Cryptographie légère intrinsèquement résistante aux attaques physiques pour l'Internet des objets.

Benjamin Lac

▶ **To cite this version:**

Benjamin Lac. Cryptographie légère intrinsèquement résistante aux attaques physiques pour l'Internet des objets.. Autre. Université de Lyon, 2018. Français. NNT : 2018LYSEM021 . tel-02884873

HAL Id: tel-02884873
https://theses.hal.science/tel-02884873

Submitted on 30 Jun 2020

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 : 2018LYSEM021

THESE de DOCTORAT DE L'UNIVERSITE DE LYON
opérée au sein de
l'Ecole des Mines de Saint-Etienne

Ecole Doctorale N° 488
Sciences, Ingénierie, Santé

Spécialité de doctorat : Microélectronique
Discipline : Cryptographie

Soutenue publiquement le 18/10/2018, par :
Benjamin Lac

Cryptographie légère intrinsèquement résistante aux attaques physiques pour l'Internet des Objets

---

Devant le jury composé de :

Clavier, Christophe	Professeur, Univ. de Limoges	Rapporteur
Standaert, François-Xavier	Professeur, Univ. catholique de Louvain	Rapporteur
Bossuet, Lilian	Professeur, Univ. Jean Monnet	Examinateur
Goubin, Louis	Professeur, Univ. de Versailles	Examinateur
Canteaut, Anne	Directrice de recherche, Inria	Directrice de thèse
Fournier, Jacques	Responsable scientifique, CEA	Directeur de thèse
Gérard, Benoît	Chercheur, DGA-MI	Correspondant DGA
Sirdey, Renaud	Directeur de recherche, CEA	Encadrant CEA

Spécialités doctorales	Responsables :	Spécialités doctorales	Responsables
SCIENCES ET GENIE DES MATERIAUX	K. Wolski Directeur de recherche	MATHEMATIQUES APPLIQUEES	O. Roustant, Maître-assistant
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	D. Graillot, Directeur de recherche	MICROELECTRONIQUE	Ph. Lalevée, Professeur

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

ABSI	Nabil	MR	Génie industriel	CMP
AUGUSTO	Vincent	CR	Image, Vision, Signal	CIS
AVRIL	Stéphane	PR2	Mécanique et ingénierie	CIS
BADEL	Pierre	MA(MDC)	Mécanique et ingénierie	CIS
BALBO	Flavien	PR2	Informatique	FAYOL
BASSEREAU	Jean-François	PR	Sciences et génie des matériaux	SMS
BATTON-HUBERT	Mireille	PR2	Sciences et génie de l'environnement	FAYOL
BEIGBEDER	Michel	MA(MDC)	Informatique	FAYOL
BLAYAC	Sylvain	MA(MDC)	Microélectronique	CMP
BOISSIER	Olivier	PR1	Informatique	FAYOL
BONNEFOY	Olivier	MA(MDC)	Génie des Procédés	SPIN
BORBELY	Andras	MR(DR2)	Sciences et génie des matériaux	SMS
BOUCHER	Xavier	PR2	Génie Industriel	FAYOL
BRODHAG	Christian	DR	Sciences et génie de l'environnement	FAYOL
BRUCHON	Julien	MA(MDC)	Mécanique et ingénierie	SMS
CAMEIRAO	Ana	MA(MDC)	Génie des Procédés	SPIN
CHRISTIEN	Frédéric	PR	Science et génie des matériaux	SMS
DAUZERE-PERES	Stéphane	PR1	Génie Industriel	CMP
DEBAYLE	Johan	MR	Sciences des Images et des Formes	SPIN
DEGEORGE	Jean-Michel	MA(MDC)	Génie industriel	Fayol
DELAFOSSE	David	PR0	Sciences et génie des matériaux	SMS
DELORME	Xavier	MA(MDC)	Génie industriel	FAYOL
DESRAYAUD	Christophe	PR1	Mécanique et ingénierie	SMS
DJENIZIAN	Thierry	PR	Science et génie des matériaux	CMP
DOUCE	Sandrine	PR2	Sciences de gestion	FAYOL
DRAPIER	Sylvain	PR1	Mécanique et ingénierie	SMS
FAUCHEU	Jenny	MA(MDC)	Sciences et génie des matériaux	SMS
FAVERGEON	Loïc	CR	Génie des Procédés	SPIN
FEILLET	Dominique	PR1	Génie Industriel	CMP
FOREST	Valérie	MA(MDC)	Génie des Procédés	CIS
FRACZKIEWICZ	Anna	DR	Sciences et génie des matériaux	SMS
GARCIA	Daniel	MR(DR2)	Sciences de la Terre	SPIN
GAVET	Yann	MA(MDC)	Sciences des Images et des Formes	SPIN
GERINGER	Jean	MA(MDC)	Sciences et génie des matériaux	CIS
GOEURIOT	Dominique	DR	Sciences et génie des matériaux	SMS
GONDRAN	Natacha	MA(MDC)	Sciences et génie de l'environnement	FAYOL
GONZALEZ FELIU	Jesus	MA(MDC)	Sciences économiques	FAYOL
GRAILLOT	Didier	DR	Sciences et génie de l'environnement	SPIN
GROSSEAU	Philippe	DR	Génie des Procédés	SPIN
GRUY	Frédéric	PR1	Génie des Procédés	SPIN
GUY	Bernard	DR	Sciences de la Terre	SPIN
HAN	Woo-Suck	MR	Mécanique et ingénierie	SMS
HERRI	Jean Michel	PR1	Génie des Procédés	SPIN
KERMOUCHE	Guillaume	PR2	Mécanique et Ingénierie	SMS
KLOCKER	Helmut	DR	Sciences et génie des matériaux	SMS
LAFOREST	Valérie	MR(DR2)	Sciences et génie de l'environnement	FAYOL
LERICHE	Rodolphe	CR	Mécanique et ingénierie	FAYOL
MALLIARAS	Georges	PR1	Microélectronique	CMP
MOLIMARD	Jérôme	PR2	Mécanique et ingénierie	CIS
MOUTTE	Jacques	CR	Génie des Procédés	SPIN
NEUBERT	Gilles			FAYOL
NIKOLOVSKI	Jean-Pierre	Ingénieur de recherche	Mécanique et ingénierie	CMP
NORTIER	Patrice	PR1	Génie des Procédés	SPIN
O CONNOR	Rodney Philip	MA(MDC)	Microélectronique	CMP
OWENS	Rosin	MA(MDC)	Microélectronique	CMP
PERES	Véronique	MR	Génie des Procédés	SPIN
PICARD	Gauthier	MA(MDC)	Informatique	FAYOL
PIJOLAT	Christophe	PR0	Génie des Procédés	SPIN
PINOLI	Jean Charles	PR0	Sciences des Images et des Formes	SPIN
POURCHEZ	Jérémy	MR	Génie des Procédés	CIS
ROUSSY	Agnès	MA(MDC)	Microélectronique	CMP
ROUSTANT	Olivier	MA(MDC)	Mathématiques appliquées	FAYOL
SANAUR	Sébastien	MA(MDC)	Microélectronique	CMP
STOLARZ	Jacques	CR	Sciences et génie des matériaux	SMS
TRIA	Assia	Ingénieur de recherche	Microélectronique	CMP
VALDIVIESO	François	PR2	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	PR0	Génie industriel	CIS
YUGMA	Gallian	CR	Génie industriel	CMP

Mise à jour : 03/02/2017

Remerciements

C'est au sein du laboratoire SAS (pour "Systèmes et Architectures Sécurisés"), situé à Gardanne dans les Bouches-du-Rhône, que j'ai eu la chance de mener mes travaux de recherche, dont la réalisation a été possible grâce à une implication du CEA (pour "Commissariat à l'énergie atomique et aux énergies alternatives"), de la DGA (pour "Direction Générale de l'Armement") et de l'INRIA (pour "Institut National de Recherche en Informatique et en Automatique").

De ce fait, je tiens tout d'abord à remercier tout le personnel du laboratoire SAS pour leur accueil au sein de l'équipe.

Je souhaite ensuite remercier Christophe Clavier et François-Xavier Standaert d'avoir accepté d'examiner mes travaux de recherche en qualité de rapporteurs de thèse, ainsi que Lilian Bossuet, Benoît Gérard et Louis Goubin d'avoir accepté de faire partie de mon jury de thèse.

Une thèse n'étant pas une aventure solitaire, je tiens naturellement à adresser mes plus vifs remerciements à Anne Canteaut, Jacques Fournier et Renaud Sirdey, qui ont toujours assuré un excellent suivi de mes travaux de thèse, et ce malgré la distance, tout au long de ces trois années.

Je les remercie sincèrement pour la confiance qu'ils m'ont accordée, pour leur accueil lors de nos rencontres à Paris, Grenoble et Saclay, pour leur disponibilité et pour leurs précieux conseils, notamment au travers de leurs relectures qui ont permis de nettement améliorer la qualité de chaque publication ainsi que de ce manuscrit.

Merci à vous trois d'avoir partagé avec moi votre expertise, et de m'avoir aidé à trouver rapidement une solution à chaque problème qui s'est présenté.

Je tiens ensuite à adresser des remerciements particuliers à David El-Baze et Maxime Lecomte, qui m'ont aidé à mieux comprendre les rouages d'un FPGA, ainsi qu'à Mounia Kharbouche et Elias Kharbouche, qui m'ont expliqué le fonctionnement d'un inverseur CMOS et m'ont généreusement proposé leur aide pour simuler la valeur de la tension électrique en différentes positions sur ce dernier.

J'ai passé trois années riches en rencontres et en échanges, et je souhaite donc plus généralement remercier toutes les personnes que j'ai eu le plaisir de croiser au cours de ma thèse, que je n'ai pas citées dans ces quelques lignes.

Enfin, mes derniers remerciements ne peuvent aller qu'à ma famille, et en particulier à mon épouse, Lauriane, pour m'avoir soutenu à chaque instant de cette aventure.

À Lauriane.

> *Le monde est si vide si l'on n'y imagine que montagnes, fleuves et villes, mais d'y savoir quelqu'un avec qui l'on s'entend, avec qui l'on peut vivre en silence, c'est ce qui fait de ce globe un jardin habité.*
>
> **— Johann Wolfgang von Goethe**

Liste des acronymes

Acronyme	Description	Occurrences
AES	Advanced Encryption Standard	Pages 8 à 152
ANSSI	Agence Nationale de la Sécurité des Systèmes d'Information	Page 13
APN	Almost Perfect Nonlinear	Pages 115 à 116
ARX	Addition-Rotation-XOR	Pages 38 à 144
ASIC	Application-Specific Integrated Circuit	Page 13
CBC	Cipher Block Chaining	Page 11
CCD	Charge Coupled Device	Page 24
CEA	Commissariat à l'énergie atomique et aux énergies alternatives	Page 16
CEMA	Correlation ElectroMagnetic Analysis	Pages 18 à 140
CMOS	Complementary Metal-Oxyde Semiconductor	Pages 20 à 38
CPA	Correlation Power Analysis	Page 29
CRT	Chinese Remainder Theorem	Page 51
CTR	CounTeR	Pages 12 à 85
DFA	Differential Fault Analysis	Pages 18 à 136
DGA	Direction Générale de l'Armement	Page 16
DoS	Denial of Service	Pages 3 à 53
DPA	Differential Power Analysis	Page 29
EEPROM	Electrically-Erasable Programmable Read-Only Memory	Page 47
FHE	Fully Homomorphic Encryption	Page 18
FPGA	Field-Programmable Gate Array	Pages 13 à 78
FSM	Finite State Machine	Page 78
GARFIELD	Guaranteed yet Affordable Resistance against Fault Injection for Embedded Lightweight Devices	Pages 17 à 165
IoT	Internet of Things	Pages 1 à 141
IIR	Intra-Instruction Redundancy	Pages 87 à 153

Acronyme	Description	Occurrences
INRIA	Institut National de Recherche en Informatique et en Automatique	Page 16
IP	Internet Protocol	Page 2
IRC	Internal Redundancy Countermeasure	Pages 17 à 185
ISA	Instruction Set Architecture	Pages 93 à 110
ISW	Ishai-Sahai-Wagner	Pages 40 à 43
IV	Initialization Vector	Pages 9 à 183
L-box	Linear box	Page 118
LAN	Local Area Network	Page 2
LFSR	Linear Feedback Shift Register	Page 86
MAC	Message Authentication Code	Page 67
MEDP	Maximum Expected Differential approximation Probability	Pages 128 à 129
MELP	Maximum Expected Linear Potential	Pages 128 à 129
MIA	Mutual Information Analysis	Page 29
MOSFET	Metal Oxide Semiconductor Field Effect Transistor	Page 20
NIST	National Institute of Standards and Technology	Pages 18 à 143
NVM	Non-Volatile Memory	Page 47
OFB	Output FeedBack	Pages 11 à 85
PAN	Personal Area Network	Page 2
PGCD	Plus Grand Commun Diviseur	Page 51
RFID	Radio Frequency Identification	Pages 2 à 3
RSA	Rivest-Shamir-Adleman	Pages 8 à 53
S-box	Substitution box	Pages 8 à 134
SAS	Systèmes et Architectures Sécurisés	Page 16
SCADA	Supervisory Control And Data Acquisition	Pages 2 à 3
SIMD	Single Instruction on Multiple Data	Pages 91 à 184
SNR	Signal-to-Noise Ratio	Pages 25 à 38
SPN	Substitution-Permutation Network	Pages 11 à 141
TAE	Tweakable Authenticated Encryption	Page 77
UART	Universal Asynchronous Receiver Transmitter	Pages 22 à 137
WAN	Wide Area Network	Page 2
WDDL	Wave Dynamic Differential Logic	Page 38
XOR	eXclusive OR	Pages 9 à 177

Liste des symboles

Symbole	Description	Occurrences
$\#$	Cardinal d'un ensemble	Pages 114 à 128
$\|\|$	Concaténation de deux mots	Page 34
$\mathbb{F}_2$	Corps fini de cardinal 2	Pages 118 à 132
$\sim X$ ou $\overline{X}$	Inversion binaire de X	Pages 151 à 181

Publications de l'auteur

[Adomnicai et al. 2016] - Adomnicai, A., Lac, B., Canteaut, A., Fournier, J. J. A., Masson, L., Sirdey, R. et Tria, A. (2016). "On the importance of considering physical attacks when implementing lightweight cryptography". In : *Lightweight Cryptography Workshop - LCW 2016*. Gaithersburg, Maryland : National Institute of Standards & Technology.

[Canteaut et al. 2017] - Canteaut, A., Carpov, S., Fontaine, C., Fournier, J. J. A., Lac, B., Naya-Plasencia, M., Sirdey, R. et Tria, A. (2017). "End-to-end data security for IoT : from a cloud of encryptions to encryption in the cloud". In : *Computer & Electronics Security Applications Rendez-vous - C&ESAR 2017*. Rennes, France.

[Lac et al. 2016] - Lac, B., Beunardeau, M., Canteaut, A., Fournier, J. J. A. et Sirdey, R. (2016). "A First DFA on PRIDE : From Theory to Practice". In : *CRiSIS 2016*. Sous la dir. de Frédéric Cuppens, Nora Cuppens, Jean-Louis Lanet et Axel Legay. T. 10158. LNCS. Roscoff, France : Springer, Heidelberg, Germany, p. 214–238.

[Lac et al. 2017] - Lac, B., Canteaut, A., Fournier, J. et Sirdey, R. (2017). "DFA on LS-Designs with a Practical Implementation on SCREAM". In : *COSADE 2017*. Sous la dir. de Sylvain Guilley. T. 10348. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 223–247.

[Lac et al. 2018] - Lac, B., Canteaut, A., Fournier, J. J. A. et Sirdey, R. (2018). "Thwarting Fault Attacks Against Lightweight Cryptography Using SIMD Instructions". In : *International Symposium on Circuits and Systems - ISCAS 2018*. Florence, Italie : IEEE Computer Society, p. 1–5.

