIGURE 4.6 – Calcul de $\mathbf{A} \cdot \mathbf{z} - c\mathbf{t}$

Ainsi, l'espace mémoire alloué pour la vérification de la signature suit le schéma suivant :

- 2544 octets sont réservés pour la clé publique ;
- $2 \times 256 \times 4$ octets sont réservés pour le stockage des composantes de la signature : le polynôme c et les polynômes de $\mathbf{z}$. Le polynôme tmp_2 est placé à la même adresse que les polynômes z_i car ils n'interagissent pas entre eux lors des calculs, le même constat est fait pour les polynômes de $\mathbf{w}$ avec le polynôme tmp_2 ;
- $2 \times 256 \times 4$ octets sont réservés pour le stockage des polynômes de $\mathbf{A}$ et du polynôme tmp_1. L'élément $\mathbf{t}$ n'interagissant pas avec $\mathbf{A}$, les polynômes t_i sont placés à la même adresse que les polynômes A_{ij}.

L'espace mémoire requis pour le stockage des principaux éléments nécessaires à la vérification de signature s'élève à 6640 octets. En pratique, en considérant les buffers supplémentaires internes au procédé de vérification, nous obtenons un total de 7900 octets pour exécuter la vérification de signature.

4.3 Résultats expérimentaux

Nous avons implanté Crystals-Dilithium suivant l'implantation fournie pour la standardisation du NIST, en apportant nos modifications afin de réduire le coût mémoire, ainsi qu'en implantant quelques fonctions en assembleur ARM. L'implantation s'est effectuée en utilisant l'outil de développement IAR pour CORTEX-M3.
Nous avons, dans un premier temps, considéré le candidat du second tour de la standardisation Crystals-Dilithium-III dont les paramètres correspondaient à 128 bits de sécurité classique [DKL+19]. Dans un second temps, nous avons pris en compte les

modifications apportées par les auteurs pour le troisième tour de la standardisation.

4.3.1 État de l'art

Plusieurs travaux portant sur des implantations optimisées en mémoire ou en performances de Crystals-Dilithium ont été effectués pour CORTEX-M3 et CORTEX-M4 :

- avec les paramètres du second tour :
  - Dans [GKOS18], Ravi et al. fournissent une première évaluation des cryptosystèmes à base de réseaux euclidiens sur le microcontrôleur ARM CORTEX-M4F. Leur comparaison comprend l'implantation de Crystals-Dilithium.
  - Dans [RGCB20] Ravi et al. réalisent des optimisations, notamment dans le procédé de signature, permettant ainsi d'obtenir de meilleures performances sur un microcontrôleur ARM CORTEX-M4.
  - Dans [GKS20], Denisa O. C. Greconici et al. présentent différents scénarios afin d'évaluer le coût mémoire/performances d'une implantation de Crystals-Dilithium sur ARM CORTEX-M3 et ARM CORTEX-M4. Ils décrivent également une implantation, en temps constant et langage assembleur pour CORTEX-M3 et CORTEX-M4, de la transformation `NTT`.
- avec les paramètres du troisième tour :
  - Dans [GHK+21], Gonzalez et al. exécutent la vérification de signature de différents schémas post-quantiques avec 8 ko de RAM dont Crystals-Dilithium.

4.3.2 Résultats obtenus

Dans le Tableau 4.5, nous présentons les résultats obtenus, à partir de notre implantation, pour les paramètres du second tour. Nous comparons l'espace mémoire requis pour notre implantation avec les résultats de l'implantation la plus optimisée en mémoire, fournis dans [GKS20]. Nous présentons également les performances de l'exécution du schéma de signature Crystals-Dilithium. Pour cela, nous avons exécuté plusieurs signatures et effectué une moyenne des temps d'exécution. Le nombre de cycles a été compté avec la fonctionnalité de débugueur d'IAR [2]. L'implantation étant réalisée de manière totalement logicielle et non porté sur microcontrôleur, les performances indiquées prennent en compte la régénération de plusieurs éléments. Elles ne sont alors pas représentatives d'une implantation en pratique, notamment dans le cas où les éléments pourraient être stockés en mémoire FLASH et non-régénérés à chaque itération.

2. `https://www.iar.com/knowledge/learn/debugging/how-to-measure-execution-time-with-cyclecounter/`

	Ce travail		[GKS20]
	Performances	coût mémoire	coût mémoire
Procédé	(cycles $\times 10^6$)	(en octets)	(en octets)
Génération des clés	$10,74$	8048	8940
Signature	$39,26$	16144	9948
Vérification	$9,76$	9025	10028

Table 4.5 – Comparaison de l'espace mémoire requis (en octets) pour exécuter les trois algorithmes composant la signature Cyrstals-Dilithium avec les paramètres du second tour ainsi que les performances de l'implantation

Dans le Tableau 4.6, nous présentons les résultats obtenus pour les paramètres du troisième tour de la standardisation et donnons le coût mémoire requis pour le procédé de vérification donné dans [GHK+21].

	Ce travail		[GHK+21]
	Performances	coût mémoire	coût mémoire
Procédé	(cycles $\times 10^6$)	(en octets)	(en octets)
Génération des clés	$6,58$	6928	-
Signature	$37,47$	13200	-
Vérification	$6,77$	7900	8048

Table 4.6 – Résultats pour les paramètres du troisième tour

4.4 Conclusion

Nous avons proposé une évaluation du coût mémoire du candidat Crystals-Dilithium. Bien que nous ne puissions exécuter le procédé de signature avec 8 ko de RAM, comme nous avons pu le voir dans l'état de l'art, des optimisations restent possibles afin de réduire le coût mémoire. Cependant, nous obtenons des résultats intéressants concernant la génération des clés et la vérification de la signature. D'un point de vue performance, la régénération des différents éléments à partir de graines aléatoires a un réel impact, notamment dans le procédé de signature où la méthode de rejet nous contraint de régénérer les éléments plusieurs fois. Pour la suite, il serait alors intéressant de réaliser une implantation sur microcontrôleur afin d'utiliser la mémoire FLASH et ainsi, évaluer les performances dans un cas pratique. De plus, il sera également nécessaire de prendre en compte les attaques existantes [MGTF19] afin d'intégrer les contre-mesures à l'implantation ou encore de considérer les différentes implantations en temps constant de la multiplication `NTT` en assembleur présenté dans [GKS20].

Conclusion générale et perspectives

Face au développement de la recherche sur la physique quantique générant ainsi le développement à grande échelle d'ordinateurs quantiques, l'algorithme quantique de Shor devient de plus en plus une menace pour la cryptographie asymétrique actuellement déployée. La nécessité de préparer la transition vers de futures solutions cryptographiques résistantes aux attaques quantiques connues a ainsi poussé le NIST à lancer une campagne de standardisation de cryptosystèmes post-quantiques. Dans cette thèse, nous avons donc étudié des cryptosystèmes prometteurs pour de futures implantations dans des environnements restreints. Ces candidats présentent des avantages de par les tailles de paramètres utilisés ou de par les nombreuses années d'études (sécurité mathématique, sécurité matérielle).
Dans cette thèse, nous avons choisi le candidat ROLLO parmi les mécanismes d'encapsulation de clé, le candidat NTRU pour le système de chiffrement et Crystals-Dilithium comme schéma de signature. L'étude de ces candidats a permis d'avoir une première estimation de l'impact qu'engendrerait l'intégration de ces cryptosystèmes, tant au niveau espace mémoire requis qu'au niveau de l'aspect sécurité de ces implantations.

Nous avons commencé par l'évaluation du cryptosystème ROLLO, à base de codes correcteurs en métrique rang. Bien que les codes utilisés soient assez récents, ce choix s'est justifié par une comparaison avec les candidats, à base de codes correcteurs, de la standardisation présents au second tour. Le candidat ROLLO présentait alors les tailles de paramètres les plus intéressantes. Les deux premiers niveaux de sécurité purent ainsi être implantés environ 4 ko de RAM, permettant ainsi d'envisager une intégration future sans le développement de nouveaux circuits. Malgré l'attaque algébrique trouvée et la soumission de nouveaux paramètres, ROLLO reste un candidat intéressant en termes d'espace mémoire requis pour son exécution. En effet, ROLLO-I-128 peut être exécuté avec 8 ko de RAM. Cependant, avec ses nouveaux paramètres, des optimisations doivent encore être prises en compte afin d'améliorer les performances telles que la multiplication et l'inversion dans $\mathbb{F}_{2^m}[X]/(P_n)$. Il serait également intéressant de considérer la bibliothèque *rbc_library*, dédiée à l'implantation rapide de cryptosystèmes à base de codes en métrique rang.
D'un point de vue sécurité, nous avons pu mettre en évidence que différentes implantations du pivot de Gauss (en temps constant ou non) peuvent être vulnérables aux attaques par canaux auxiliaires menant au recouvrement de la clé privée dans la sou-

mission ROLLO. Nous avons finalement proposé des contre-mesures afin de protéger le protocole des attaques proposées. Ces travaux sont disponibles dans [MBC+20] et [CMRM21].