Sommaire

Remerciements iii

Liste des acronymes viii

Liste des symboles ix

Publications de l'auteur xi

1 Introduction à la sécurité des objets connectés 1
- 1.1 Contexte général de l'IoT . . . 2
- 1.2 Enjeux de sécurité des objets connectés . . . 3
- 1.3 La cryptographie : un outil de sécurité . . . 4
  - 1.3.1 Histoire de la cryptographie . . . 4
  - 1.3.2 Cryptographie moderne . . . 6
- 1.4 Nouveaux enjeux cryptographiques . . . 13
  - 1.4.1 Cryptographie légère . . . 13
  - 1.4.2 Chemins d'attaques émergents . . . 15
- 1.5 Positionnement de la thèse . . . 16
- 1.6 Structure du manuscrit de thèse . . . 16
- 1.7 Contributions . . . 18

2 Attaques par observation et contre-mesures 19
- 2.1 Grandeurs physiques observables . . . 20
  - 2.1.1 Consommation de courant . . . 20
  - 2.1.2 Rayonnement électromagnétique . . . 22
  - 2.1.3 Température du circuit intégré . . . 23
  - 2.1.4 Émissions de photons . . . 24
  - 2.1.5 Temps d'exécution . . . 24
- 2.2 Dépendances avec les données . . . 25
  - 2.2.1 Traitement du signal . . . 25
  - 2.2.2 Analyse simple . . . 26
  - 2.2.3 Analyse différentielle . . . 27
  - 2.2.4 Analyse par modèles . . . 30
- 2.3 CEMA généralisée aux chiffrements SPN . . . 30
  - 2.3.1 Application en fonction de l'implémentation . . . 31
  - 2.3.2 Exemple de CEMA conduite en laboratoire . . . 33

2.4 Contre-mesures . . . 38
2.4.1 Diminuer les fuites . . . 38
2.4.2 Augmenter le bruit . . . 39
2.5 Le parallélisme par tranches de bits pour faciliter le masquage . . . 42
2.6 Amélioration sur une architecture 32 bits . . . 42

3 Attaques par perturbation et contre-mesures 45
3.1 Techniques d'injection de fautes . . . 46
3.1.1 Perturbation de la tension d'alimentation . . . 46
3.1.2 Modification de la température . . . 47
3.1.3 Injection électromagnétique . . . 47
3.1.4 Perturbation du signal d'horloge . . . 48
3.1.5 Injection par rayonnement lumineux . . . 49
3.2 Exploitation des fautes obtenues . . . 49
3.2.1 Analyse simple de fautes . . . 50
3.2.2 Analyse différentielle de fautes . . . 51
3.2.3 Analyse d'erreurs sur l'exécution . . . 53
3.3 DFA généralisée aux chiffrements SPN . . . 54
3.3.1 Application aux différentes structures SPN . . . 54
3.3.2 Estimation du nombre de fautes requises . . . 62
3.3.3 Simulation avec un modèle de faute idéal . . . 67
3.3.4 Exemples de DFA conduite en laboratoire . . . 69
3.4 Contre-mesures . . . 81
3.4.1 Matérielles . . . 82
3.4.2 Logicielles . . . 82
3.4.3 Évaluation des contre-mesures face à la DFA . . . 85
3.5 Intra-Instruction Redundancy . . . 87

4 Internal Redundancy Countermeasure 89
4.1 Principe général . . . 90
4.1.1 Prérequis . . . 92
4.1.2 Implémentation . . . 93
4.2 Déploiement de l'IRC sur les schémas de chiffrement à clé secrète . . . 94
4.2.1 Application aux chiffrements par blocs . . . 95
4.2.2 Application aux chiffrements à flot . . . 96
4.3 Implémentation efficace du masquage en utilisant l'IRC . . . 99
4.3.1 Masquage de premier ordre . . . 99
4.3.2 Masquage d'ordre supérieur . . . 100
4.4 Tests pratiques conduits en laboratoire . . . 101
4.4.1 Implémentation de PRIDE protégée par l'IRC . . . 102
4.4.2 Implémentation de TRIVIUM protégée par l'IRC . . . 104
4.5 Généralisation . . . 108

5 GARFIELD : un schéma de chiffrement pour l'IoT **111**
5.1 Pourquoi GARFIELD ? . . . 112
5.2 Spécifications . . . 113
5.2.1 Structure générale . . . 113
5.2.2 Couche de substitution . . . 113
5.2.3 Étage linéaire . . . 117
5.2.4 Constantes de tour . . . 122
5.2.5 Implémentations selon les besoins en sécurité . . . 123
5.3 Performances . . . 124
5.4 Résistance aux attaques mathématiques . . . 127
5.4.1 Cryptanalyse linéaire et différentielle . . . 128
5.4.2 Attaques d'ordre supérieur, par interpolation et variantes . . . 129
5.4.3 Attaques par sous-espace invariant . . . 130
5.4.4 Attaques par invariant non-linéaire . . . 132
5.5 Mise en œuvre d'attaques physiques . . . 133
5.5.1 DFA menée en laboratoire sur GARFIELD . . . 134
5.5.2 CEMA menée en laboratoire sur GARFIELD . . . 137

6 Conclusion et perspectives **141**
6.1 Conclusion . . . 141
6.2 Perspectives . . . 142

A Implémentations utilisées **145**

Références **187**

Abstract/Résumé **213**

Introduction à la sécurité des objets connectés

Chapitre 1

> *Une science ne débute qu'avec une délimitation suffisante des problèmes susceptibles de circonscrire un terrain de recherche sur lequel l'accord des esprits est possible.*
>
> **— Jean Piaget**

Introduction

Les réseaux informatiques prennent une place importante dans notre quotidien. Utilisés par diverses applications telles que le World Wide Web, la messagerie électronique ou le transfert de fichiers informatiques, ils nous permettent aussi bien d'acquérir de l'information que de communiquer et d'échanger des données en permanence. Il est aujourd'hui envisageable de connecter n'importe quel objet de notre quotidien à un réseau, et on parle d'Internet des Objets (IoT en anglais pour "Internet of Things") pour désigner l'ensemble de ces objets connectés. L'IoT a de nombreux domaines applicatifs et offre ainsi un potentiel immense pour les entreprises, les industries et les utilisateurs. Après avoir présenté les différentes possibilités d'applications offertes par l'IoT, nous analysons les enjeux de sécurité des objets connectés, liés d'un côté au nombre important de données qu'ils manipulent, et d'un autre au fait qu'ils sont souvent en milieu hostile et accessibles physiquement. Nous décrivons ensuite le positionnement de la thèse au sein de ce vaste domaine qu'est l'IoT avant de conclure par la structure de ce manuscrit et les contributions apportées.

Sommaire

1.1 Contexte général de l'IoT **2**
1.2 Enjeux de sécurité des objets connectés **3**
1.3 La cryptographie : un outil de sécurité **4**
1.3.1 Histoire de la cryptographie 4
1.3.2 Cryptographie moderne 6
1.4 Nouveaux enjeux cryptographiques **13**
1.4.1 Cryptographie légère 13
1.4.2 Chemins d'attaques émergents 15
1.5 Positionnement de la thèse **16**
1.6 Structure du manuscrit de thèse **16**
1.7 Contributions **18**

1.1 Contexte général de l'IoT

Les réseaux informatiques n'assurent plus seulement aujourd'hui une interconnexion des systèmes informatisés, mais également de tous les objets qui disposent d'une adresse IP (pour "Internet Protocol") et qui ont la capacité de transférer des données sur un réseau. Il peut s'agir de puces électroniques, de stimulateurs cardiaques, de capteurs, de technologies d'identification automatique par radio fréquence (RFID en anglais pour "Radio Frequency Identification") ou de tout autre système d'acquisition et de contrôle de données (SCADA en anglais pour "Supervisory Control And Data Acquisition") connecté à un réseau informatique.

L'IoT apparaît ainsi sous différentes formes, et utilise différents réseaux informatiques tels que les réseaux personnels (PAN en anglais pour "Personal Area Network"), les réseaux locaux (LAN en anglais pour "Local Area Network") ou les réseaux étendus (WAN en anglais pour "Wide Area Network"). Il est donc naturel de rencontrer l'IoT dans de nombreux domaines très divers comme l'illustre la figure 1.1.

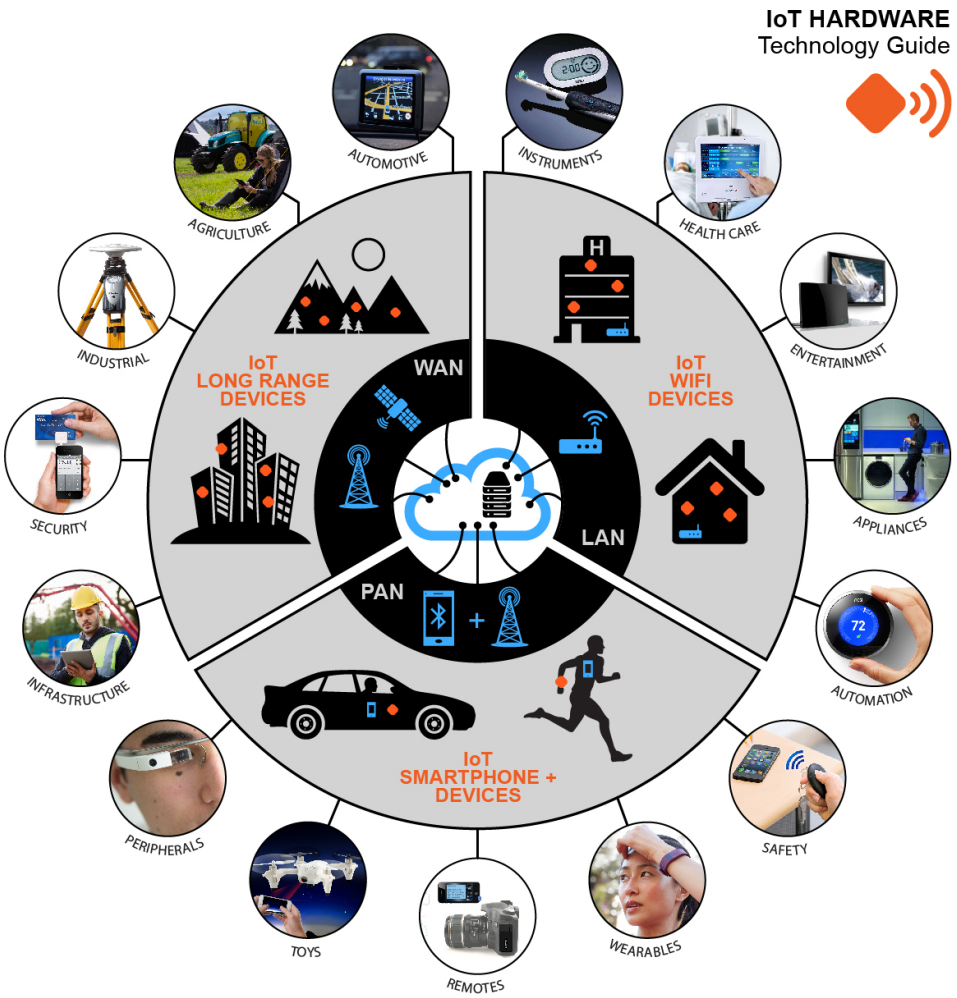


Figure 1.1.: Illustration des différents domaines touchés par l'IoT. [BARTON 2017]

On retrouve ainsi des objets connectés pour la surveillance environnementale dans le but de détecter d'éventuelles catastrophes et d'augmenter les chances de s'en prémunir, pour les transports afin de déceler des anomalies dans les véhicules, pour la gestion de l'énergie, pour la téléopération qui consiste à contrôler des objets à distance tels les drones, pour la santé afin de stocker par exemple toutes les données médicales d'un patient, pour la gestion des infrastructures ou pour le contrôle des processus de fabrication afin de faciliter le routage, l'inventaire et éviter les pertes.

Selon une récente étude de Gatner [Van Der Meulen 2017], il y aurait aujourd'hui plus de 8 milliards d'objets connectés à Internet dont 63% d'applications grand public, et il y en aura plus de 20 milliards d'ici 2020. L'IoT va donc générer une importante quantité de données manipulées en permanence, ce qui amènera les entreprises à créer de nouveaux services. L'IoT de demain nous offre ainsi de grandes possibilités, et il ne tient qu'à nous de les explorer.

1.2 Enjeux de sécurité des objets connectés

Les objets connectés apportent de nombreux avantages aux entreprises, aux industries et aux utilisateurs, mais également à d'éventuels attaquants qui peuvent y voir une opportunité pour altérer le cadre d'utilisation d'un service lié à l'IoT, détourner des informations sensibles ou secrètes manipulées par un objet connecté ou provoquer un déni de service (DoS en anglais pour "Denial of Service") d'une application ou de tout un système.

Il est donc essentiel de prendre en considération la valeur des données manipulées et les risques que peut engendrer une mauvaise utilisation de ces applications. Là où certains objets connectés semblent sans risque, comme par exemple un capteur de température ou une technologie RFID utilisée pour identifier un produit, il y en a d'autres qui se révèlent beaucoup plus sensibles, comme un stimulateur cardiaque, un téléphone mobile employé pour effectuer un paiement ou un système SCADA d'authentification par biométrie utilisé pour un contrôle d'accès à une zone privée.

Ainsi, les besoins en résilience des réseaux, la confidentialité et l'intégrité des données échangées, la protection de la vie privée des utilisateurs et l'intégrité et l'authenticité des circuits intégrés déployés sont de réelles problématiques de sécurité aujourd'hui largement identifiées et au centre des débats. Il est d'autant plus important de répondre à ces besoins en sécurité avant le déploiement d'objets connectés à si grande échelle, car apporter des correctifs a posteriori est extrêmement difficile, en raison du nombre élevé d'objets déployés, et de leur répartition géographique qui s'étend généralement sur une zone très vaste.

La nécessité de protéger des messages n'est pas une nouveauté pour autant. Aujourd'hui sous forme de données numériques, autrefois écrits sur des parchemins, l'être humain a de tout temps eu besoin de sécuriser ses communications, et il existe une science qui essaie d'y répondre depuis l'aube de l'humanité : la cryptographie.

1.3 La cryptographie : un outil de sécurité

La cryptographie vise à assurer des fonctionnalités qui sont essentielles pour protéger l'information, dont les principales sont les suivantes :

- La confidentialité : elle assure que les données ne sont compréhensibles qu'aux personnes ou entités autorisées.
- L'intégrité : elle assure que les données ne sont pas altérées de façon non autorisée pendant leur transmission ou leur stockage.
- L'authentification : elle prouve que la personne ou l'entité qui envoie les messages est exactement celle qu'elle prétend être.
- La non-répudiation : elle assure que la personne ou l'entité qui envoie les messages ne peut pas nier être à l'origine de l'envoi.

La cryptographie permet ainsi de protéger un message clair, généralement par l'utilisation d'un secret, afin d'obtenir un chiffré compréhensible uniquement par l'utilisateur légitime. Il s'agit de l'une des deux disciplines de la cryptologie, l'autre étant la cryptanalyse qui consiste à analyser un chiffré, sans la connaissance du secret, afin d'en déduire des informations sur le message clair ou sur le secret.

La cryptographie n'est considérée que depuis peu comme une science, mais il s'agit d'une discipline qui existe depuis l'Antiquité.

1.3.1 Histoire de la cryptographie

L'apparition de la cryptographie remonte à la fin de l'Ancien Empire égyptien vers 2350 avant Jésus Christ où l'on retrouve différentes techniques par permutation ou par substitution basées sur les hiéroglyphes égyptiens [Drioton 1953].

Figure 1.2.: Illustration d'une scytale.

Néanmoins, pendant de nombreux siècles, les outils cryptographiques utilisés étaient basés sur des concepts mathématiques assez simples, dont la cryptanalyse était souvent largement réalisable. On peut notamment citer la scytale utilisée par les Spartiates et illustrée en figure 1.2. Le message était écrit

sur une fine lanière de cuir ou de parchemin enroulée autour d'un bâton de bois, dont le diamètre était gardé secret, ce qui rendait le message illisible une fois la lanière déroulée. La technique utilisée applique ainsi une simple permutation sur les lettres qui composent le message, il n'est donc pas difficile de retrouver le message clair à partir du chiffré contenu sur la lanière, en utilisant par exemple plusieurs scytales de différents diamètres jusqu'à retrouver le bon diamètre.

La plupart des méthodes utilisées les siècles suivants reposaient sur des substitutions mono-alphabétiques. Une substitution mono-alphabétique consiste à remplacer l'alphabet habituel par un autre alphabet, le nouvel alphabet étant le secret de cette technique de chiffrement. Cependant, il est possible de retrouver l'alphabet d'origine en calculant la fréquence d'apparition de chaque lettre dans le chiffré et en la comparant à la fréquence des lettres dans la langue d'origine.

En 1917, G. Vernam a introduit le chiffre de Vernam, ou masque jetable, qui consiste à combiner le message clair avec une suite de caractères aléatoires au moins aussi longue que le message à chiffrer, utilisable qu'une seule fois. Bien que cette méthode offre une sécurité théorique absolue [SHANNON 1949], elle est difficilement applicable. En effet, il faut d'une part être capable de générer une suite de caractères complètement aléatoires, et d'autre part être en mesure de partager et de stocker un secret de si grande taille, qui doit être mis à jour après chaque utilisation.

Figure 1.3.: Machine Enigma. [CANTEAUT 2012]

La cryptographie et la cryptanalyse ont connu un essor important au cours de la seconde guerre mondiale, et ont eu un immense impact sur son dénouement, notamment avec les machines Enigma utilisées par les Allemands pour chiffrer leurs communications. Enigma était une machine à écrire particulière, où la pression d'une lettre sur son clavier faisait passer un courant électrique à travers un tableau modifiant l'alphabet puis à travers un réseau complexe de fils, contenus dans trois rotors, afin d'allumer une lampe indiquant la lettre chiffrée correspondante. Après avoir tapé une lettre, une rotation était appliquée sur les rotors modifiant à chaque fois la configuration du réseau. Par ce système, chaque lettre était remplacée par une lettre d'un autre alphabet, différent pour chaque nouvelle lettre à chiffrer. Le choix des rotors, leur ordre dans la machine, leur position initiale et les connexions du tableau constituaient le secret de cette technique de chiffrement. Il s'agit d'une substitution poly-alphabétique, dont la cryptanalyse est plus complexe mais reste possible. En effet, la connaissance de la méthode employée et de certaines parties connues des messages clairs, comme

les bulletins météo qui étaient transmis chaque matin à la même heure par les Allemands, a permis aux Alliés de réduire considérablement le nombre de possibilités pour les paramètres secrets utilisés, et a ainsi conduit à la cryptanalyse de certaines communications chiffrées par Enigma. Il a d'ailleurs été estimé que ces travaux ont réduit le conflit en Europe d'au minimum deux ans [HINSLEY 1992].

L'histoire nous montre que la permutation ou la substitution utilisées seules ne suffisent pas à sécuriser une communication. Pourtant, ces deux techniques constituent la base théorique de la cryptographie moderne. La substitution crée de la confusion sur le message et la permutation diffuse cette confusion.

1.3.2 — Cryptographie moderne

A. Kerckhoffs énonce en 1883 les six "desiderata de la cryptographie militaire" [KERCKHOFFS 1883]. L'un d'entre eux, que l'on nomme maintenant "principe de Kerckhoffs", revient à dire que la sécurité d'un outil cryptographique ne doit en aucun cas résider dans le secret de la méthode employée, qui doit donc pouvoir tomber sans inconvénient entre les mains de l'ennemi.

Il s'agit d'un pilier fondamental de la cryptographie moderne, le secret réside généralement dans ce que l'on appelle aujourd'hui une clé, qui doit pouvoir prendre suffisamment de valeurs différentes afin d'empêcher une attaque exhaustive qui consiste à tester toutes les valeurs de clé possibles.

L'ensemble des techniques utilisées aujourd'hui peut se diviser en quatre principales catégories : les fonctions de hachage cryptographiques, les codes d'authentification de message (MAC en anglais pour "Message Authentication Code"), les schémas de chiffrement à clé publique et les schémas de chiffrement à clé secrète.

Un schéma de chiffrement est composé des éléments suivants :

- Un ensemble fini de messages clairs noté $\mathcal{P}$,
- Un ensemble fini de messages chiffrés noté $\mathcal{C}$,
- Un ensemble fini de clés noté $\mathcal{K}$ et,
- Une fonction de chiffrement $\mathcal{E}_{\mathrm{K}} : \mathcal{P} \longrightarrow \mathcal{C}$ avec K dans $\mathcal{K}$ et une fonction de déchiffrement $\mathcal{D}_{\mathrm{K}'} : \mathcal{C} \longrightarrow \mathcal{P}$ avec K' dans $\mathcal{K}$ telles que $\mathcal{D}_{\mathrm{K}'} \circ \mathcal{E}_{\mathrm{K}}(\mathrm{M}) = \mathrm{M}$ pour tout message clair M dans $\mathcal{P}$.