Dans un second temps, nous nous sommes penchés sur le chiffrement NTRU, étudié depuis de nombreuses années et aujourd'hui finaliste de la standardisation. Bien que ce chiffrement ait été prouvé sûr contre la cryptanalyse, ses implantations sont moins triviales à sécuriser contre les attaques par canaux auxiliaires. Une attaque concerne le procédé de déchiffrement dans lequel une multiplication sur $\mathcal{R}_q$ doit être protégée, puisque la clé privée est directement impliquée avec la donnée d'entrée pour que cette dernière soit déchiffrée. Dans ce cas particulier, il est bien connu que des attaques par canaux auxiliaires sont assez efficaces pour retrouver la clé privée. Ainsi, après avoir établi les différentes attaques existantes sur différentes implantations de NTRU, nous nous sommes concentrés sur l'étude d'attaques profilées sur ce cryptosystème ; plus précisément, des attaques à base de réseaux de neurones sur des traces simulées et par template sur des traces réelles. Le candidat NTRU se prête assez bien à ce genre d'attaques puisque nous pouvons modéliser la consommation de courant du circuit en fonction des trois valeurs possibles des coefficients de la clé privée, manipulés au cours de la multiplication. Finalement, nous avons proposé de nouvelles pistes de contre-mesures pour protéger l'implantation. Ces résultats sont en cours de finalisation pour une future soumission.

Pour notre dernière étude, nous avons pu estimer le coût mémoire induit par le schéma de signature post-quantique Crystals-Dilithium, finaliste de la standardisation du NIST. Les signatures étant largement déployées dans les systèmes embarqués, il était nécessaire, d'un point de vue industriel, d'évaluer l'impact sur les ressources mémoires pour de futures intégrations. À cet effet, l'étude s'est principalement portée sur l'estimation du coût mémoire de Crystals-Dilithium qui présentait des tailles de paramètres également intéressantes et semblait plus simple à protéger des attaques par canaux auxiliaires que son concurrent FALCON. Même si la réduction mémoire n'a pu atteindre les 8 ko pour exécuter la signature, au vu de l'état de l'art, des optimisations restent cependant possibles et nos résultats sont raisonnablement comparables avec les publications [GKS20] et [GHK+21]. Afin de limiter l'impact de la régénération de certains éléments, des fonctions ont été codées en assembleur ARM pour CORTEX-M3 telles que la transformation `NTT`. En perspective, des modifications peuvent encore être prises en compte telles que l'implantation en temps constant de certaines fonctions ainsi que l'analyse générale de la sécurité de l'implantation.

Annexe A

Algorithmes

A.1 Algorithme d'inversion sur $\mathbb{F}_{2^m}$ tiré de [HMV04, Algo. 2.48]

Algorithme 17 : Inversion sur $\mathbb{F}_{2^m}$

Entrées : a un polynôme binaire non nul de degré au plus $m-1$

Résultat : $r(z) = a(z)^{-1} \mod P_m(z)$

1 $u \leftarrow a, v \leftarrow P_m$
2 $g_1 \leftarrow 1, g_2 \leftarrow 0$
3 **tant que** $u \neq 1$ **faire**
4 $\quad j \leftarrow deg(u) - deg(v)$
5 $\quad$ **si** $j < 0$ **alors**
6 $\quad\quad u \leftrightarrow v$
7 $\quad\quad g_1 \leftrightarrow g_2$
8 $\quad\quad j \leftarrow -j$
9 $\quad u \leftarrow u + z^j.v$
10 $\quad g_1 \leftarrow g_1 + z^j.g_2$
11 **retourner R**

A.2 Algorithme de Fisher-Yates utilisé pour permuter aléatoirement les éléments d'une liste L

Algorithme 18 : FisherYates

Entrées : Une liste L de n éléments
Résultat : La liste L mélangée

```
1 pour i allant de n − 1 à 0 faire
2 |  j = random()  mod i
3 |_ échanger L_i et L_j
```

Annexe B

Codes sources des attaques

B.1 Code source de l'attaque sur l'implantation de pivot de Gauss de référence, en temps constant (en utilisant *SageMath*)

```
1  def matrix_equation(pivot_index, mask_firstloop, mask_secondloop,
     nbrow):
2
3    # Initialization of the two matrices that will determine the
     system of equations
4    Meq1 = identity_matrix(Zmod(2),nbrow)
5    Meq2 = identity_matrix(Zmod(2),nbrow)
6
7    # Placing coefficients at 1 for the additions on the pivot row,
     defined thanks to the 'masks' of the first loop
8    for i in range(len(mask_firstloop)):
9      if(mask_firstloop[i]):
10       Meq1[pivot_index,i]=1
11
12   # Placing coefficients at 1 for the additions on rows with the
     leading coefficient at 1, defined thanks to the masks of the
     second loop
13   for i in range(len(mask_secondloop)):
14     if(mask_secondloop[i]):
15       Meq2[i,pivot_index]=1
16
17   return Meq1,Meq2
18
19 def matrix_equation_columnindex(masks_firstloop, masks_secondloop,
     nbrow,columnindex):
20
21   M =[]
22
23   # Initialization of M with the matrices of equations
24   for i in range(columnindex+1):
25     M.append(matrix_equation(i, masks_firstloop[i],
     masks_secondloop[i], nbrow))
26
```

```
27    return M
28
29  def equations_to_solve(masks_firstloop, masks_secondloop,nbcolumn,
      nbrow,columnindex):
30
31    # Initiatialization of the matrices to define the system of
      equations
32    Meq = matrix_equation_columnindex(masks_firstloop,
      masks_secondloop,nbrow,columnindex)
33
34    # Multiplication of the matrices to determine all the equations
      for the column "columnindex"
35    Stmp = Meq[0][0]
36    for i in range(columnindex):
37      Stmpbis = Meq[i][1]* Stmp
38      Stmp =Meq[i+1][0] * Stmpbis
39
40    #Solving the system of equations
41    S = Stmp.solve_right(matrix(Zmod(2),masks_secondloop[columnindex
      ]).transpose())
42
43    return S
```

B.2 Code source de l'attaque sur l'implantation du pivot de Gauss en temps constant disponible sur *GitHub* (en utilisant *SageMath*)

```
1  # matrix_equation returns a matrix of all the additions made on
      rows to get the system of equation for given pivot index and
      masks.
2  def matrix_equation(index_column, pivot_index, mask1, mask2, mask3,
      nbrow):
3    copy_pivot_index = pivot_index
4
5      # If mask1 is full of ones then the column does not contain a
      pivot or the pivot is on the last row.
6      # In the first case we return the identity matrix,
7      # Else the position of the first zero on mask1 determines the
      pivot's position.
8    if(mask1.count(0)==0):
9      if(mask2[-1]&mask3[-1] == 1):
10       pivot_position= nbrow-1
11     else:
12       return identity_matrix(Zmod(2),nbrow), copy_pivot_index
13   else:
14     if(mask1.index(0)==0):
15       pivot_position= pivot_index
16     else:
17       pivot_position = mask1.index(0) -1
18
19   # Initialization of the two matrices that will determine the
```

```
    system of equations
20   Mpivot = identity_matrix(Zmod(2),nbrow)
21   Mrows = identity_matrix(Zmod(2),nbrow)
22
23   # Then the matrix Mpivot get an additionnal one that indicates
     which row has been added to pivot row
24   Mpivot[pivot_index,pivot_position] = 1
25
26     # The pivot row is added to the processed row when (mask2[i]
     and mask3[i])=1.
27     # We use those mask to determine when operations on the rows
     have been performed except for the pivot row.
28     # All the operations are represented by a one in the matrix
     Mrows at the position i (number of the processed row) and the
     pivot index.
29   for i in range(nbrow):
30     if((mask2[i]&mask3[i])==1):
31       Mrows[i,pivot_index]=1
32
33   # Meq is the matrix representation of all the addition to make to
       get the system of equation
34   Meq = Mrows*Mpivot
35   copy_pivot_index = pivot_index + 1
36   return Meq, copy_pivot_index
37
38 def matrix_equation_columnindex(mask1s, mask2s,mask3s,nbrow,
     columnindex):
39   M =[]
40   pivot_index=0
41   # Initialization of M with the matrices of equations
42   for i in range(columnindex):
43     R = matrix_equation(i, pivot_index, mask1s[i], mask2s[i],
     mask3s[i],nbrow)
44     # In the case there is no pivot in a column, the index pivot is
       unchanged
45     pivot_index = R[1]
46     M.append(R[0])
47
48   return M
49
50 def solution_vector(mask2s,columnindex):
51   # Return the vector solution of the system of equation for the
     column "columnindex"
52   return matrix(Zmod(2),mask2s[columnindex])
53
54 def equations_to_solve(mask1s, mask2s,mask3s,nbcolumn,nbrow,
     columnindex):
55   # In the case we want to recover the column 0 of the  matrix, the
       vector mask2 gives directly the solution
56   if(columnindex==0):
57     S = solution_vector(mask2s,columnindex)
58     return S
59
60   # Initiatialization of the matrices to define the system of
     equations
61   Meq = matrix_equation_columnindex(mask1s,mask2s,mask3s,nbrow,
```

```
     columnindex)
62
63   # Multiplication of the matrices to determine all the equations
      for the column "columnindex"
64   Stmp = identity_matrix(Zmod(2),nbrow)
65   for i in range(columnindex):
66     Stmp = Meq[i]*Stmp
67
68   #Solving the system of equations
69   v_sol = solution_vector(mask2s,columnindex)
70   S = Stmp.solve_right(v_sol.transpose()).transpose()
71
72   return S
```

Bibliographie

[ABB+17a] Nicolas Aragon, Paulo S. L. M. Barreto, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Shay Gueron, Tim Guneysu, Carlos Aguilar Melchor, Rafael Misoczki, Edoardo Persichetti, Nicolas Sendrier, Jean-Pierre Tillich, and Gilles Zémor. BIKE : Bit Flipping Key Encapsulation, December 2017. Submission to the NIST post quantum standardization process.

[ABB+17b] Nicolas Aragon, Olivier Blazy, Slim Bettaieb, Loïc Bidoux, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC first round submisssion : LOCKER - LOw rank parity ChecK codes EncRyption , 2017.