1.3.2.1 – Fonctions de hachage cryptographiques

Une fonction de hachage cryptographique calcule une empreinte d'un message afin d'être en mesure de l'identifier rapidement. Il s'agit d'une fonction à sens unique, c'est-à-dire qui est difficile[1] à inverser.

Afin d'assurer l'intégrité d'un message, une fonction de hachage cryptographique h doit respecter les propriétés suivantes [ROGAWAY et al. 2004] :

- La résistance aux attaques par pré-image : pour toute empreinte E d'un message inconnu, il doit être difficile[1] de trouver un message M tel que $h(\mathrm{M}) = \mathrm{E}$.
- La résistance aux attaques en seconde pré-image : pour tout message M_1, il doit être difficile[1] d'obtenir un message M_2 tel que $\mathrm{M}_1 \neq \mathrm{M}_2$ et $h(\mathrm{M}_1) = h(\mathrm{M}_2)$.
- La résistance aux attaques par collision : il doit être difficile[1] de trouver deux messages M_1 et M_2 tels que $\mathrm{M}_1 \neq \mathrm{M}_2$ et $h(\mathrm{M}_1) = h(\mathrm{M}_2)$.

De plus, les empreintes de deux messages quasi-identiques sont généralement très différentes. Il est à noter que la résistance aux attaques par collision implique la résistance aux attaques en seconde pré-image [A. J. MENEZES et al. 1996].

Parmi les fonctions de hachage cryptographiques les plus connues, on peut notamment citer MD5 [RIVEST 1992], SHA-1 [NIST 1995], SHA-2 [NIST 1995] ou SHA-3 [PRENEEL 2008] par ordre d'apparition.

1.3.2.2 – Codes d'authentification de message

De façon similaire aux fonctions de hachage cryptographiques, un MAC calcule une empreinte d'un message, mais cette fois en utilisant une clé secrète afin d'assurer à la fois l'intégrité du message et son authenticité.

Il est donc nécessaire qu'un MAC vérifie les mêmes propriétés qu'une fonction de hachage cryptographique afin d'effectuer une authentification.

Parmi les codes d'authentification de message les plus répandus, on peut notamment citer NMAC [BELLARE et al. 1996], HMAC [KRAWCZYK et al. 1997], CBC-MAC [PETRANK et al. 2000] ou GMAC [DWORKIN 2007] par ordre d'apparition.

1.3.2.3 – Schémas de chiffrement à clé publique

Un schéma de chiffrement à clé publique, ou asymétrique, manipule deux clés. L'une est publique, transmissible sans restriction, et est utilisée par la fonction de

1. difficile signifie qu'un attaquant ne doit pas pouvoir y parvenir dans un temps raisonnable.

chiffrement. L'autre est privée, connue uniquement de son propriétaire, et est utilisée par la fonction de déchiffrement.

Un des tout premiers schémas de chiffrement à clé publique, et le plus célèbre, a été proposé en 1978 par R. Rivest, A. Shamir et L. Adleman et est désigné par RSA (pour "Rivest-Shamir-Adleman") [RIVEST et al. 1978]. Le RSA repose sur l'exponentiation modulaire, sa clé publique est composée d'un entier e et d'un entier n qui doit être le produit de deux nombres premiers p et q, et sa clé privée est un entier d tel que $ed \equiv 1$ modulo $(p-1)(q-1)$. Pour chiffrer un message M, il suffit alors de calculer $C = M^e$ modulo n, qui pourra bien être déchiffré en calculant $M = C^d$ modulo n, d'après le théorème d'Euler en arithmétique modulaire [EULER 1763].

Il est à noter que les fonctions de chiffrement et de déchiffrement du RSA peuvent être utilisées pour effectuer une authentification. En effet, il suffit d'appliquer la fonction de déchiffrement avec la clé privée sur un message clair connu de tous afin d'obtenir un chiffré à transmettre pour vérification. N'importe qui peut alors appliquer la fonction de chiffrement avec la clé publique sur le chiffré afin de s'assurer que le message ainsi obtenu correspond bien au message clair.

On peut également citer le cryptosystème d'ElGamal [ELGAMAL 1984] ou le cryptosystème de Paillier [PAILLIER 1999] parmi les autres schémas de chiffrement à clé publique les plus connus.

1.3.2.4 – Schémas de chiffrement à clé secrète

Un schéma de chiffrement à clé secrète, ou symétrique, utilise une seule clé qui doit être connue des deux interlocuteurs. La fonction de chiffrement et la fonction de déchiffrement effectuent ainsi des opérations avec la même clé secrète.

Le plus célèbre schéma de chiffrement à clé secrète a été proposé en 1998 par J. Daemen et V. Rijmen sous le nom de Rijndael avant d'être choisi comme standard [DAEMEN et al. 2002] sous le nom d'AES (pour "Advanced Encryption Standard") [NIST 2001]. L'AES prend en entrée un bloc de taille fixe, et effectue 10 tours composés d'une couche de substitution qui utilise une table de substitution (S-box en anglais pour "Substitution box"), d'un étage linéaire, et d'une couche d'addition avec une clé de tour générée à partir de la clé secrète avec un algorithme de cadencement de clés ("key schedule" en anglais).

L'ensemble des schémas de chiffrement à clé secrète actuels peut encore se diviser en deux principales catégories : les chiffrements à flot et les chiffrements par blocs.

1.3.2.4.a - Chiffrements à flot

Proche de l'idée du masque jetable, un chiffrement à flot synchrone génère une suite binaire pseudo-aléatoire, c'est-à-dire qui a des caractéristiques proches de celles d'une suite aléatoire, appelée suite chiffrante ("keystream" en anglais). Il effectue alors une opération inversible entre le message clair et la suite chiffrante obtenue, il s'agit généralement d'un OU exclusif (XOR en anglais pour "eXclusive OR") effectué bit à bit, ce qui permet de traiter les données à chiffrer en continu.

La suite chiffrante est obtenue à partir de la clé secrète et d'un vecteur d'initialisation (IV en anglais pour "Initialization Vector") qui peut être commun à l'expéditeur et au récepteur comme par exemple la valeur d'un compteur de message ou bien transmis en clair avec le message. L'IV doit être changé après chaque utilisation, afin qu'une même suite chiffrante ne soit utilisée qu'une seule fois.

Un chiffrement à flot est généralement aujourd'hui constitué de deux étapes :

- Une étape d'initialisation : une fonction d'initialisation $\mathcal{I}$ génère un état interne initial IS_0 à partir de la clé secrète et de l'IV.
- Une étape de génération de la suite chiffrante : une fonction de transition $\mathcal{T}$ et une fonction d'extraction $\mathcal{E}$ sont appliquées autant de fois que nécessaire sur l'état interne afin de générer une longueur suffisante de suite chiffrante. À chaque itération, la fonction $\mathcal{T}$ modifie la valeur de l'état interne et la fonction $\mathcal{E}$ retourne une partie de la suite chiffrante, généralement un seul bit.

La figure 1.4 illustre la structure d'un chiffrement à flot appliqué à un message clair M en utilisant une clé secrète K. Le k-ème bit de M (resp. de la suite chiffrante générée KS et du chiffré obtenu C) est noté M_k (resp. KS_k et C_k).

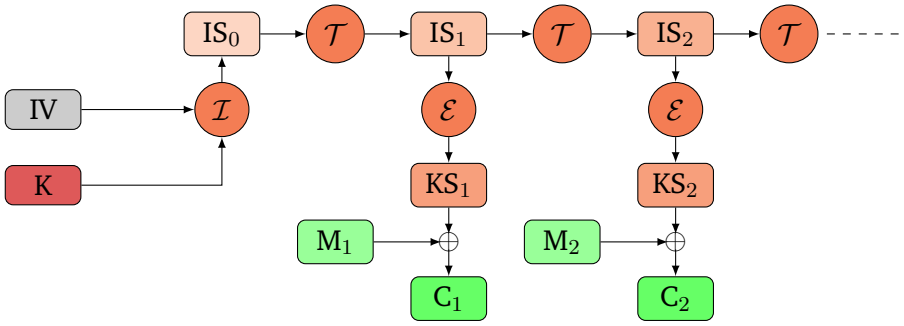


Figure 1.4.: Structure d'un chiffrement à flot.

Un chiffrement à flot utilise la fonction de chiffrement avec les mêmes paramètres pour effectuer le déchiffrement. En effet, il génère de cette façon la même suite chiffrante avec la clé secrète et l'IV utilisés lors du chiffrement, puis l'additionne au chiffré afin de retrouver les données initiales.

Parmi les chiffrement à flots les plus connus, on peut notamment citer RC4 [CypherPunks 1994], A5/1 [Briceno et al. 1999], E0 [Fluhrer et al. 2001], Trivium [De Cannière 2006], Grain [Hell et al. 2007], MICKEY [Babbage et al. 2008] ou ChaCha [Bernstein 2008] par ordre d'apparition.

Certaines attaques lors de la cryptanalyse d'un chiffrement à flot exploitent les propriétés statistiques du générateur de suites pseudo-aléatoires utilisé par le schéma de chiffrement. Il existe alors un certain nombre de tests statistiques qui doivent être vérifiés par le générateur afin de se prémunir de ces attaques, comme les tests de normalité et de compression décrits par D. E. Knuth [Knuth 1998] et S. W. Golomb [Golomb 1981] ou les tests du NIST [Bassham et al. 2010], dont la vérification est généralement indispensable mais qui sont néanmoins bien souvent insuffisants.

Par exemple, une suite binaire pseudo-aléatoire doit avoir autant de 0 que de 1, et autant de plages de 0 que de plages de 1. Une plage de 0 (resp. de 1) est un ensemble de 0 (resp. de 1) consécutifs encadrés par deux 1 (resp. deux 0).

1.3.2.4.b - Chiffrements par blocs

Un chiffrement par blocs est une fonction bijective de l'ensemble des mots binaires de taille fixe paramétrée par la clé secrète. Il se compose généralement de plusieurs tours, tous quasi-identiques, qui utilisent des clés de tour extraites à partir de la clé secrète avec un algorithme de cadencement de clés. Parmi les chiffrements par blocs, les deux principales familles les plus utilisées sont les suivantes :

- Les réseaux de Feistel : ils traitent le bloc de données en plusieurs parties. À chaque tour, ils effectuent des opérations impliquant certaines parties du bloc puis appliquent une permutation sur ces dernières. L'entrée peut être divisée en deux parties, il s'agit dans ce cas d'un réseau de Feistel classique, ou en plus de deux parties, on parle alors d'un réseau de Feistel généralisé. La figure 1.5 illustre la structure générale d'un réseau de Feistel classique de r tours appliqué aux deux parties de n bits du bloc en entrée L_1 et R_1, utilisant une fonction $\mathcal{F}$ avec des clés de tour RK_i pour $1 \leq i \leq r$, afin d'obtenir les deux parties de n bits du bloc en sortie L_r et R_r.

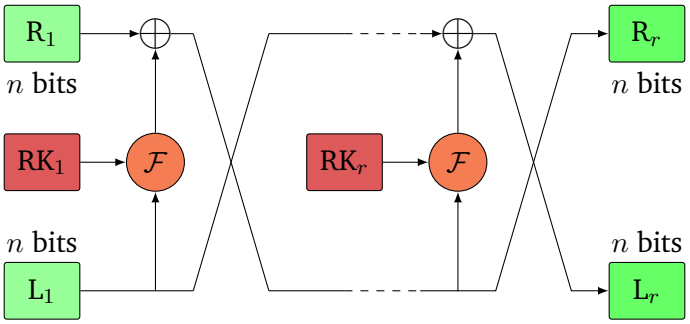


Figure 1.5.: Structure générale d'un réseau de Feistel classique.

Parmi les réseaux de Feistel les plus répandus, on peut notamment citer le DES [NIST 1977], le 3DES [KARN et al. 1995], Blowfish [SCHNEIER 1994], RC5 [RIVEST 1995], MISTY1 [MATSUI 1997], RC6 [RIVEST et al. 1998], Twofish [SCHNEIER et al. 1999], Camellia [AOKI et al. 2001], CLEFIA [SHIRAI et al. 2007] ou SIMON et SPECK [BEAULIEU et al. 2015].

- Les réseaux de substitution-permutation (SPN en anglais pour "Substitution-Permutation Network") : ils appliquent à chaque tour une couche de substitution, qui utilise généralement une S-box, un étage linéaire, et un XOR avec la clé de tour. Ils appliquent aussi un XOR avec une première clé de tour en entrée du chiffrement. La figure 1.6 illustre la structure générale d'un chiffrement SPN de r tours appliqué à un bloc M en entrée de n bits, utilisant une fonction $\mathcal{S}$l pour sa couche de substitution, une fonction $\mathcal{L}$l pour son étage linéaire et des clés de tour RK_i pour $0 \leq i \leq r$, afin d'obtenir un bloc C en sortie.

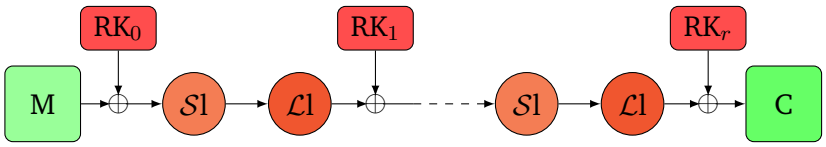


Figure 1.6.: Structure générale d'un chiffrement SPN.

Parmi les chiffrements SPN les plus connus, on peut notamment citer SHARK [RIJMEN et al. 1996], KHAZAD [BARRETO et al. 2000], l'AES [NIST 2001] que nous avons déjà mentionné, PRESENT [BOGDANOV et al. 2007], KLEIN [GONG et al. 2011], PRINCE [BORGHOFF et al. 2012] ou PRIDE [ALBRECHT et al. 2014b] par ordre d'apparition.

Afin de chiffrer un message de taille quelconque, les chiffrements par blocs utilisent un mode opératoire qui définit la manière dont on enchaîne les appels successifs au chiffrement par bloc. Les modes opératoires les plus répandus sont les suivants :

- Le mode opératoire CBC (pour "Cipher Block Chaining") : il effectue à chaque appel du chiffrement par blocs un XOR entre le dernier bloc obtenu en sortie et le prochain bloc en entrée. Il utilise un IV aléatoire pour le premier bloc. La figure 1.7 décrit le mode opératoire CBC avec une fonction $\mathcal{E}_\mathrm{K}$.

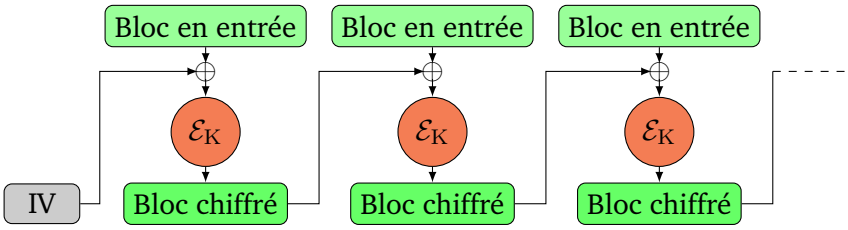


Figure 1.7.: Mode opératoire CBC.

- Le mode opératoire OFB (pour "Output FeedBack") : il applique la fonction de chiffrement successivement sur un IV aléatoire, et génère ainsi une suite chiffrante qui est ajoutée par un XOR au message clair. La figure 1.8 décrit le

mode opératoire OFB avec une fonction $\mathcal{E}_K$.

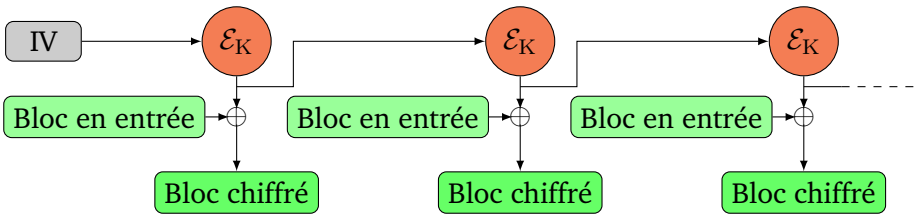


Figure 1.8.: Mode opératoire OFB.

- Le mode opératoire CTR (pour "CounTeR") : il applique la fonction de chiffrement à un compteur initialisé par un IV et incrémenté à chaque nouveau bloc. L'IV doit être choisi de telle sorte que toutes les valeurs prises par le compteur soient différentes pour une même clé. Il génère ainsi une suite chiffrante qui est ajoutée par un XOR au message clair. La figure 1.9 décrit le mode opératoire CTR avec une fonction $\mathcal{E}_K$ et un compteur Ctr initialisé par un IV.

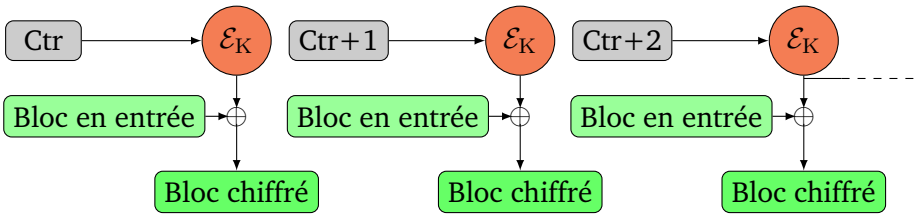


Figure 1.9.: Mode opératoire CTR.

Il est à noter que certains modes opératoires tels que le mode OFB ou le mode CTR permettent de transformer un chiffrement par blocs en un chiffrement à flot.

1.3.2.5 – Schémas de chiffrement hybrides

L'avantage de la cryptographie asymétrique réside dans la gestion des clés dans un groupe de plusieurs personnes communicantes. En effet, il suffit d'une clé publique et d'une clé privée pour chaque personne dans le cas d'un schéma de chiffrement à clé publique, alors qu'il faut une clé secrète pour chaque binôme dans le cas d'un schéma de chiffrement à clé secrète.

Cependant, la cryptographie symétrique est nettement plus rapide et moins coûteuse que la cryptographie asymétrique. Par exemple, un schéma de chiffrement à clé publique demande plusieurs dizaines de milliers de cycles d'horloge pour chiffrer un octet de données, alors qu'un AES ne demande qu'entre 20 et 30 cycles.

Le site eBACS [Bernstein et al. 2007] donne des tests de performances détaillés de plusieurs schémas de chiffrement sur différents processeurs. Dans la pratique, les schémas de chiffrement à clé publique peuvent être utilisés pour transmettre une clé secrète, qui est ensuite utilisée par un schéma de chiffrement à clé secrète pour sécuriser une communication, on parle alors de schémas de chiffrement hybrides.

Par exemple, il est possible d'utiliser l'échange de clés de Diffie-Hellman [DIFFIE et al. 1976], qui repose sur l'exponentiation dans un groupe multiplicatif, afin de partager une clé commune, et d'utiliser ensuite un schéma de chiffrement à clé secrète avec la clé commune obtenue pour effectuer une communication sécurisée.

1.4 Nouveaux enjeux cryptographiques

L'ensemble des schémas de chiffrement à clé publique et à clé secrète constitue un panel très large de primitives cryptographiques.

Les aspects de taille, temps d'exécution et consommation sont des données importantes à prendre en considération aujourd'hui lors de la conception d'un schéma de chiffrement destiné à la sécurisation des circuits intégrés déployés au sein de l'IoT.

En effet, les objets connectés ont souvent des contraintes très fortes en matière de performances, et nécessitent donc d'embarquer une primitive cryptographique à la fois peu coûteuse et fournissant un niveau de sécurité suffisant.

1.4.1 Cryptographie légère

La cryptographie légère tente de répondre aux nouveaux besoins des objets connectés, en essayant de trouver le meilleur compromis entre sécurité et performances.

Pour cela, les schémas de chiffrement légers actuels reposent sur la cryptographie symétrique [BIRYUKOV et al. 2017], et sont généralement à la limite de la sécurité recommandée, en réduisant au maximum la taille des clés secrètes utilisées et le nombre d'opérations effectuées.

Le niveau de sécurité recommandé en 2014 par l'ANSSI (pour "Agence Nationale de la Sécurité des Systèmes d'Information") se caractérise par une borne inférieure sur la taille des clés, qui est de 100 bits si le schéma de chiffrement n'est plus utilisé après 2020, et de 128 bits sinon pour la cryptographie symétrique [ANSSI 2014].

L'architecture du circuit intégré ciblé, qu'il s'agisse d'un réseau de portes programmables (FPGA en anglais pour "Field-Programmable Gate Array") ou d'un circuit intégré propre à une application (ASIC en anglais pour "Application-Specific Integrated Circuit"), doit également être prise en considération lors de la conception et du déploiement d'un schéma de chiffrement léger.

En effet, certains circuits intégrés n'utilisent que des portes logiques pour effectuer des opérations, chacune étant de ce fait directement appliquée entre des bits.

D'autres utilisent quant à eux un jeu d'instructions, qui décrit l'ensemble des opérations pouvant être effectuées par le processeur, et manipulent des registres de taille fixe égale à 8, 16, 32 ou 64 bits, et on parle dans ce cas respectivement d'architectures 8, 16, 32 ou 64 bits.

Une implémentation matérielle (resp. logicielle de 8, 16, 32 ou 64 bits) d'un schéma de chiffrement consiste alors à effectuer chaque opération directement entre des bits (resp. des mots de 8, 16, 32 ou 64 bits).

Il existe de nombreux schémas de chiffrement légers [Biryukov et al. 2017], chacun proposant d'optimiser certains des facteurs requis à la cryptographie légère, et ciblant une architecture spécifique afin d'optimiser l'implémentation correspondante.

Parmi les chiffrements à flot légers les plus répandus, on peut notamment citer E0 [Fluhrer et al. 2001], A5/1 [Briceno et al. 1999], SNOW 3G [ETSI/SAGE 2006], Trivium [De Cannière 2006], Grain [Hell et al. 2007], MICKEY [Babbage et al. 2008], ChaCha [Bernstein 2008] ou Enocoro [Watanabe et al. 2008].

Du côté des chiffrements par blocs légers les plus étudiés, on peut mentionner NOEKEON [Daemen et al. 2000], HIGHT [Hong et al. 2006], mCrypton [Lim et al. 2005], CLEFIA [Shirai et al. 2007], PRESENT [Bogdanov et al. 2007], KATAN et KTANTAN [De Cannière et al. 2009], LBlock [Wu et al. 2011], LED [Guo et al. 2011], Piccolo [Shibutani et al. 2011], KLEIN [Gong et al. 2011], PRINCE [Borghoff et al. 2012], TWINE [Suzaki et al. 2013], Zorro [Gérard et al. 2013], LEA [Hong et al. 2013], PRIDE [Albrecht et al. 2014b], Robin et Fantomas [Grosso et al. 2015], Midori [Banik et al. 2015], RECTANGLE [Zhang et al. 2015], SIMON et SPECK [Beaulieu et al. 2015], SKINNY et Mantis [Beierle et al. 2016], SPARX [Dinu et al. 2016] ou Mysterion [Journault et al. 2017a].

Il est à noter que Enocoro et Trivium sont standardisés par la norme ISO/IEC 29192-3 :2012, et CLEFIA et PRESENT par la norme ISO/IEC 29192-2 :2012 toutes deux consacrées à la cryptographie légère.

Il faut cependant être prudent quant à l'utilisation de la cryptographie légère : c'est seulement après plusieurs études par la communauté scientifique que l'on peut envisager d'utiliser un nouveau schéma de chiffrement, bien que malgré cela, rien ne permet d'assurer qu'aucune attaque ne puisse exister sur ce dernier.

1.4.2 Chemins d'attaques émergents

Les attaques mathématiques ne sont pas les seules menaces dont il faut se soucier lorsque l'on tente de sécuriser un circuit intégré pour l'IoT. En effet, les objets connectés sont généralement déployés dans des milieux hostiles, à portée de main de tout type d'attaquant, et ce sur des durées souvent indéterminées.

Les schémas de chiffrement, ou plutôt les implémentations logicielles et matérielles de ces systèmes sur les circuits intégrés, sont aujourd'hui sensibles à ce que l'on appelle les attaques physiques. Il s'agit d'observer ou de perturber le fonctionnement d'un circuit intégré afin de déduire des informations sur le secret du schéma de chiffrement qu'il utilise ou d'altérer son exécution.

Un exemple d'attaque par observation a récemment été proposé en pratique sur des ampoules connectées [Ronen et al. 2017].

L'attaque a permis d'extraire la clé secrète utilisée par un AES pour autoriser les mises à jour sur les ampoules, ce qui a rendu possible l'infection d'un réseau complet d'ampoules connectées par transmission de la mise à jour corrompue entre ampoules voisines, afin d'altérer leur fonctionnement ou de provoquer un DoS.

Quelques études ont été menées sur la sécurité de la cryptographie légère face aux attaques par observation. Certains schémas de chiffrement ont notamment intégré cette notion lors de leur conception tels que PICARO [Piret et al. 2012], Zorro [Gérard et al. 2013] ou les LS-Designs [Grosso et al. 2015].

En effet, tous ces schémas de chiffrement ont une structure facilitant le masquage, une contre-mesure efficace contre les attaques par observation que nous étudierons plus en détail dans la suite de ce manuscrit.

Par contre, la sécurité face aux attaques par perturbation a été très peu abordée pour la cryptographie légère. Pour obtenir une implémentation résistante à ce type d'attaques qui respecte les fortes contraintes de coûts et de performances liées aux objets connectés, il serait probablement plus efficace d'intégrer la nécessité de résister aux attaques physiques dès la conception des schémas de chiffrement légers, au lieu de le faire a posteriori avec le surcoût que cela risquerait d'engendrer.

Cette dernière réflexion a donné lieu à l'élaboration de ces travaux de recherche, que nous estimons nécessaires pour répondre aux besoins en matière de sécurité et performances des objets connectés.

1.5 Positionnement de la thèse

Au sein du vaste domaine qu'est l'IoT, notre étude porte donc sur la sécurisation des objets connectés face aux attaques physiques. Ces travaux de recherche ont été menés au sein du laboratoire SAS (pour "Systèmes et Architectures Sécurisés") situé à Gardanne dans les Bouches-du-Rhône, et leur réalisation a été possible grâce à une implication du CEA (pour "Commissariat à l'énergie atomique et aux énergies alternatives"), de la DGA (pour "Direction Générale de l'Armement") et de l'INRIA (pour "Institut National de Recherche en Informatique et en Automatique").

Notre étude s'intéresse, entre autres, à la définition des besoins en matière de sécurité et performances pour la cryptographie légère dans le contexte de l'IoT, à l'analyse de la résistance de divers schémas de chiffrement légers aux attaques physiques, et à la conception et à l'étude de solutions afin de s'en prémunir.

Après avoir déterminé les besoins en matière de sécurité et performances des objets connectés tels que nous les avons décrits dans ce chapitre, et afin de tenter d'y répondre, nous nous sommes fixés les principaux objectifs suivants :

- Évaluer la résistance des schémas de chiffrement légers existant dans la littérature face aux attaques par observation et par perturbation, et proposer des mises en œuvre de ces attaques sur des cibles réelles en laboratoire afin d'établir des références concrètes dans le domaine.
- Étudier les contre-mesures proposées dans la littérature pour se prémunir de ces attaques, et concevoir une solution performante et sécurisée pour l'IoT.

Nous allons détailler dans les prochains chapitres de ce manuscrit les éléments de réponse que nous avons apportés pour couvrir l'ensemble de ces objectifs.

1.6 Structure du manuscrit de thèse

Le second chapitre de ce manuscrit est consacré aux attaques par observation. Nous décrivons dans un premier temps les grandeurs physiques d'un circuit intégré qui dépendent des données manipulées et les outils statistiques qu'un attaquant peut utiliser pour exploiter ces dépendances. Nous introduisons ensuite une attaque par observation générique applicable aux chiffrements SPN et nous analysons une mise en œuvre pratique de cette attaque que nous avons conduite en laboratoire sur une cible réelle. Nous présentons à la fin les contre-mesures connues dans la littérature pour se prémunir de ces attaques et nous détaillons le cas du masquage.

Après les attaques par observation, le troisième chapitre porte sur les attaques par perturbation. Nous introduisons d'abord les différentes techniques d'injection de fautes et les méthodes permettant à un attaquant d'en exploiter les effets. Nous décrivons ensuite une attaque par perturbation générique que nous avons introduite sur les chiffrements SPN, particulièrement efficace sur les structures entrelacées [ALBRECHT et al. 2014b] et sur les LS-Designs [GROSSO et al. 2015], et nous examinons des mises en œuvre pratiques de cette attaque que nous avons conduites en laboratoire sur un micro-contrôleur Cortex-M3 et sur un FPGA. Nous énonçons enfin les contre-mesures logicielles et matérielles connues dans la littérature pour se prémunir de ces attaques et nous détaillons le cas de la redondance.

Le quatrième chapitre introduit une contre-mesure logicielle que nous proposons pour se prémunir des attaques par perturbation sur une architecture 32 bits et que nous avons nommée l'IRC (pour "Internal Redundancy Countermeasure"). Nous introduisons premièrement le principe de l'IRC et les prérequis nécessaires à son utilisation avant d'expliquer comment l'appliquer aux chiffrements par blocs et aux chiffrements à flot. Nous décrivons ensuite une technique pour implémenter efficacement le masquage en utilisant l'IRC et nous analysons des tests pratiques sur l'efficacité et les performances de l'IRC que nous avons menés en laboratoire sur des implémentations protégées par l'IRC. Nous présentons pour conclure une généralisation de l'IRC aux autres architectures.

Le cinquième chapitre est consacré à un nouveau chiffrement par blocs que nous avons conçu pour répondre aux besoins de sécurité et de performance des objets connectés et que nous avons nommé GARFIELD (pour "Guaranteed yet Affordable Resistance against Fault Injection for Embedded Lightweight Devices"). Nous présentons dans un premier temps les spécifications de GARFIELD et nous comparons ses performances avec d'autres schémas de chiffrement en fonction de différents niveaux de sécurité. Nous étudions ensuite sa résistance vis-à-vis des cryptanalyses classiques connues à ce jour dans la littérature. Nous analysons enfin sa résistance aux attaques physiques à partir de mises en œuvre pratiques que nous avons menées en laboratoire sur une cible réelle.

Le sixième et dernier chapitre établit un bilan des travaux effectués durant ces trois années de thèse. Nous résumons les résultats que nous avons obtenus et nous analysons les perspectives qui en découlent.

Enfin, une annexe met à disposition les différentes implémentations 32 bits que nous avons utilisées pour la réalisation de nos travaux.

1.7 Contributions

En première année de thèse, nous avons dans un premier temps porté nos travaux de recherche sur le chiffrement par blocs PRIDE [Albrecht et al. 2014b], qui est à ce jour l'un des plus performant en terme d'implémentation logicielle, et nous avons proposé la première analyse différentielle de fautes (DFA en anglais pour "Differential Fault Analysis") sur PRIDE que nous avons validée en pratique sur un micro-contrôleur ARM Cortex-M3. Nous avons également proposé une évaluation des contre-mesures existantes face à notre attaque, et ces travaux ont fait l'objet d'un article présenté à la conférence "CRiSIS" en 2016 [Lac et al. 2016].

Nous avons ensuite participé à la rédaction d'un article sur les différents chemins d'attaques physiques existant sur PRIDE présenté au "Lightweight Cryptography Workshop" du NIST en 2016 [Adomnicai et al. 2016], dans lequel nous avons proposé une analyse de corrélation électromagnétique (CEMA en anglais pour "Correlation ElectroMagnetic Analysis") sur ce schéma de chiffrement que nous avons également validée en pratique sur un micro-contrôleur ARM Cortex-M3.

En seconde année de thèse, nous avons généralisé la DFA que nous avons introduite sur PRIDE aux autres structures de chiffrement SPN similaires à PRIDE, notamment les LS-Designs sur lesquels l'attaque est très efficace, et nous l'avons validée en pratique sur un FPGA. Ces travaux ont aussi fait l'objet d'un article présenté à la conférence "COSADE" en 2017 [Lac et al. 2017].

Nous avons également participé à la rédaction d'un article faisant le lien entre les chiffrements légers et les chiffrements totalement homomorphes (FHE en anglais pour "Fully Homomorphic Encryption"), qui consistent à appliquer des opérations sur un message clair en ne modifiant que la valeur de son chiffré, présenté à la conférence "C&ESAR" en 2017 [Canteaut et al. 2017].

Nous avons ensuite introduit l'IRC, et nous avons proposé des tests pratiques d'attaques physiques menées en laboratoire sur des implémentations protégées et non protégées par cette contre-mesure. Ces travaux ont également fait l'objet d'un article présenté à la conférence "ISCAS" en 2018 [Lac et al. 2018].

Enfin, en troisième et dernière année de thèse, nous avons conçu GARFIELD, et nous avons analysé les performances et la sécurité de ce dernier.

Attaques par observation et contre-mesures

Chapitre 2

Bien connaître pour mieux défendre.

— **Gérard Haas**

Introduction

Une attaque par observation exploite l'existence de certaines dépendances entre les données manipulées par un circuit intégré et les grandeurs physiques observables sur ce dernier. Après avoir présenté ces dépendances et les outils statistiques disponibles pour les exploiter, nous introduisons une attaque par observation générique applicable aux chiffrements SPN que nous avons validée en laboratoire sur une cible réelle. Nous décrivons ensuite les possibilités de contre-mesures connues dans la littérature pour se prémunir de ces attaques en détaillant le cas du masquage. Ces travaux ont abouti à un article présenté au "Lightweight Cryptography Workshop" du NIST en 2016 [ADOMNICAI et al. 2016] et à un article présenté à la conférence "C&ESAR" en 2017 [CANTEAUT et al. 2017].

Sommaire

2.1 Grandeurs physiques observables **20**
2.1.1 Consommation de courant 20
2.1.2 Rayonnement électromagnétique 22
2.1.3 Température du circuit intégré 23
2.1.4 Émissions de photons 24
2.1.5 Temps d'exécution 24
2.2 Dépendances avec les données **25**
2.2.1 Traitement du signal 25
2.2.2 Analyse simple 26
2.2.3 Analyse différentielle 27
2.2.4 Analyse par modèles 30
2.3 CEMA généralisée aux chiffrements SPN **30**
2.3.1 Application en fonction de l'implémentation 31
2.3.2 Exemple de CEMA conduite en laboratoire 33
2.4 Contre-mesures **38**
2.4.1 Diminuer les fuites 38
2.4.2 Augmenter le bruit 39
2.5 Le parallélisme par tranches de bits pour faciliter le masquage . **42**
2.6 Amélioration sur une architecture 32 bits **42**

2.1 Grandeurs physiques observables

Certaines grandeurs physiques sont liées aux données manipulées ou aux opérations effectuées lors de l'exécution d'une implémentation logicielle ou matérielle d'un schéma de chiffrement sur un circuit intégré. Il peut s'agir de grandeurs physiques observables sur le circuit intégré comme la consommation de courant [KOCHER et al. 1999], les émanations électromagnétiques [GANDOLFI et al. 2001], la température [BROUCHIER et al. 2009] ou les émissions de photons [SCHLÖSSER et al. 2012] du circuit intégré, ou également de grandeurs physiques propres à l'implémentation du schéma de chiffrement comme son temps d'exécution [KOCHER 1996].

Il est très important d'étudier les informations qu'il est possible d'obtenir en observant ces différentes grandeurs physiques. En effet, la mesure d'une grandeur physique ne requiert généralement aucune modification du circuit intégré, demande un coût assez faible de mise en œuvre, et peut donc être facilement obtenue par un attaquant.

2.1.1 Consommation de courant

Les fuites d'information liées à la consommation de courant dépendent de la technologie avec laquelle le circuit intégré est fabriqué. La technologie CMOS (pour "Complementary Metal-Oxyde Semiconductor") est à la base de la plupart des circuits intégrés actuels, qu'il s'agisse d'un FPGA ou d'un ASIC.

Afin d'effectuer les opérations d'un schéma de chiffrement, un circuit intégré peut alors être vu comme un réseau de fils électriques reliés entre eux par des portes logiques élémentaires (NOT, AND, OR, XOR, ...).

Le courant électrique qui circule sur chaque fil est assimilé à une valeur binaire (un état 0 si la valeur absolue du courant est inférieure à un certain seuil qui dépend du circuit intégré, 1 sinon), et est modifié en fonction de sa valeur par chaque porte logique à l'aide de transistors à effet de champ à grille isolée (MOSFET en anglais pour "Metal Oxide Semiconductor Field Effect Transistor").