[ABD+17a] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC first round submisssion : LAKE - Low rAnk parity check codes Key Exchange, 2017.

[ABD+17b] Nicolas Aragon, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, Jean-Pierre Tillich, and Gilles Zémor. NIST PQC first round submisssion : Ouroboros-R , 2017.

[AD97] Miklós Ajtai and Cynthia Dwork. A public-key cryptosystem with worst-case/average-case equivalence. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing - STOC '97*. ACM Press, 1997.

[AG19] Nicolas Aragon and Philippe Gaborit. A key recovery attack against LRPC using decryption failures. In *International Workshop on Coding and Cryptography (WCC), Saint-Jacut-de-la-Mer, France*, 2019.

[AGH+19] Nicolas Aragon, Philippe Gaborit, Adrien Hauteville, Olivier Ruatta, and Gilles Zemor. Low rank parity check codes : New decoding algorithms and applications to cryptography. *IEEE Transactions on Information Theory*, 65(12) :7697–7717, dec 2019.

[Ajt96] Miklós Ajtai. Generating hard instances of lattice problems (extended abstract). In Gary L. Miller, editor, *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, pages 99–108. ACM, 1996.

[AKJ+18] Soojung An, Suhri Kim, Sunghyun Jin, HanBit Kim, and HeeSeok Kim. Single Trace Side Channel Analysis on NTRU Implementation. *Applied Sciences*, 8 :2014, 10 2018.

[AMAB+17] Carlos Aguilar Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Jean-Christophe Deneuville, Philippe Gaborit, Edoardo Persichetti, and Gilles Zémor. Hamming Quasi-Cyclic (HQC), November 2017. Submission to the NIST post quantum standardization process. 2017.

[AMAB+19] 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.

[AMAB+20a] 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, 2020.

[AMAB+20b] Carlos Aguilar-Melchor, Nicolas Aragon, Emanuele Bellini, Florian Caullery, Rusydi H. Makarim, and Chiara Marcolla. Constant time algorithms for rollo-i-128. Cryptology ePrint Archive, Report 2020/1066, 2020. Code source available at `https://github.com/peacker/constant_time_rollo.git`.

[AMAB+20c] Carlos Aguilar-Melchor, Nicolas Aragon, Slim Bettaieb, Loïc Bidoux, Olivier Blazy, Maxime Bros, Alain Couvreur, Jean-Christophe Deneuville, Philippe Gaborit, and Gilles Zémor. Rank Quasi-Cyclic (RQC), 2020. Available at `https://pqc-rqc.org/`.

[BBB+20] 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*, pages 64–93. Springer International Publishing, 2020.

[BBC+20] Marco Baldi, Alessandro Barenghi, Franco Chiaraluce, Garardo Pelosi, and Paolo Santini. LEDAcrypt : Low-dEnsity parity-check coDebAsed cryptographic systems, 2020. Submission to the NIST post quantum standardization process.

[BCL+17] Daniel J. Bernstein, Tung Chou, Tanja Lange, Ingo von Maurich, Rafael Misoczki, Ruben Niederhagen, Edoardo Persichetti, Christiane Peters, Peter Schwabe, Nicolas Sendrier, Jakub Szefer, and Wen Wang. Classic McEliece, 2017. Submission to the NIST post quantum standardization process.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004.

[BCS13] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. McBits : Fast constant-time code-based cryptography. In Guido Bertoni and Jean-

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

[BMvT78] E. Berlekamp, R. McEliece, and H. van Tilborg. On the inherent intractability of certain coding problems (corresp.). *IEEE Transactions on Information Theory*, 24(3) :384–386, may 1978.

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional Neural Networks with Data Augmentation against Jitter-Based Countermeasures - Profiling Attacks without Pre-Processing -. *IACR Cryptology ePrint Archive*, 2017 :740, 2017.

[Cer09] Certicom Research. *SEC 1. Standards for Efficient Cryptography Group : Elliptic Curve Cryptography - version 2.0*, (2009). Available at https://www.secg.org/sec1-v2.pdf.

[CHWZ17] Chen Cong, Jeffrey Hoffstein, William Whyte, and Zhenfei Zhang. NIST PQ Submisssion : NTRUEncrypt A lattice based encryption algorithm, 2017.

[CJL+16] Lily Chen, Stephen Jordan, Yi-Kai Liu, Dustin Moody, Rene Peralta, Ray Perlner, and Daniel Smith-Tone. Report on post-quantum cryptography. Technical report, NIST, apr 2016.

[CMRM21] Agathe Cheriere, Lina Mortajine, Tania Richmond, and Nadia El Mrabet. Side-channel attack on rollo post-quantum cryptographic scheme. Cryptology ePrint Archive, Report 2021/477, April 2021. https://eprint.iacr.org/2021/477.

[CN11] Yuanmi Chen and Phong Q. Nguyen. BKZ 2.0 : Better lattice security estimates. In *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2011.

[CRR03] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, pages 13–28. Springer Berlin Heidelberg, 2003.

[CT65] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90) :297–301, 1965.

[Del78] Philippe Delsarte. Bilinear forms over a finite field, with applications to coding theory. *J. Comb. Theory, Ser. A*, 25 :226–241, 1978.

[DH76] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6) :644–654, November 1976.

[DKL+19] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–dilithium : Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (Second Round), 2019. Available at https://pq-crystals.org/dilithium/data/dilithium-specification-round2.pdf.

[DKL+21] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. CRYSTALS–dilithium : Algorithm specification and supporting documentation. Submission to the NIST Post-Quantum Cryptography Standardization Project (Third Round), 2021. Available at https://pq-crystals.org/dilithium/data/dilithium-specification-round3-20210208.pdf.

[Gab85] Ernest Mukhamedovich Gabidulin. Theory of codes with maximum rank distance. *Problemy Peredachi Informatsii*, 21(1) :3–16, 1985.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *In Proc. STOC*, pages 169–178, 2009.

[GGH97] Oded Goldreich, Shafi Goldwasser, and Shai Halevi. Public-key cryptosystems from lattice reduction problems. In *Advances in Cryptology — CRYPTO '97*, pages 112–131. Springer Berlin Heidelberg, 1997.

[GHK+21] Ruben Gonzalez, Andreas Hülsing, Matthias J. Kannwischer, Juliane Krämer, Tanja Lange, Marc Stöttinger, Elisabeth Waitz, Thom Wiggers, and Bo-Yin Yang. Verifying post-quantum signatures in 8 kb of ram. Cryptology ePrint Archive, Report 2021/662, May 2021. https://eprint.iacr.org/2021/662.

[GKOS18] Tim Guneysu, Markus Krausz, Tobias Oder, and Julian Speith. Evaluation of lattice-based signature schemes in embedded systems. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, dec 2018.

[GKS20] Denisa O. C. Greconici, Matthias J. Kannwischer, and Daan Sprenkels. Compact dilithium implementations on cortex-m3 and cortex-m4. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 1–24, dec 2020.

[GLRP06] Benedikt Gierlichs, Kerstin Lemke-Rust, and Christof Paar. Templates vs. stochastic methods. In *Lecture Notes in Computer Science*, pages 15–29. Springer Berlin Heidelberg, 2006.

[GMRZ13a] Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zemor. Low Rank Parity Check codes and their application to cryptography. In Lilya Budaghyan, Tor Helleseth, and Matthew G. Parker, editors, *International Workshop on Coding and Cryptography (WCC)*, Bergen, Norway, Apr 2013. ISBN 978-82-308-2269-2.

[GMRZ13b] Philippe Gaborit, Gaétan Murat, Olivier Ruatta, and Gilles Zemor. Low Rank Parity Check codes and their application to cryptography. In Lilya Budaghyan, Tor Helleseth, and Matthew G. Parker, editors, *The International Workshop on Coding and Cryptography (WCC 13)*, page 13 p., Bergen, Norway, April 2013. ISBN 978-82-308-2269-2.

[GPQ11] Laurie Genelle, Emmanuel Prouf, and Michael Quisquater. Montgomery's Trick and Fast Implementation of Masked AES. In *AFRICACRYPT 2011 - 4th International Conference on Cryptology in Africa*,

volume 6737 of *Lecture Notes in Computer Science*, pages 153–169, Dakar, Senegal, July 2011. springer.

[GS66] W. M. Gentleman and G. Sande. Fast fourier transforms. In *Proceedings of the November 7-10, 1966, fall joint computer conference on XX - AFIPS '66 (Fall)*. ACM Press, 1966.

[Ham50] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2) :147–160, apr 1950.

[HCY19] Wei-Lun Huang, Jiun-Peng Chen, and Bo-Yin Yang. Correlation Power Analysis on NTRU Prime and Related Countermeasures. Cryptology ePrint Archive, Report 2019/100, 2019. https://eprint.iacr.org/2019/100.

[HMV04] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag GmbH, 2004.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU : A ring-based public key cryptosystem. In *Lecture Notes in Computer Science*, pages 267–288. Springer Berlin Heidelberg, 1998.

[IS15] Sergey Ioffe and Christian Szegedy. Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift. pages 448–456, 2015.

[JMV01] Don Johnson, Alfred Menezes, and Scott A. Vanstone. The elliptic curve digital signature algorithm (ecdsa). *Int. J. Inf. Sec.*, 1(1) :36–63, 2001.

[Ker] Keras : The Python Deep Learning library. https://keras.io/.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Advances in Cryptology — CRYPTO' 99*, pages 388–397. Springer Berlin Heidelberg, 1999.

[KO62] Anatolii Alekseevich Karatsuba and Yu P Ofman. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, volume 145 of *2*, pages 293–294. Russian Academy of Sciences, 1962.