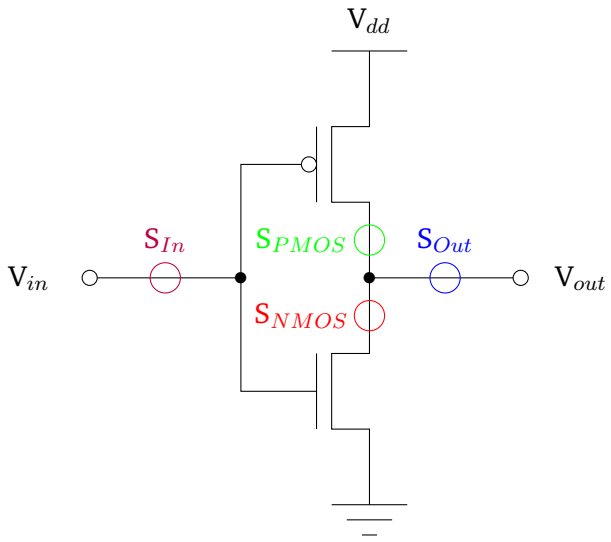
La transition d'un état 0 vers un état 1, ou d'un état 1 vers un état 0, en entrée d'une porte logique provoque un pic de tension électrique en sortie des transistors, ce qui n'est pas le cas lorsque l'état reste inchangé [GUILLEY et al. 2004].

La connaissance de la valeur de la tension électrique par sondage en sortie des transistors permet ainsi à l'attaquant de connaître les transitions effectuées en entrée de la porte logique ciblée [KOCHER et al. 1999], c'est-à-dire la distance de Hamming

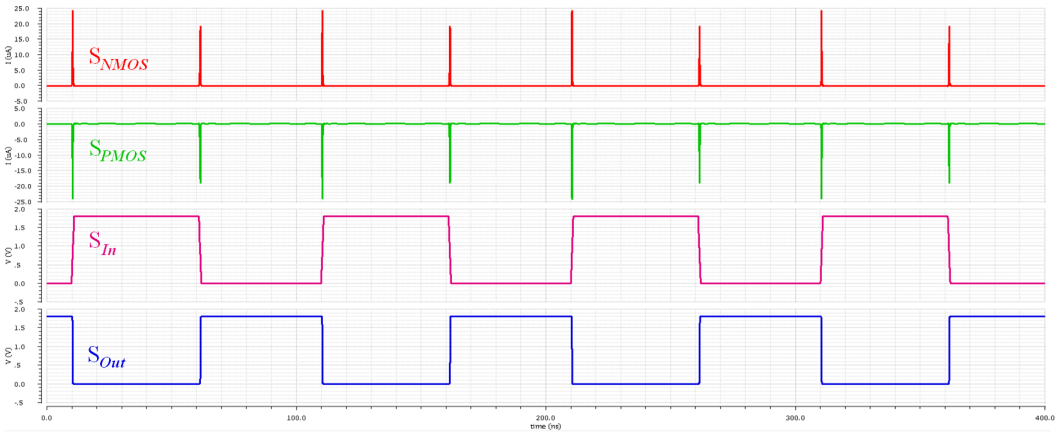
entre les valeurs des différents états. La distance de Hamming entre deux valeurs est le nombre de positions dans leurs représentations binaires où leurs bits diffèrent.

De plus, la valeur du pic de tension électrique est généralement différente en fonction de la transition effectuée, et dans ce cas l'attaquant retrouve directement la valeur des différents états en entrée de la porte logique ciblée au cours du temps.

Par exemple, la figure 2.1 décrit un inverseur CMOS, qui réalise une porte logique élémentaire NOT à l'aide d'un transistor PMOS et d'un transistor NMOS, ainsi que la valeur de la tension électrique obtenue par simulation en laboratoire avec des sondes S_{PMOS} et S_{NMOS} positionnées en sortie de chaque transistor, et des sondes S_{In} et S_{Out} positionnées en entrée et sortie de la porte logique, en faisant varier l'état en entrée de la porte logique.



(a) Inverseur CMOS.



(b) Valeur de la tension électrique obtenue par simulation.

Figure 2.1.: Inverseur CMOS **(a)** et valeur de la tension électrique obtenue par simulation avec les sondes S_{PMOS}, S_{NMOS}, S_{In} et S_{Out} **(b)**.

On observe bien sur la figure 2.1 deux pics de tension électrique différents lors d'une transition d'un état 0 vers un état 1 et d'un état 1 vers un état 0, et aucun pic de tension lorsque l'état reste inchangé.

Il est cependant très difficile de placer une sonde en sortie d'un transistor en pratique, mais la connaissance de la valeur de la consommation de courant du circuit intégré donne des informations à l'attaquant sur les transitions effectuées par le circuit intégré, c'est-à-dire la distance de Hamming entre les valeurs intermédiaires des données manipulées par le circuit intégré, puisque la consommation de courant ne varie qu'en fonction des transitions d'état en entrée des transistors.

Il existe différentes techniques pour mesurer la consommation de courant d'un circuit intégré, ou pour obtenir une valeur qui soit proportionnelle à l'intensité du courant, comme par exemple en utilisant un ampèremètre lorsque cela est possible ou en mesurant la tension entre les bornes d'une résistance insérée en série avec des lignes d'alimentation du circuit intégré [Rius et al. 2003].

2.1.2 — Rayonnement électromagnétique

Le rayonnement électromagnétique dépend de la consommation de courant du circuit intégré, et ce lien est décrit par les équations de Maxwell [Maxwell 1873]. De ce fait, le rayonnement électromagnétique ne varie qu'en fonction des transitions d'état en entrée des transistors, et donne donc des informations à l'attaquant sur la distance de Hamming entre les valeurs intermédiaires des données manipulées.

L'avantage par rapport à la consommation de courant est que la mesure des émanations électromagnétiques permet de cibler une zone plus localisée du circuit intégré, afin de pouvoir parfois se focaliser sur une seule partie des calculs lorsque plusieurs fonctions sont effectuées en parallèle, bien que l'analyse dépende également dans ce cas du positionnement de la sonde électromagnétique [Agrawal et al. 2003].

Afin de mesurer le rayonnement électromagnétique, il est possible d'utiliser une sonde électromagnétique, qui peut être une boucle [Peeters et al. 2007] ou un solénoïde [Mounier et al. 2012], placée en champ proche du circuit intégré, et reliée à un oscilloscope, souvent par le biais d'un amplificateur. Un exemple de banc d'écoute du rayonnement électromagnétique que nous avons utilisé en laboratoire est présenté à la figure 2.2, où nous avons utilisé un solénoïde, appelé antenne d'écoute, comme sonde électromagnétique relié à un oscilloscope par le biais d'un amplificateur, et placé en champ proche d'un micro-contrôleur ARM Cortex-M3, lui-même relié à un ordinateur de contrôle via un UART (pour "Universal Asynchronous Receiver Transmitter").

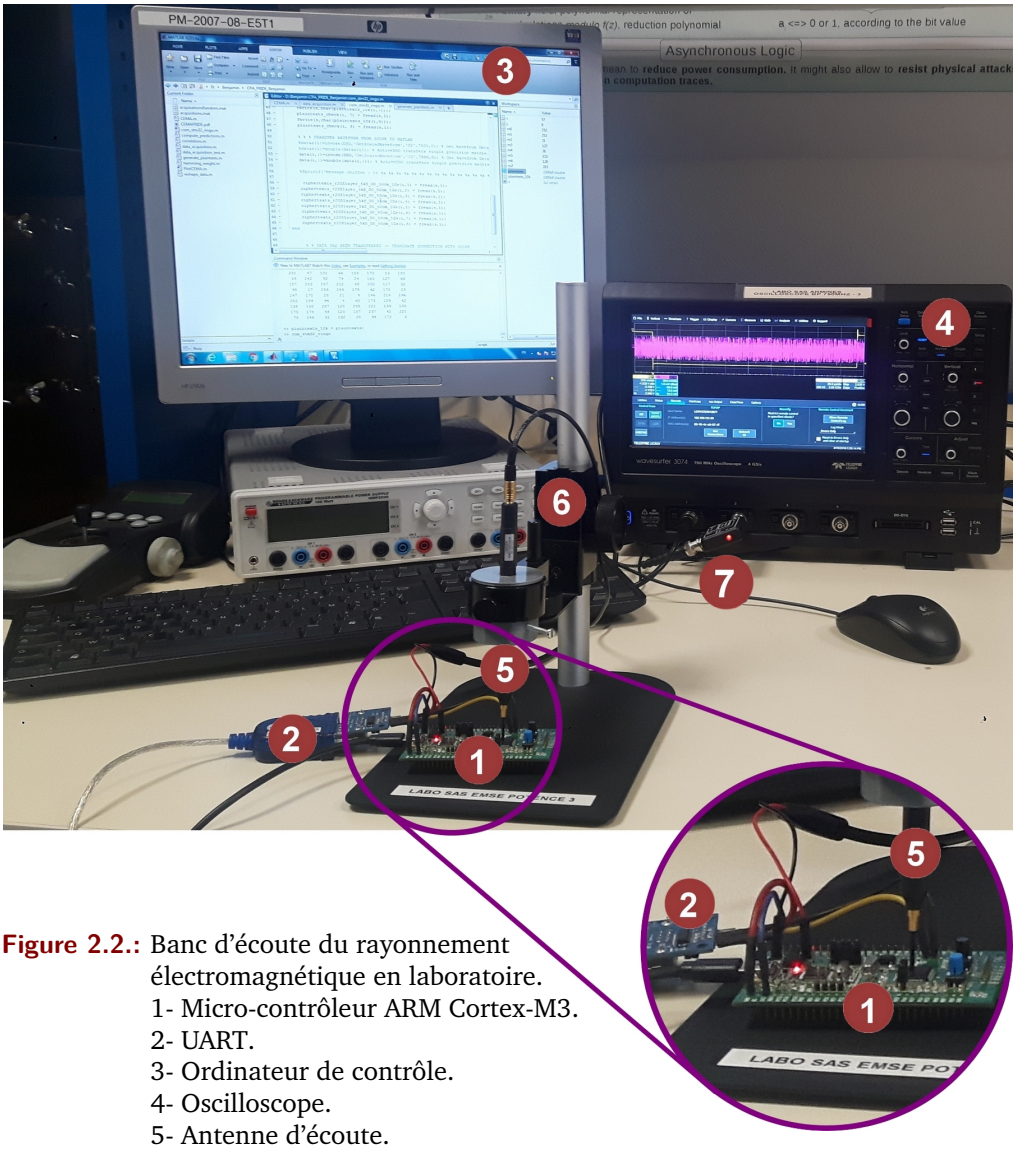


Figure 2.2.: Banc d'écoute du rayonnement électromagnétique en laboratoire.
1- Micro-contrôleur ARM Cortex-M3.
2- UART.
3- Ordinateur de contrôle.
4- Oscilloscope.
5- Antenne d'écoute.
6- Support d'antenne.
7- Amplificateur.

2.1.3 Température du circuit intégré

Un circuit intégré est généralement protégé par un boîtier qui permet de le séparer de l'environnement extérieur et dissipe en partie la puissance thermique produite par le passage du courant électrique (effet Joule [JOULE 1841]).

Bien qu'il existe certaines sources de perturbation thermique comme la température du milieu ambiant, la durée d'utilisation du circuit intégré, l'humidité, le rayonnement solaire ou la pression atmosphérique [BOUARROUDJ-BERKANI 2008], les variations de la température du circuit intégré dépendent principalement de son activité, et donc des données qu'il manipule [BROUCHIER et al. 2009].

L'analyse de la température est particulièrement efficace lorsque le circuit intégré est fabriqué à l'aide de la technologie CMOS [VIJAYKUMAR 2012]. Afin de mesurer la température, un simple capteur permet de la convertir en un courant de sortie proportionnel [HUTTER et al. 2013].

2.1.4 — Émissions de photons

Le nombre de photons émis au niveau d'un transistor en technologie CMOS dépend des transitions d'états effectuées sur ce dernier, ce qui est en particulier prononcé au niveau des transistors NMOS [SCHLÖSSER et al. 2012].

De ce fait, la quantité de photons émis par un circuit intégré dépend du poids de Hamming des valeurs intermédiaires des données qu'il manipule [CARMON et al. 2016]. Le poids de Hamming d'une valeur v est le nombre de bits de v égaux à 1 dans sa représentation binaire.

Les émissions de photons peuvent être mesurées en face arrière du circuit intégré avec un dispositif à transfert de charge (CCD en anglais pour "Charge Coupled Device") et une photodiode [SCHLÖSSER et al. 2012].

2.1.5 — Temps d'exécution

Les éléments énergétiques émis par un circuit intégré ne sont pas les seules grandeurs physiques potentiellement exploitables par un attaquant, il est en effet possible d'obtenir des informations en observant des grandeurs physiques dépendantes seulement de l'implémentation du schéma de chiffrement contenue sur le circuit intégré.

C'est le cas du temps d'exécution d'une fonction de chiffrement qui peut dépendre des données manipulées par le schéma de chiffrement, dans le cas où l'implémentation n'utilise aucune contre-mesure face à ce type de fuite d'information.

```
Entrées : C, d = (d_k, ··· , d_0)_2, n
Sortie  : M = C^d modulo n
 M ← 1
  pour i=k:0 faire
   M ← MOD(SQUARE(M),n)
   si  d_i alors
    M ← MOD(MULTIPLY(M,C),n)

 retourner M
```

Figure 2.3.: Exponentiation modulaire.

Un exemple commun est celui de l'algorithme d'exponentiation modulaire effectué par la fonction de déchiffrement du RSA, dont une implémentation est donnée à la figure 2.3, où l'exposant d est la clé privée. La fonction MOD(A,B) effectue le calcul de A modulo B, la fonction SQUARE(A) élève A au carré et la fonction MULTIPLY(A,B) multiplie A par B.

La fonction MULTIPLY est appliquée lors d'une itération seulement si le bit correspondant de l'exposant $d = (d_k, \cdots, d_0)_2$ est égal à 1. Le temps d'exécution du code va donc varier en fonction du poids de Hamming de la clé privée d.

2.2 Dépendances avec les données

Afin d'exploiter les dépendances qui existent entre les grandeurs physiques observables et les données manipulées par le circuit intégré, une attaque par observation utilise généralement différents outils statistiques de corrélation.

Il existe trois principales catégories d'attaques par observation : l'analyse simple, l'analyse différentielle et l'analyse par modèles.

Avant d'appliquer l'une de ces attaques, une connaissance de l'implémentation utilisée sur le circuit intégré est souvent nécessaire à l'attaquant afin de savoir quelle zone temporelle est susceptible de fournir une fuite d'information en fonction des opérations effectuées sur les données.

Il est également parfois nécessaire dans un premier temps d'effectuer un traitement du signal sur les données obtenues par l'observation d'une des grandeurs physiques précédemment décrites [OSWALD et al. 2012], qui vise à augmenter le rapport signal sur bruit (SNR en anglais pour "Signal-to-Noise Ratio"), le signal étant la grandeur physique observée, et le bruit étant les perturbations sur le signal.

2.2.1 Traitement du signal

Le signal obtenu par un attaquant est généralement représenté par une courbe des valeurs prises par la grandeur physique mesurée, ou par une sortie proportionnelle à la grandeur physique, en fonction du temps.

Certaines sources de perturbation dues à l'environnement extérieur peuvent provoquer des résultats erronés, et cela peut être détecté dans un premier temps visuellement si la courbe obtenue est très différente des autres courbes, ou dans un second temps en utilisant des outils statistiques vérifiant que les écarts par rapport à la moyenne ne sont pas trop importants.

Le calcul de l'écart-type [HANSON 1975] par exemple peut permettre une première identification des résultats erronés, que l'attaquant peut alors simplement supprimer. L'attaquant peut également calculer chaque courbe comme la moyenne des valeurs après un nombre fixe d'exécutions pour atténuer ce problème.

La partie la plus importante du traitement du signal est l'alignement des courbes. Il est en effet le plus souvent nécessaire d'effectuer des acquisitions à partir d'une même position temporelle sur deux exécutions qui ne sont pas effectuées en parallèle, puisque l'attaquant n'a aucun moyen de savoir quand commence l'exécution du schéma de chiffrement en pratique.

Il est donc important d'aligner les courbes afin que chaque acquisition commence à la même position temporelle. Pour cela, différentes techniques d'alignement ont été proposées comme l'alignement élastique [Van Woudenberg et al. 2011a] ou l'utilisation d'ondelettes [Muijrers et al. 2011], [Debande et al. 2012].

Certaines techniques comme SCATTER [Thiebeauld et al. 2018] utilisent une autre représentation des données d'acquisition afin d'éviter d'avoir à effectuer un alignement des courbes, qui peut être parfois un processus très long et complexe.

2.2.2 — Analyse simple

L'analyse simple consiste à déduire des informations directement depuis les données d'acquisition de la grandeur physique observée, généralement à partir de la connaissance du schéma de chiffrement exécuté sur le circuit intégré.

Ce type d'attaque peut permettre de retrouver entièrement la clé simplement en établissant un lien entre les valeurs prises par la grandeur physique observée et les valeurs possibles d'un ou plusieurs bits du secret.

Un exemple commun est alors celui d'une analyse simple de la consommation présentée à la figure 2.4 que l'on peut obtenir à partir d'une seule mesure de la consommation de courant de l'algorithme d'exponentiation modulaire décrit à la figure 2.3. La consommation de la fonction MULTIPLY n'est pas la même que celle de la fonction SQUARE.

Figure 2.4.: Analyse simple de la consommation d'une exponentiation modulaire. [Newell et al. 2013]

Ainsi, lorsque l'attaquant observe deux motifs successifs différents sur la courbe de consommation, il s'agit de la fonction SQUARE suivie de la fonction MULTIPLY, et le bit correspondant de l'exposant est égal à 1. Cette simple observation permet donc à l'attaquant de distinguer le motif de consommation de chaque fonction, et de retrouver ainsi complètement l'exposant secret en une seule acquisition.

Un autre exemple sur l'algorithme d'exponentiation modulaire décrit à la figure 2.3 est celui d'une analyse simple du temps d'exécution qui permet à l'attaquant de déduire le poids de Hamming de l'exposant secret. En effet, comme nous l'avons expliqué en section 2.1.5, le temps d'exécution du code de la figure 2.3 varie en fonction du poids de Hamming de l'exposant.

Cela ne permet pas à un attaquant de retrouver directement la clé privée de ce schéma de chiffrement, mais réduit considérablement l'espace des clés, c'est-à-dire le nombre de clés possibles à rechercher.

Une analyse simple du temps d'exécution beaucoup plus efficace a été proposée à partir du cache des processeurs contre une implémentation de l'AES qui utilise un tableau pour implémenter la S-box [Bernstein 2005], [Osvik et al. 2006]. L'attaque repose sur les temps mis pour accéder au tableau, qui dépendent du fait que la valeur en entrée est déjà en cache ou pas, c'est-à-dire si elle a déjà été utilisée.

L'analyse simple peut également permettre de faire de l'ingénierie inverse qui consiste à identifier toutes les étapes d'un schéma de chiffrement, et à les situer dans l'exécution temporellement [Eisenbarth et al. 2010].

2.2.3 — Analyse différentielle

Une analyse différentielle compare les données d'acquisition de la grandeur physique observée avec des données de prédiction d'une valeur intermédiaire en utilisant différents messages clairs en entrée [Kocher et al. 1999].

Le principe derrière une analyse différentielle est d'appliquer la stratégie de "diviser pour mieux régner" [Brassard et al. 1996], qui consiste à réduire un problème trop difficile à résoudre en plusieurs sous-problèmes résolubles en un temps raisonnable. Dans le cas d'une analyse différentielle, l'objectif est de chercher à retrouver des parties de la clé indépendamment les unes des autres.

L'idéal pour un attaquant serait donc de mener l'analyse sur chaque bit de la clé indépendamment les uns des autres, mais cela n'est pas toujours possible et dépend principalement des mots manipulés par l'implémentation ciblée. De ce fait, l'analyse s'effectue généralement octet par octet puisque les implémentation 8 bits sont les plus utilisées dans le contexte de l'IoT.

On peut classer les analyses différentielles en deux catégories : les analyses différentielles de premier ordre et les analyses différentielles d'ordre supérieur.

2.2.3.1 – Analyse différentielle de premier ordre

Dans un premier temps, l'attaquant exécute le schéma de chiffrement sur un ensemble $\mathcal{E}$ de messages clairs, et réalise l'acquisition d'une grandeur physique sur le circuit intégré ciblé pour chaque exécution. Il obtient ainsi des courbes d'acquisition où chaque courbe correspond à une exécution.

L'attaquant cible ensuite une valeur intermédiaire de l'exécution qui peut s'exprimer avec pour seule inconnue une partie de la clé, c'est-à-dire qui doit être fonction d'une partie du message clair ou du chiffré et d'une partie de la clé. Il calcule alors la valeur intermédiaire pour chaque message clair de $\mathcal{E}$ avec un ensemble $\mathcal{K}$ de clés qui varient seulement sur la partie dont dépend la valeur intermédiaire ciblée.

Il est normalement possible d'effectuer les calculs nécessaires pour obtenir la valeur intermédiaire puisqu'elle peut s'exprimer avec pour seule inconnue la partie de la clé, et que comme nous l'avons vu, d'après le "principe de Kerckhoffs", le schéma de chiffrement utilisé doit être public.

Il stocke alors des données liées à la valeur intermédiaire pour chaque clé de $\mathcal{K}$, appelées données de prédiction [PEETERS et al. 2007]. Il peut s'agir par exemple du poids de Hamming de la valeur intermédiaire ou de la distance de Hamming avec la valeur intermédiaire précédente dans le cas où la grandeur physique observée est la consommation ou le rayonnement électromagnétique.

Finalement, l'attaquant compare pour chaque clé de $\mathcal{K}$ les données de prédiction avec les courbes d'acquisition en utilisant un distingueur, et classe les clés en fonction des résultats obtenus avec un algorithme d'énumération des clés [VEYRAT-CHARVILLON et al. 2013]. Le distingueur permet à l'attaquant d'extraire la valeur la plus probable de la clé sur la partie dont dépend la valeur intermédiaire.

De cette façon, il peut retrouver complètement la clé en réitérant l'attaque sur d'autres valeurs intermédiaires.

2.2.3.2 – Analyse différentielle d'ordre supérieur

Une analyse différentielle d'ordre d consiste à appliquer les mêmes étapes que précédemment mais en ciblant d valeurs intermédiaires différentes qui peuvent toutes s'exprimer avec pour seule inconnue une partie de la clé.

Plus précisément, après s'être procuré des courbes d'acquisition à partir de l'exécution du schéma de chiffrement sur un ensemble de messages clairs, et après avoir

calculé des données de prédiction liées à chaque valeur intermédiaire pour chaque hypothèse de clé sur la partie ciblée de la clé, l'attaquant utilise là encore un distingueur pour comparer les données de prédiction avec les courbes d'acquisition pour chaque hypothèse de clé, et classe les clés en fonction des résultats obtenus avec un algorithme d'énumération des clés.

2.2.3.3 – Distingueurs classiques

La première possibilité est d'utiliser un outil statistique de corrélation comme la différence entre les moyennes des courbes obtenues, qui a conduit à la première analyse différentielle de consommation (DPA en anglais pour "Differential Power Analysis") [KOCHER et al. 1999], ou le coefficient de corrélation de Pearson, qui a introduit l'analyse de consommation par corrélation (CPA en anglais pour "Correlation Power Analysis") [BRIER et al. 2004] et qui se calcule en utilisant la formule suivante appliquée entre les prédictions et les courbes d'acquisition :

$$\rho(X,Y) = \frac{\mathsf{E}(XY) - \mathsf{E}(X)\mathsf{E}(Y)}{\sqrt{\mathsf{E}\Big((X-\mathsf{E}(X))^2\Big)\mathsf{E}\Big((Y-\mathsf{E}(Y))^2\Big)}}$$

où $\mathsf{E}(X)$ est l'espérance mathématique de la variable aléatoire X.

Le coefficient de corrélation de Pearson permet d'éliminer certains mauvais pics de corrélation causés par des perturbations que l'on peut parfois constater sur d'autres points des courbes d'acquisition que les valeurs intermédiaires ciblées avec d'autres outils statistiques de corrélation.

Une autre possibilité est d'utiliser l'information mutuelle, qui a conduit à l'analyse de l'information mutuelle (MIA en anglais pour "Mutual Information Analysis") [GIERLICHS et al. 2008] et qui se calcule en utilisant la formule suivante appliquée entre les prédictions et les courbes d'acquisition :

$$I(X,Y) = \frac{1}{2}\log\left(\frac{\det(P(X))}{\det(P(X/Y))}\right)$$

où $P(X)$ est la matrice de covariance de la variable aléatoire X.

Certains distingueurs, comme le coefficient de corrélation de Pearson, ne s'appliquent qu'entre deux variables. Afin d'utiliser un tel distingueur pour une analyse différentielle d'ordre supérieur, il est possible d'appliquer une fonction de combinaison entre les prédictions des différentes valeurs intermédiaires. On peut notamment citer la combinaison par produits [CHARI et al. 1999] qui consiste à multiplier les différentes prédictions entre elles, ou la combinaison par différence absolue [MESSERGES 2000] qui calcule la différence absolue des prédictions.

D'autres distingueurs, comme l'information mutuelle, peuvent quant à eux s'appliquer sur plusieurs variables, et donc directement sur les prédictions des différentes valeurs intermédiaires [GIERLICHS et al. 2010].

2.2.4 Analyse par modèles

Afin de réaliser une analyse par modèles ("template analysis" en anglais), l'attaquant doit être en mesure d'utiliser un dispositif identique au circuit intégré ciblé, mais sur lequel il a la capacité de modifier la valeur de la clé [CHARI et al. 2003].

Il réalise alors un ensemble de mesures d'une ou plusieurs grandeurs physiques sur le circuit intégré ciblé, et effectue ensuite les mêmes mesures sur l'autre dispositif, pas nécessairement avec les mêmes messages [RECHBERGER et al. 2004], en faisant varier seulement une partie de la clé.

Il cible généralement une partie de la clé qui est manipulée par le schéma de chiffrement, comme un octet par exemple. L'ensemble des mesures associées à chaque clé est appelé un modèle ("template" en anglais).

L'attaquant compare finalement les modèles obtenus avec les mesures effectuées sur le circuit intégré ciblé. Le modèle le plus proche lui indique la valeur correcte de la clé sur la partie ciblée. De cette façon, il peut une fois de plus retrouver entièrement la clé en réitérant l'attaque sur d'autres parties de cette dernière.

2.3 CEMA généralisée aux chiffrements SPN

L'analyse du rayonnement électromagnétique par corrélation (CEMA en anglais pour "Correlation ElectroMagnetic Analysis") est une analyse différentielle qui consiste à appliquer le coefficient de corrélation de Pearson entre les données de prédictions et la mesure des émanations électromagnétiques.

La mesure des émanations électromagnétiques requiert un coût relativement bas de mise en œuvre, et permet dans de nombreux cas de ne pas avoir à modifier le circuit intégré en positionnant la sonde au dessus d'un rail d'alimentation afin d'obtenir le champ électromagnétique associé à la consommation de courant globale.

Le coefficient de corrélation de Pearson est quant à lui très facile à appliquer, et permet généralement de retrouver la clé à partir de peu de courbes lors d'une CEMA sur une implémentation non protégée [DING et al. 2009] (quelques milliers de courbes sont la plupart du temps largement suffisantes dans ce cas).

Afin d'appliquer une CEMA sur un chiffrement SPN, un attaquant peut cibler la première ou la dernière couche de substitution. En effet, la première (resp. la dernière) couche de substitution est appliquée directement après (resp. avant) l'addition entre une valeur connue de l'attaquant, dérivée du message clair (resp. du chiffré), et la première (resp. dernière) clé de tour.

Ainsi, l'entrée (resp. la sortie) de la première (resp. la dernière) couche de substitution peut s'exprimer avec pour seule inconnue la première (resp. la dernière) clé de tour. L'attaquant est alors capable de retrouver la première (resp. la dernière) clé de tour, et réitère l'attaquant autant de fois que nécessaire pour retrouver entièrement la clé secrète (généralement deux clés de tour suffisent).

2.3.1 Application en fonction de l'implémentation

Il faut distinguer deux méthodes d'implémentation de la couche de substitution utilisée par les chiffrements SPN : l'utilisation de tableaux afin de mettre à jour directement les valeurs en entrées des S-boxes ou l'application d'une technique appelée parallélisme par tranches de bits ("bit-slicing" en anglais).

2.3.1.1 Implémentation à partir de tableaux

La plupart des chiffrements SPN appliquent une ou plusieurs S-boxes en parallèle sur le bloc de données pour effectuer la couche de substitution, dont l'implémentation s'effectue dans ce cas à partir de tableaux.

On retrouve de nombreuses attaques par observation sur des implémentations de chiffrements SPN à partir de tableaux [Osvik et al. 2006], [Masoomi et al. 2010], [Zhang et al. 2010], [Biryukov et al. 2016] ou [Lo et al. 2017].

Qu'il s'agisse d'une implémentation logicielle ou matérielle, l'utilisation de tableaux pour appliquer la couche de substitution traite les entrées des S-boxes indépendamment les unes des autres, ce qui permet à l'attaquant d'appliquer la stratégie de "diviser pour mieux régner". Ainsi, la valeur du rayonnement électromagnétique lors de l'application d'une S-box dépend uniquement de la valeur mise à jour.

Afin de mener une CEMA, l'attaquant effectue des prédictions sur les valeurs possibles en sortie (resp. en entrée) de chaque S-box, et fait des acquisitions du rayonnement électromagnétique de la première (resp. la dernière) couche de substitution. La corrélation la plus importante entre les prédictions et les acquisitions obtenues lui permet alors de retrouver la valeur de la clé secrète utilisée lors du premier (resp. dernier) tour du chiffrement.

2.3.1.2 – Parallélisme par tranches de bits

Le parallélisme par tranches de bits consiste à transformer une fonction appliquée n fois sur des mots de m bits $M_1, \cdots, M_n$ en une autre fonction produisant le même résultat, mais appliquée sur m mots de n bits $N_1, \cdots, N_m$. Pour cela, le bit $1 \leq j \leq m$ du mot M_i correspond au bit i du mot N_j pour tout $1 \leq i \leq n$.

L'utilisation de cette technique permet d'exploiter pleinement les registres d'une architecture 8, 16, 32 ou 64 bits, mais demande un surcoût souvent important à cause de la transformation booléenne nécessaire pour obtenir uniquement des opérations bit à bit [PORNIN 2001], et impose de prendre n fois plus de mots en entrée et donc de stocker n fois plus de données.

Certains chiffrements SPN, comme PRIDE [ALBRECHT et al. 2014b], les LS-Designs [GROSSO et al. 2015] ou les structures CUBE [BERGER et al. 2015], utilisent une structure particulière afin d'appliquer efficacement le parallélisme par tranches de bits uniquement sur la couche de substitution. Ils appliquent pour cela une même S-box de m bits n fois en parallèle (où n dépend de la taille du bloc), mais le bit $1 \leq j \leq m$ de la S-box en position i correspond au bit i du mot de n bits en position j en entrée de la couche de substitution pour tout $1 \leq i \leq n$.

La figure 2.5 illustre cette structure pour appliquer efficacement le parallélisme par tranches de bits sur un bloc de 64 bits avec une même S-box $\mathcal{S}$ de 4 bits.

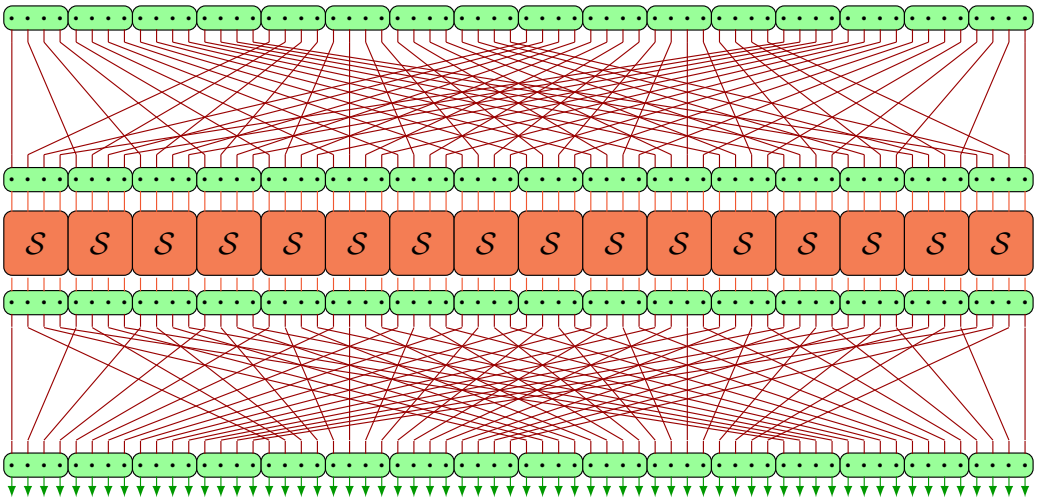


Figure 2.5.: Structure pour appliquer efficacement le parallélisme par tranches de bits sur un bloc de 64 bits avec une même S-box $\mathcal{S}$ de 4 bits.

L'implémentation utilise alors la forme algébrique normale de chaque coordonnée de la S-box, c'est-à-dire que chaque bit en sortie est exprimé comme une fonction booléenne des bits en entrée, ce qui permet d'appliquer la couche de substitution en effectuant seulement des opérations bit à bit entre les mots de n bits en entrée.

À la différence d'une implémentation à partir de tableaux, pour laquelle la taille des mots manipulés par la couche de substitution est fixe et dépend seulement des tableaux utilisés, le parallélisme par tranches de bits manipule des mots dont la taille dépend principalement de l'architecture utilisée, ce qui rend une attaque par observation plus difficile à réaliser.

Afin d'appliquer une CEMA sur une couche de substitution implémentée avec le parallélisme par tranches de bits, nous avons proposé une nouvelle attaque que nous avons validée expérimentalement sur une cible réelle en laboratoire [ADOMNICAI et al. 2016], en collaboration avec A. Adomnicai, un autre doctorant du laboratoire. L'attaque consiste cette fois à effectuer des prédictions sur la valeur de chaque mot manipulé par le parallélisme par tranches de bits en entrée (resp. en sortie) de la première (resp. la dernière) couche de substitution.

La corrélation la plus importante entre les prédictions et les acquisitions du rayonnement électromagnétique de la couche de substitution permet à l'attaquant de retrouver la valeur de la clé utilisée lors du premier (resp. dernier) tour.

Pour donner un exemple d'une CEMA sur ce type d'implémentation, nous avons mené une expérimentation de l'attaque en laboratoire sur le schéma de chiffrement PRIDE [ALBRECHT et al. 2014b], dont la structure permet d'utiliser efficacement le parallélisme par tranches de bits sur la couche de substitution.

2.3.2 Exemple de CEMA conduite en laboratoire

Afin de tester la faisabilité de l'attaque sur PRIDE, nous avons implémenté et exécuté ce dernier sur un micro-contrôleur ARM Cortex-M3 car il s'agit d'une architecture 32 bits assez représentative des processeurs embarqués dans les dispositifs IoT. Avant de décrire les manipulations que nous avons effectuées et les résultats que nous avons obtenus, nous allons brièvement donner les spécifications de PRIDE.

2.3.2.1 Le schéma de chiffrement PRIDE

PRIDE est une preuve de concept de la notion de structure entrelacée. Il s'agit d'une structure SPN qui consiste à appliquer des matrices sur des mots de taille fixe du bloc de données lors de l'étage linéaire, et à utiliser la structure décrite à la figure 2.5 pour effectuer la couche de substitution.

Ce type de structure permet d'obtenir des propriétés cryptographiques intéressantes, notamment sur la diffusion de l'étage linéaire, tout en proposant le plus souvent des implémentations très compactes.

PRIDE est composé de 20 tours, prend en entrée un bloc de 64 bits et utilise une clé secrète de 128 bits $K = K_0||K_1$ où $||$ désigne la concaténation. Les 64 premiers bits K_0 sont utilisés pour effectuer un blanchiment de message (“key whitening” en anglais), qui consiste à appliquer une clé avant et après la fonction de chiffrement, et les 64 derniers bits K_1 sont manipulés par un algorithme de cadencement de clés afin de produire les clés de tours $f_r(K_1)$ pour $1 \leq r \leq 20$ en ajoutant simplement des constantes aux octets d'indice impair de K_1.