[Koc96] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology — CRYPTO '96*, pages 104–113. Springer Berlin Heidelberg, 1996.

[LESCH10] Mun-Kyu Lee, Jeong Eun Song, Dooho Choi, and Dong-Guk Han. Countermeasures against Power Analysis Attacks for the NTRU Public Key Cryptosystem. *IEICE Transactions*, 93-A :153–163, 01 2010.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4) :515–534, dec 1982.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*, pages 1–23. Springer Berlin Heidelberg, 2010.

[LRW97] Eugene M. Luks, Ferenc Rákóczi, and Charles R.B. Wright. Some algorithms for nilpotent permutation groups. *Journal of Symbolic Computation*, 23(4) :335–354, apr 1997.

[LS14] Adeline Langlois and Damien Stehlé. Worst-case to average-case reductions for module lattices. *Designs, Codes and Cryptography*, 75(3) :565–599, feb 2014.

[Lyu09] Vadim Lyubashevsky. Fiat-shamir with aborts : Applications to lattice and factoring-based signatures. In *Advances in Cryptology – ASIACRYPT 2009*, pages 598–616. Springer Berlin Heidelberg, 2009.

[MAA+20] Dustin Moody, Gorjan Alagic, Daniel C Apon, David A Cooper, Quynh H Dang, John M Kelsey, Yi-Kai Liu, Carl A Miller, Rene C Peralta, Ray A Perlner, Angela Y Robinson, Daniel C Smith-Tone, and Jacob Alperin-Sheriff. Status report on the second round of the NIST post-quantum cryptography standardization process. Technical report, NIST, jul 2020.

[MBC+20] Lina Mortajine, Othman Benchaalal, Pierre-Louis Cayrel, Nadia El Mrabet, and Jérôme Lablanche. Optimized and secure implementation of ROLLO-i. In *Code-Based Cryptography*, pages 117–137. Springer International Publishing, 2020.

[McE78] Robert James McEliece. A public-key cryptosystem based on algebraic coding theory. Technical Report 44, California Inst. Technol., Pasadena, CA, January 1978.

[MGTF19] Vincent Migliore, Benoît Gérard, Mehdi Tibouchi, and Pierre-Alain Fouque. Masking dilithium : Efficient implementation and side-channel evaluation. In *Applied Cryptography and Network Security*, pages 344–362. Springer International Publishing, 2019.

[Mic07] Daniele Micciancio. Generalized compact knapsacks, cyclic lattices, and efficient one-way functions. *computational complexity*, 16(4) :365–411, dec 2007.

[MNY17] Moustafa Mahmoud, Mouna Nakkar, and Amr Youssef. A power analysis resistant FPGA implementation of NTRUEncrypt. In *2017 29th International Conference on Microelectronics (ICM)*, pages 1–4, Dec 2017.

[MPP16] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. Breaking Cryptographic Implementations Using Deep Learning Techniques. In *Security, Privacy, and Applied Cryptography Engineering - 6th International Conference, SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings*, pages 3–26, 2016.

[Nat99] National Institute of Standards and Technology. Data encryption standard (des). FIPS Publication 46-3, October 1999.

[Nat01] National Institute of Standards and Technology. Advanced encryption standard (AES). Technical report, NIST, nov 2001.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing - STOC '05*. ACM Press, 2005.

[RGCB20] Prasanna Ravi, Sourav Sen Gupta, Anupam Chattopadhyay, and Shivam Bhasin. Improving speed of dilithium's signing procedure. In *Smart Card Research and Advanced Applications*, pages 57–73. Springer International Publishing, 2020.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, feb 1978.

[Sch94] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *Fast Software Encryption*, pages 191–204. Springer Berlin Heidelberg, 1994.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3) :379–423, jul 1948.

[Sho94] P. Shor. Algorithms for quantum computation : discrete logarithms and factoring. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 124–134, Los Alamitos, CA, USA, nov 1994. IEEE Computer Society.

[SKL+20] Bo-Yeon Sim, Jihoon Kwon, Joohee Lee, Il-Ju Kim, Taeho Lee, Jaeseung Han, Hyojin Yoon, Jihoon Cho, and Dong-Guk Han. Single-trace attacks on the message encoding of lattice-based kems. Cryptology ePrint Archive, Report 2020/992, December 2020. https://eprint.iacr.org/2020/992.

[SSPB19] Simona Samardjiska, Paolo Santini, Edoardo Persichetti, and Gustavo Banegas. A reaction attack against cryptosystems based on LRPC codes. In *Progress in Cryptology – LATINCRYPT*, pages 197–216. Springer International Publishing, 2019.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In *Advances in Cryptology – ASIACRYPT 2009*, pages 617–635. Springer Berlin Heidelberg, 2009.

[SW07] Joseph H. Silverman and William Whyte. Timing Attacks on NTRUEncrypt Via Variation in the Number of Hash Calls. In *CT-RSA*, volume 4377 of *Lecture Notes in Computer Science*, pages 208–224. Springer, 2007.