Soit $(K_1)_i$ l'octet $0 \leq i \leq 7$ de K_1 alors $f_r(K_1)$ est égal à

$$(K_1)_0||g_r^{(193)}((K_1)_1)||(K_1)_2||g_r^{(165)}((K_1)_3)||(K_1)_4||g_r^{(81)}((K_1)_5)||(K_1)_6||g_r^{(197)}((K_1)_7)$$

avec $g_r^{(n)}(x) = (x + nr)$ modulo 256.

PRIDE utilise une S-box $\mathcal{S}$, une permutation des bits $\mathcal{P}$ et des matrices $\mathcal{L}_0$, $\mathcal{L}_1$, $\mathcal{L}_2$ et $\mathcal{L}_3$ données dans [ALBRECHT et al. 2014b]. Le tour d'indice r pour $1 \leq r \leq 19$, noté $\mathcal{R}$, est alors composé des étapes suivantes :

- Appliquer la permutation inverse $\mathcal{P}^{-1}$ à $f_r(K_1)$ et appliquer un XOR entre le résultat obtenu et l'entrée du tour r,
- Appliquer la S-box $\mathcal{S}$ à chaque quartet du bloc de données,
- Appliquer la permutation $\mathcal{P}$ au bloc de données,
- Multiplier le mot $0 \leq i \leq 3$ de 16 bits du bloc de données par la matrice $\mathcal{L}_i$,
- Appliquer la permutation inverse $\mathcal{P}^{-1}$ pour obtenir la sortie du tour r.

Pour le dernier tour, noté $\mathcal{R}'$, seules les deux premières étapes sont appliquées. La figure 2.6 décrit un tour de PRIDE.

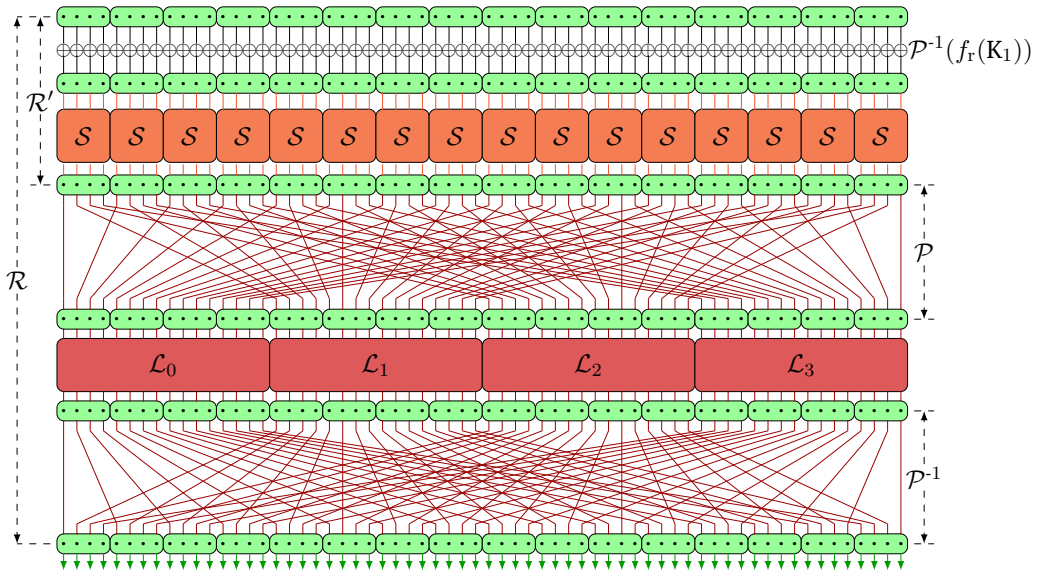


Figure 2.6.: Un tour de PRIDE.

Afin de chiffrer un message clair M, la fonction de chiffrement applique $\mathcal{P}^{-1}$ à M. Ensuite, elle effectue un XOR entre le résultat et K_0. Elle applique alors les 20 tours que nous venons de décrire et effectue une fois de plus un XOR avec K_0. Enfin, $\mathcal{P}$ est appliquée au résultat pour obtenir le message chiffré C. La figure 2.7 illustre la structure générale de PRIDE.

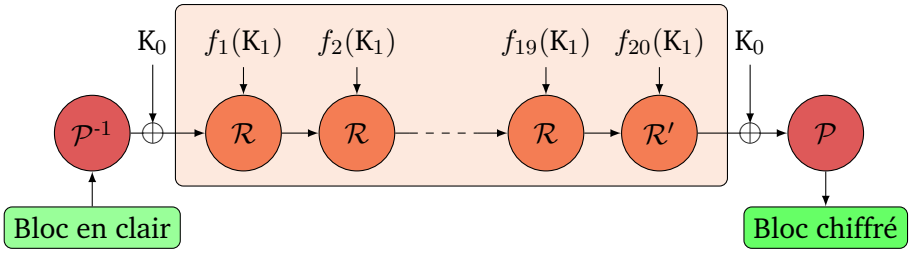


Figure 2.7.: Structure de PRIDE.

Dit autrement, les données peuvent être représentées par une matrice binaire de w lignes et c colonnes, et la couche de substitution applique alors une même S-box à chaque colonne, tandis que l'étage linéaire multiplie chaque ligne par une matrice binaire de c lignes et c colonnes.

Il est à noter qu'une analyse différentielle sur 18 tours de PRIDE a récemment été proposée [Lallemand et al. 2017]. Ce type d'attaque mathématique appliquée à un nombre de tours aussi proche que celui spécifié par les concepteurs (20) montre que la marge de sécurité fournie par PRIDE est très faible. Cependant, PRIDE a une structure très intéressante pour la cryptographie légère, et étudier les possibilités d'attaques physiques sur ce type de schéma de chiffrement est d'un grand intérêt.

2.3.2.2 – CEMA conduite en laboratoire sur PRIDE

Afin de mener une CEMA sur PRIDE, nous avons utilisé l'implémentation 8 bits de référence de PRIDE, dont le code source est donné en annexe A.4.3.

La première étape consiste à retrouver $\mathcal{P}(K_0)$, qui est appliquée directement après la dernière couche de substitution dans le cas de l'implémentation de référence qui utilise le parallélisme par tranches de bits pour implémenter la couche de substitution. Il suffit donc de cibler la dernière couche de substitution pour les acquisitions, et d'effectuer des prédictions sur les octets de l'addition entre le chiffré et $\mathcal{P}(K_0)$.

Pour cela, nous avons utilisé le banc d'écoute du rayonnement électromagnétique décrit à la figure 2.2. Nous avons effectué dans un premier temps une analyse des rayonnements électromagnétiques de PRIDE afin de situer temporellement la dernière couche de substitution.

Il s'agit d'une étape d'ingénierie inverse qui consiste à identifier toutes les étapes du schéma de chiffrement, et à les situer temporellement, comme nous l'avions mentionné en section 2.2.2.

La figure 2.8 donne les courbes que nous avons obtenues, et qui nous ont bien permis d'identifier toutes les étapes de PRIDE.

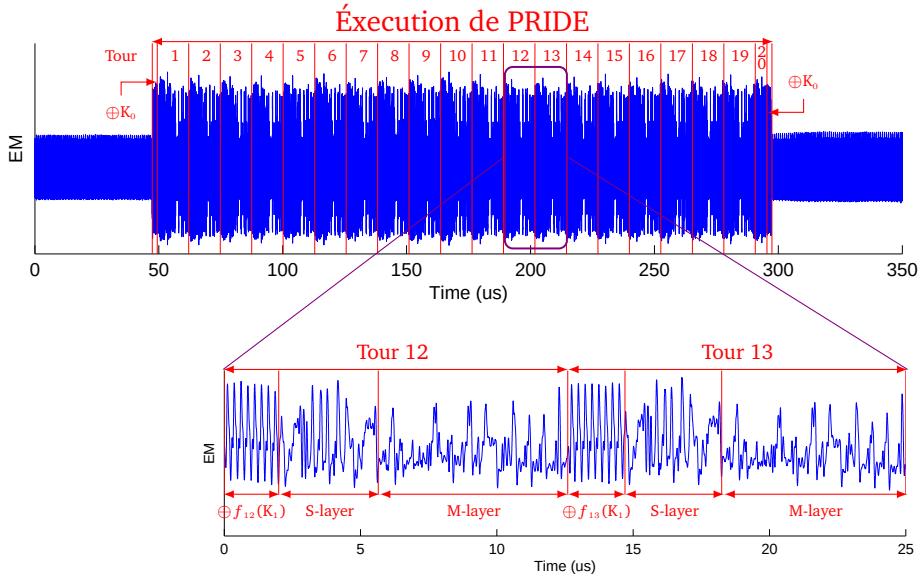


Figure 2.8.: Analyse des rayonnements électromagnétiques de PRIDE.

Ensuite, nous avons exécuté PRIDE sur 1000 messages clairs aléatoires avec la clé secrète 0xa371b246f90cf582e417d148e239ca5d. Nous avons ciblé la dernière couche de substitution pour effectuer des acquisitions, et nous avons obtenu des traces avec 7500 points par exécution. Notons T la matrice des traces obtenues :

$$\mathrm{T} = \begin{bmatrix} \mathrm{T}_0 \\ \mathrm{T}_2 \\ \vdots \\ \mathrm{T}_{7499} \end{bmatrix} = \begin{bmatrix} \mathrm{t}_{0,0} & \mathrm{t}_{1,1} & \cdots & \mathrm{t}_{0,999} \\ \mathrm{t}_{2,0} & \mathrm{t}_{2,1} & \cdots & \mathrm{t}_{2,999} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{t}_{7499,0} & \mathrm{t}_{7499,1} & \cdots & \mathrm{t}_{7499,999} \end{bmatrix}.$$

Afin de retrouver l'octet $0 \leq i \leq 7$ de $\mathcal{P}(\mathrm{K}_0)$ nous avons d'abord calculé le poids de Hamming wt du XOR entre l'octet i de chaque chiffré, noté $(\mathrm{C}_j)_i$ pour $0 \leq j \leq 999$, et chaque hypothèse de clé $0 \leq \mathrm{H_K} \leq 255$. Notons E^i les matrices d'estimation obtenues :

$$\mathrm{E}^i = \begin{bmatrix} \mathrm{E}^i_0 \\ \mathrm{E}^i_1 \\ \vdots \\ \mathrm{E}^i_{255} \end{bmatrix} = \begin{bmatrix} \mathrm{e}^i_{0,0} & \mathrm{e}^i_{0,1} & \cdots & \mathrm{e}^i_{0,999} \\ \mathrm{e}^i_{1,0} & \mathrm{e}^i_{1,1} & \cdots & \mathrm{e}^i_{1,999} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{e}^i_{255,0} & \mathrm{e}^i_{255,1} & \cdots & \mathrm{e}^i_{255,999} \end{bmatrix}$$

avec $e^i_{H_K,j} = wt((C_j)_i \oplus H_K)$. Nous avons enfin calculé les matrices des coefficients de corrélation P^i, pour $0 \leq i \leq 7$, à partir du coefficient de corrélation de Pearson noté ρ entre E^i et T :

$$P^i = \begin{bmatrix} P^i_0 \\ P^i_1 \\ \vdots \\ P^i_{7499} \end{bmatrix} = \begin{bmatrix} \rho^i_{0,0} & \rho^i_{0,1} & \cdots & \rho^i_{0,255} \\ \rho^i_{1,0} & \rho^i_{1,1} & \cdots & \rho^i_{1,255} \\ \vdots & \vdots & \ddots & \vdots \\ \rho^i_{7499,0} & \rho^i_{7499,1} & \cdots & \rho^i_{7499,255} \end{bmatrix}$$

avec $\rho^i_{t,H_K} = \rho(T_t, E^i_{H_K})$. La figure 2.9 donne les courbes obtenues en traçant P^0 sur l'intervalle 1100 à 2100 points en abscisse.

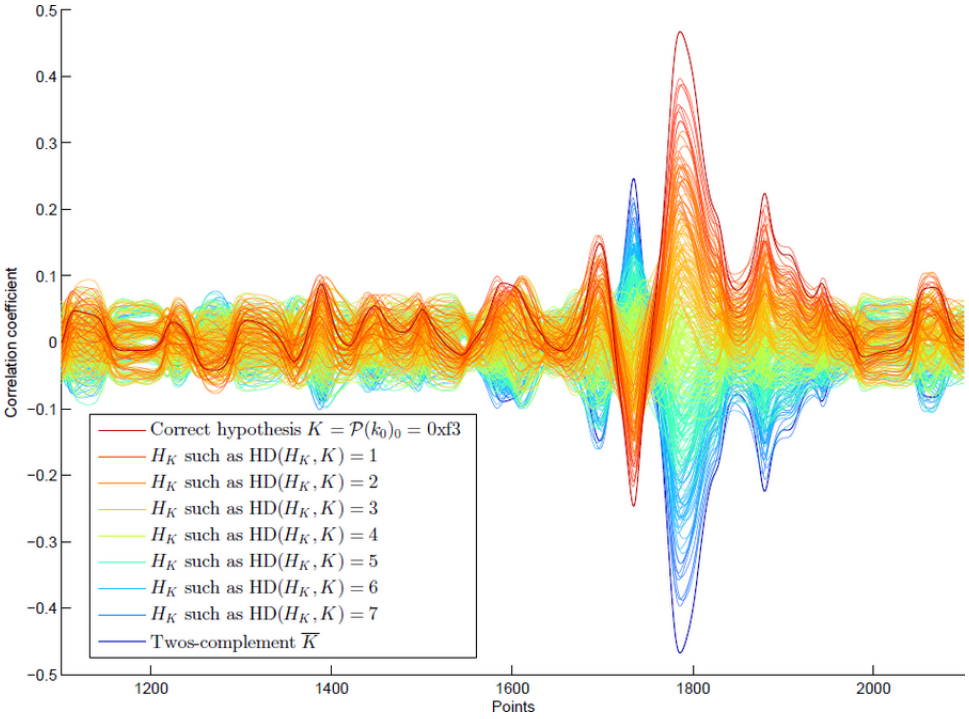


Figure 2.9.: Courbes de corrélation obtenues pour $\mathcal{P}(K_0)_0$.

Nous avons pu ainsi clairement distinguer la valeur de corrélation la plus élevée pour le premier octet de $\mathcal{P}(K_0)$ qui était 0xf3. En répétant les étapes précédentes sur les octets suivants de $\mathcal{P}(K_0)$, nous avons retrouvé la valeur correcte de $\mathcal{P}(K_0)$ = 0xf3f721cb1c882658 à partir de tous les P^i.

Après avoir retrouvé entièrement $\mathcal{P}(K_0)$, nous avons réitéré l'attaque sur le premier tour afin de retrouver $f_1(K_1)$ qui est directement ajouté par un XOR au message clair après $\mathcal{P}(K_0)$. Pour cela, nous avons ciblé la première couche de substitution pour les acquisitions, et nous avons calculé pour chaque octet le poids de Hamming du XOR entre chaque message clair, $\mathcal{P}(K_0)$ et chaque hypothèse de clé pour les prédictions.

Nous avons finalement proposé une implémentation 32 bits de PRIDE, donnée en annexe A.4.1, afin d'exploiter les bénéfices de travailler sur des mots de 32 bits.

Appliquer une CEMA sur ce type d'implémentation est plus complexe, car cela demande d'effectuer des prédictions sur un plus grand nombre de bits, du fait que chaque opération opère sur des mots de 32 bits. L'attaque reste cependant possible, et bien moins coûteuse qu'une recherche exhaustive.

2.4 Contre-mesures

Les chiffrements SPN ne sont pas les seuls à être vulnérables aux attaques par observation, puisque l'on retrouve ce type d'attaque sur la plupart des structures existantes aujourd'hui, que ce soit sur les réseaux de Feistel [Bai et al. 2009], sur les structures ARX (pour "Addition-Rotation-XOR") [Adomnicai et al. 2017], sur les chiffrements à flot [Jia et al. 2012] ou sur les schémas de chiffrement à clé publique [Kocher 1996]. Une structure ARX consiste, comme son nom l'indique, à n'utiliser que l'addition, la rotation et le XOR sur les données.

Il est donc primordial de proposer des contre-mesures efficaces pour se prémunir de ces attaques. Pour cela, les méthodes proposées visent à réduire le SNR, afin que les perturbations sur le signal soient trop importantes pour qu'un attaquant exploite la grandeur physique observable. Il existe ainsi deux principales catégories de contre-mesures pour se prémunir des attaques par observation : les premières visant à diminuer les fuites et les secondes à augmenter le bruit.

2.4.1 Diminuer les fuites

Afin de diminuer les fuites d'information liées aux grandeurs physiques observables, deux principales techniques peuvent être utilisées.

La première consiste à équilibrer la consommation de courant afin de réduire sa corrélation avec les données manipulées par le circuit intégré. Pour cela, il est possible d'utiliser des portes logiques double-rail à pré-charge [Tiri et al. 2004], [Guilley et al. 2008] ou [Danger et al. 2009], comme les portes logiques WDDL (pour "Wave Dynamic Differential Logic") [Fang et al. 2015] en technologie CMOS.

Le circuit intégré est dans ce cas constitué de deux fois plus de fils électriques : un fil sur deux a une valeur d'état associée à une donnée, et l'autre a la valeur opposée afin que le poids de Hamming des données soit constant.

Les portes logiques double-rail à pré-charge sont alors composées de deux étapes : une phase de pré-charge où les deux états sur les deux fils associés sont fixés à une valeur définie (0 ou 1), et une phase d'évaluation qui applique toujours une seule transition sur un des deux états en entrée de la porte logique afin que le nombre de transitions soit toujours constant en entrée de chaque porte logique.

La seconde technique vise à équilibrer le traitement des calculs cryptographiques afin que le temps d'exécution ne varie pas, et cela quelles que soient les données en entrée [JAFFE et al. 2003]. Il est à noter également que les mêmes opérations doivent être effectuées par le schéma de chiffrement quelle que soit la valeur de la clé afin de se prémunir des analyses simples.