[Wis18] Wisekey. *MicroXsafe 6001 Summary Datasheet*, 2018. Available at https://www.wisekey.com/microxsafe-6001/.

[ZWW13] Xuexin Zheng, An Wang, and Wei Wei. First-order collision attack on protected NTRU cryptosystem. *Microprocessors and Microsystems*, 37 :601–609, 08 2013.

École Nationale Supérieure des Mines de Saint-Étienne

NNT : 2021LYSEM028

Lina MORTAJINE

STUDY OF POST-QUANTUM ALGORITHMS THAT CAN BE IMPLEMENTED IN PRACTICE

Speciality : Microelectronics

Keywords : post-quantum, code-based cryptography, lattice-based cryptography, implementation, microcontroller, side-channel attacks

Abstract :

The emergence of quantum computer research increasingly poses a threat to the asymmetric cryptography currently in use. Thus, in 2016, NIST launched a post-quantum project campaign to standardize alternatives that are resistant to quantum attacks. Post-quantum public key encryption, key-encapsulation mechanisms and signature systems are based on hard-to-solve mathematical problems, such as problems based on lattices and error-correcting codes.
Although the theoretical aspect of some of these protocols has been studied for many years, their integration in practice faces many constraints, especially in restraint environments such as microcontrollers. Indeed, microcontrollers, which we find in many connected objects, are limited in memory space and computational power. Moreover, they are often subject to physical attacks related to the analysis of the energy consumption emanating from the circuit executing a cryptosystem. Thus, to prepare for the transition to post-quantum cryptography, the practical aspects of integrating these protocols (efficient implementations and the resistance against side-channel attacks) must be studied and considered. These various aspects constitute the context of this thesis, in order to be prepared for a rapid integration of post-quantum cryptosystems at the availability of the first standards.
In this thesis, we begin with the implementation of a cryptosystem based on codes (ROLLO) with an optimization of the memory cost and we propose a first analysis of side-channel attacks on this candidate and the countermeasures to protect the implementation. Then, we look at the security aspect of a promising candidate (NTRU), protocol based on lattices and we propose new countermeasures too. We conclude with a study of an implementation with memory optimization of a signature scheme (Crystals-Dilithium) based on lattices, finalist of the NIST standardization.

École Nationale Supérieure des Mines de Saint-Étienne

NNT : 2021LYSEM028

Lina MORTAJINE

ANALYSES D'ALGORITHMES POST-QUANTIQUES IMPLANTABLES EN PRATIQUE

Spécialité : Microélectronique

Mots clefs : post-quantique, cryptographie à base de codes correcteurs, cryptographie à base de réseaux euclidiens, implantations, microcontrôleurs, attaques par canaux auxiliaires

Résumé : L'émergence de la recherche sur l'ordinateur quantique représente de plus en plus une menace pour la cryptographie asymétrique actuellement utilisée. Ainsi, en 2016, le NIST a lancé une campagne de standardisation post-quantique visant à déployer des alternatives résistantes aux attaques quantiques. Les systèmes de chiffrement et de signature post-quantiques reposent sur des problèmes mathématiques difficiles à résoudre comme ceux que l'on trouve dans les réseaux euclidiens et dans les codes correcteurs d'erreurs. Bien que l'aspect théorique de ces protocoles soit, pour certains, étudié depuis de nombreuses années, leur intégration en pratique crée de nombreuses contraintes, notamment dans des environnements restreints tels que les microcontrôleurs. En effet, ces microcontrôleurs que nous retrouvons dans de nombreux objets connectés, sont limités en espace mémoire et en puissance de calculs. De plus, ils sont souvent sujets à des attaques physiques liées à l'analyse de la consommation de courant du circuit lorsque ce dernier exécute un cryptosystème. Afin de préparer la transition vers la cryptographie post-quantique, les aspects pratiques d'une intégration de ces protocoles (implantation efficace et résistance aux attaques par canaux auxiliaires) doivent être étudiés et pris en compte. Ces divers aspects constituent le contexte de cette thèse, en vue d'une intégration rapide de cryptosystèmes post-quantiques lors de la disponibilité des premiers standards.
Dans cette thèse, nous commençons par réaliser l'implantation d'un cryptosystème à base de codes correcteurs (ROLLO) avec une optimisation du coût mémoire et proposons une première analyse d'attaques physiques sur ce candidat ainsi que les contre-mesures associées afin de protéger l'implantation. Puis, nous nous penchons sur la sécurité d'un candidat prometteur (NTRU), protocole à base de réseaux euclidiens et nous proposons également de nouvelles contre-mesures. Nous terminons par une étude portant sur l'implantation avec optimisation en mémoire d'un schéma de signature (Crystals-Dilithium) à base de réseaux euclidiens, finaliste de la standardisation du NIST.