Par exemple, afin d'apporter une première mesure de sécurité contre les attaques par observation sur l'algorithme d'exponentiation modulaire, que nous avons déjà mentionné et dont une implémentation non sécurisée est donnée à la figure 2.3, il est possible d'appliquer la fonction MULTIPLY quelle que soit la valeur du bit traité de l'exposant secret d. La figure 2.10 donne une telle implémentation de l'algorithme d'exponentiation modulaire.

```
Entrées : C, d = (d_k, ··· , d_0)_2, n
Sortie  : M_0 =C^d modulo n
 M_0 ← 1
 M_1 ← 1
  pour i=k:0 faire
    M_0 ← MOD(SQUARE(M_0),n)
    M_1 ← MOD(MULTIPLY(M_0,C),n)
    M_0 ← M_{d_i}
 retourner M_0
```

Figure 2.10.: Exponentiation modulaire sécurisée face aux analyses simples.

Le temps d'exécution ne dépend plus de la valeur de d, et une analyse simple de la consommation ou des rayonnements électromagnétiques ne permet pas à un attaquant de retrouver d puisque les mêmes motifs se répètent à chaque itération.

2.4.2 Augmenter le bruit

Afin d'augmenter le bruit, deux principales techniques peuvent également être utilisées. La première technique, proposée pour améliorer la sécurité contre les attaques par observation dès l'apparition de la DPA [KOCHER et al. 1999], consiste à rendre aléatoire la position temporelle de chaque opération du schéma de chiffrement.

Pour cela, il est possible d'ajouter des délais aléatoires dans l'exécution [CLAVIER et al. 2000], [TUNSTALL et al. 2007] ou de rendre aléatoire l'ordre dans lequel certaines opérations indépendantes les unes des autres, comme l'application de plusieurs S-boxes en parallèle, sont effectuées ("operation shuffling" en anglais) [HERBST et al. 2006], [RIVAIN et al. 2009], [LUO et al. 2015].

De cette façon, le traitement du signal est plus difficile à réaliser pour l'attaquant, notamment la partie sur l'alignement des courbes [VEYRAT-CHARVILLON et al. 2012].

Cette technique n'ajoute aucun bruit directement sur le signal, mais permet de le modifier à chaque exécution afin d'en complexifier son exploitation.

La seconde technique est le masquage [MESSERGES 2001], qui consiste à appliquer un ou plusieurs masques aléatoires sur les données, où la valeur de chaque masque doit être différente à chaque exécution et le plus souvent secrète, soit avec l'opération XOR et on parle dans ce cas de masques logiques, soit avec l'opération d'addition sur plusieurs bits et on parle alors de masques arithmétiques.

Ensuite, la valeur correcte de chaque masque doit être correctement calculée au cours de l'exécution, indépendamment des autres masques, afin d'être en mesure de les retirer correctement du chiffré à la fin de la zone à protéger.

Ainsi, les grandeurs physiques observables dépendent d'un côté des données, mais d'un autre également de la valeur des masques utilisés lors de chaque exécution. Le masquage permet donc une amplification du bruit sur le signal.

L'analyse différentielle reste néanmoins faisable, mais nécessite de cibler plusieurs valeurs intermédiaires, et demande un nombre de courbes qui augmente considérablement en fonction du nombre de masques utilisés [RIVAIN et al. 2009].

Généralement, utiliser d masques permet de se prémunir d'une analyse différentielle d'ordre d, et on parle dans ce cas d'un masquage d'ordre d [SCHRAMM et al. 2006], [RIVAIN et al. 2010], [JOURNAULT et al. 2017b].

La valeur d'un masque logique (resp. arithmétique) peut être facilement calculée sur la partie linéaire par rapport au XOR (resp. à l'addition sur plusieurs bits) en appliquant simplement les mêmes opérations sur le masque. Par contre, ce n'est pas le cas sur la partie non-linéaire, pour laquelle les opérations doivent être modifiées afin de calculer correctement la valeur du masque.

La plupart des méthodes actuelles de masquage reposent sur le schéma introduit en 2003 par Y. Ishai, A. Sahai et D. Wagner et désigné par ISW (pour "Ishai-Sahai-Wagner") [ISHAI et al. 2003], qui permet de mettre à jour la valeur d'un masque logique après les opérations non-linéaires AND/OR effectuées bit à bit.

Le principe du schéma ISW est le suivant : soit s_a (resp. s_b) un bit de l'état interne masqué par un bit m_a (resp. m_b). Soit x_a (resp. x_b) la valeur correcte du bit, c'est-à-dire $s_a = m_a \oplus x_a$ (resp. $s_b = m_b \oplus x_b$). Alors, pour effectuer une opération AND, la technique consiste à générer un bit aléatoire r et à calculer

$$(((r \oplus (m_a m_b)) \oplus (m_a s_b)) \oplus (m_b s_a)) \oplus (s_a s_b)$$

au lieu de $s_a s_b$ afin d'appliquer le masque r sur la valeur $x_a x_b$ puisque

$$\begin{aligned} s_a s_b &= (x_a \oplus m_a)(x_b \oplus m_b) \\ &= x_a x_b \oplus m_a x_b \oplus m_b x_a \oplus m_a m_b \end{aligned}$$

et

$$\begin{aligned} m_a s_b \oplus m_b s_a \oplus m_a m_b &= m_a(x_b \oplus m_b) \oplus m_b(x_a \oplus m_a) \oplus m_a m_b \\ &= m_a x_b \oplus m_b x_a \oplus m_a m_b \end{aligned}$$

La construction pour l'opération OR est similaire, elle consiste à calculer

$$(((\overline{r} \oplus (\overline{m_a}|\overline{m_b})) \oplus (\overline{m_a}|s_b)) \oplus (\overline{m_b}|s_a)) \oplus (s_a|s_b)$$

où $\overline{x} = x \oplus 1$, afin d'appliquer le masque r sur la valeur $x_a|x_b$.

Lorsque les deux bits sont masqués par d masques logiques $m_{a,i}$, $m_{b,i}$ pour $0 \leq i < d$, c'est-à-dire lorsque

$$s_a = \left(\bigoplus_{i=0}^{d-1} m_{a,i}\right) \oplus x_a \text{ et } s_b = \left(\bigoplus_{i=0}^{d-1} m_{b,i}\right) \oplus x_b,$$

la technique consiste de façon similaire à générer d bits aléatoires r_i avec $0 \leq i < d$, et à ajouter par un XOR les termes nécessaires à $s_a s_b$ ou $s_a|s_b$ pour obtenir

$$\left(\bigoplus_{i=0}^{d-1} r_i\right) \oplus x_a x_b \text{ ou } \left(\bigoplus_{i=0}^{d-1} r_i\right) \oplus x_a|x_b.$$

L'utilisation du schéma ISW a également été proposée pour mettre à jour un masque arithmétique après une opération non-linéaire de multiplication sur plusieurs bits [Rivain et al. 2010], et d'autres techniques dérivées du schéma ISW ont été introduites pour réduire son coût en réduisant le nombre d'utilisations des différents masques dans les calculs [Battistello et al. 2016].

Enfin, dans le cas où le schéma de chiffrement utilise une S-box, certaines techniques proposent de modifier directement la S-box afin de calculer la valeur correcte du masque [Carlet et al. 2012], [Ahn et al. 2015].

2.5 Le parallélisme par tranches de bits pour faciliter le masquage

Le coût du masquage dépend principalement des opérations effectuées par le schéma de chiffrement. Par exemple, masquer une structure ARX requiert un coût très élevé [ADOMNICAI et al. 2017]. En effet, ce type de structure nécessite à la fois un masquage logique et arithmétique, et les techniques proposées à ce jour pour passer de l'un à l'autre [GOUBIN 2001], [CORON et al. 2003] sont très coûteuses.

Au contraire, certains schémas de chiffrement comme PICARO [PIRET et al. 2012], Zorro [GÉRARD et al. 2013] ou les LS-Designs [GROSSO et al. 2015] ont une structure qui facilite le masquage. Le parallélisme par tranches de bits permet notamment de réduire le surcoût qu'il engendre [GROSSO et al. 2015], [GOUDARZI et al. 2016], [GOUDARZI et al. 2017].

Les opérations AND/OR effectuées bit à bit sont en effet les seules opérations non-linéaires utilisées par cette technique d'implémentation, et elles sont appliquées directement entre des mots du bloc de données, ce qui permet d'utiliser facilement une technique de masquage dérivée du schéma ISW.

Cependant, en plus du surcoût qu'engendrent les opérations non-linéaires, chacun des masques requiert une application supplémentaire du schéma de chiffrement à chaque exécution, c'est-à-dire le double d'opérations en mémoire (resp. en temps), afin de mettre à jour sa valeur spatialement (resp. temporellement).

2.6 Amélioration sur une architecture 32 bits

Afin de diminuer le surcoût d'une technique de masquage dérivée du schéma ISW sur une architecture 32 bits, nous proposons d'utiliser une implémentation 16 bits du schéma de chiffrement à sécuriser, simultanément appliquée sur les deux blocs de 16 bits dans chaque mot de 32 bits.

Les mots manipulés sont alors composés d'un bloc de 16 bits de données et d'un masque de 16 bits. Dans un premier temps, l'implémentation masquée stocke chaque bloc de 16 bits du message dans un mot de 32 bits, avant d'appliquer un masque aléatoire par un XOR sur les deux parties du mot de 32 bits.

Soit P un bloc de 16 bits de données et M le masque de 16 bits correspondant. La figure 2.11 illustre alors un exemple de mot de 32 bits en entrée de l'implémentation masquée avec notre approche.

Figure 2.11.: Structure d'un mot de 32 bits en entrée de notre implémentation masquée.

La partie linéaire du schéma de chiffrement n'est alors pas modifiée puisqu'elle permet de calculer la valeur correcte du masque sans aucun surcoût. Afin de calculer la valeur correcte du masque pour les opérations non-linéaires, ou plutôt de le remplacer par un nouveau masque, nous proposons deux techniques possibles.

L'objectif est une fois de plus de modifier le résultat de l'opération appliquée aux deux valeurs masquées afin d'obtenir la somme par un XOR entre le résultat de la même opération appliquée aux valeurs non-masquées et un nouveau masque en utilisant une technique de masquage dérivée du schéma ISW. La première technique utilise un nouveau masque aléatoire de 16 bits tandis que la seconde utilise le résultat du XOR entre les deux masques des valeurs en entrée.

Soit S_a (resp. S_b) un mot de 16 bits de l'état interne masqué par un mot M_a (resp. M_b). Soit X_a (resp. X_b) la valeur correcte du mot de 16 bits, c'est-à-dire $S_a = X_a \oplus M_a$ et $S_b = X_b \oplus M_b$. La première technique génère un masque aléatoire de 16 bits M_{ab} tandis que la seconde calcule directement $M_{ab} = M_a \oplus M_b$ comme nouveau masque aléatoire. Alors, la construction pour l'opération AND consiste à effectuer

$$(((M_{ab} \oplus (M_a M_b)) \oplus (M_a S_b)) \oplus (M_b S_a)) \oplus (S_a S_b)$$

au lieu de $S_a S_b$ et à stocker le résultat concaténé avec M_{ab}. La construction pour l'opération OR consiste de façon similaire à effectuer

$$(((\overline{M}_{ab} \oplus (\overline{M}_a | \overline{M}_b)) \oplus (\overline{M}_a | S_b)) \oplus (\overline{M}_b | S_a)) \oplus (S_a | S_b)$$

au lieu de $S_a | S_b$ et à stocker le résultat concaténé avec M_{ab}.

Il est à noter que la seconde technique n'est pas toujours applicable car il se peut qu'elle ne protège pas entièrement le schéma de chiffrement. En effet, après chaque opération, chaque bit du masque est une combinaison linéaire de tous ses bits. Il faut donc s'assurer qu'après chaque opération, aucun bit n'est égal à la combinaison nulle, autrement cela signifie que les bits correspondant de l'état interne ne sont pas masqués. Il est également possible de remplacer $M_a \oplus M_b$ par seulement M_a ou M_b.

Le masquage d'ordre d consiste à appliquer d masques différents sur le message clair, et à calculer la valeur correcte de chaque masque au cours de l'exécution, indépendamment des autres masques, en modifiant de façon similaire les opérations AND/OR, c'est-à-dire en générant et en appliquant d nouveaux masques de 16 bits aléatoires au lieu d'un seul dans le cas de la première technique, ou en appliquant le XOR entre tous les masques des valeurs en entrée pour la seconde technique.

Par contre, afin de stocker deux masques supplémentaires, le masquage n'utilise cette fois qu'un registre additionnel seulement par mot de 16 bits en entrée, c'est-à-dire que calculer la valeur correcte de deux masques spatialement (resp. temporellement) ne coûte que le double d'opérations en mémoire (resp. en temps).

Conclusion du chapitre

Les attaques par observation sont de réelles menaces pour les objets connectés. Nous avons vu au cours de ce chapitre que les chemins d'attaques sont nombreux, et que les techniques pour les mettre en œuvre sont diverses et variées. Il est donc nécessaire de les prendre en considération dans le contexte de l'IoT.

Nous avons notamment proposé un nouveau chemin d'attaque sur les implémentations qui utilisent le parallélisme par tranches de bits, et nous l'avons validé expérimentalement en laboratoire sur une cible réelle [Adomnicai et al. 2016].

Bien qu'il existe aussi de nombreuses contre-mesures pour se prémunir de ces attaques, le surcoût qu'elles demandent n'est généralement pas en accord avec les besoins en terme de performances des objets connectés.

Le masquage demande un surcoût raisonnable sur les implémentations qui utilisent le parallélisme par tranches de bits, et nous avons finalement proposé une optimisation possible du masquage sur les architectures 32 bits.

Nous allons à présent introduire la deuxième grande catégorie d'attaques physiques : les attaques par perturbation.

Attaques par perturbation et contre-mesures

Chapitre 3

> *L'ennemi attaque invariablement à deux occasions : quand il est prêt, et quand vous ne l'êtes pas.*
>
> — **Bernard Werber**

Introduction

Une attaque par perturbation exploite certaines fautes obtenues lors de l'exécution d'un processus sur un circuit intégré. Après avoir décrit les différentes techniques d'injection de fautes et les méthodes permettant à un attaquant d'en exploiter les effets, nous introduisons une attaque par perturbation générique que nous avons proposée sur les chiffrements SPN et que nous avons validée en laboratoire. Nous présentons ensuite les possibilités de contre-mesures logicielles et matérielles connues dans la littérature pour se prémunir de ces attaques. Ces travaux ont fait l'objet d'un article présenté à la conférence "CRiSIS" en 2016 [Lac et al. 2016] et d'un article présenté à la conférence "COSADE" en 2017 [Lac et al. 2017].

Sommaire

3.1 Techniques d'injection de fautes **46**
3.1.1 Perturbation de la tension d'alimentation 46
3.1.2 Modification de la température 47
3.1.3 Injection électromagnétique 47
3.1.4 Perturbation du signal d'horloge 48
3.1.5 Injection par rayonnement lumineux 49
3.2 Exploitation des fautes obtenues **49**
3.2.1 Analyse simple de fautes 50
3.2.2 Analyse différentielle de fautes 51
3.2.3 Analyse d'erreurs sur l'exécution 53
3.3 DFA généralisée aux chiffrements SPN **54**
3.3.1 Application aux différentes structures SPN 54
3.3.2 Estimation du nombre de fautes requises 62
3.3.3 Simulation avec un modèle de faute idéal 67
3.3.4 Exemples de DFA conduite en laboratoire 69
3.4 Contre-mesures **81**
3.4.1 Matérielles 82
3.4.2 Logicielles 82
3.4.3 Évaluation des contre-mesures face à la DFA 85
3.5 Intra-Instruction Redundancy **87**

3.1 Techniques d'injection de fautes

Certaines grandeurs physiques, comme la consommation de courant, la température, le rayonnement électromagnétique, la fréquence d'horloge ou le rayonnement lumineux peuvent être exploitées par un attaquant afin de perturber le fonctionnement d'un circuit intégré [BAR-EL et al. 2006].

Les techniques d'injection de fautes actuellement proposées dans la littérature exploitent ces grandeurs physiques, et permettent de corrompre l'exécution d'un schéma de chiffrement, de modifier des valeurs, de contourner des tests de droits d'accès ou de provoquer un DoS. Les effets qu'il est possible d'obtenir dépendent alors de la technique d'injection de fautes utilisée et du circuit intégré ciblé, et peuvent être exploités par différentes méthodes en fonction des fautes obtenues.

Il est très important d'étudier les informations qu'il est possible d'acquérir à partir des différentes fautes réalisables, puisque les techniques d'injection de fautes sont non-invasives, c'est-à-dire ne requièrent aucune modification du circuit intégré, ou semi-invasives, c'est-à-dire ne demandent qu'une ouverture du boîtier.

De plus, la plupart ont un coût relativement faible de mise en œuvre, et peuvent donc être facilement réalisées.

3.1.1 Perturbation de la tension d'alimentation

Modifier la tension d'alimentation du circuit intégré à un instant précis peut provoquer deux effets exploitables par un attaquant.

Le premier effet peut être un changement d'état sur un ou plusieurs fils électriques du circuit intégré, ce qui peut provoquer une modification sur l'entrée, et donc la sortie, d'une ou plusieurs portes logiques. Il est en effet possible de constater un changement d'état lorsque la valeur absolue de la tension électrique sur certains fils, normalement supérieure (resp. inférieure) à la tension de seuil du circuit intégré, devient inférieure (resp. supérieure) à cette dernière [DJELLID-OUAR et al. 2006].

Le second effet peut être une modification de la vitesse de circulation des données entre des éléments de mémorisation, ce qui peut entraîner une violation des contraintes temporelles du circuit intégré [ZUSSA et al. 2013]. Il est possible d'observer une violation des contraintes temporelles lorsque le délai d'un accès à une mémoire devient plus long qu'un cycle d'horloge [CARPI et al. 2013].

Dans les deux cas, ces effets provoquent des fautes globales sur l'exécution d'un schéma de chiffrement, c'est-à-dire des fautes qui modifient la valeur de plusieurs bits de données [TUMMELTSHAMMER et al. 2009].

3.1.2 — Modification de la température

Une modification de la température en dehors des conditions normales d'utilisation d'un circuit intégré peut entraîner une violation des contraintes temporelles du circuit, due une fois de plus à un ralentissement de la vitesse de circulation des données entre des éléments de mémorisation [BROUCHIER et al. 2009].

C'est en particulier le cas lors de l'accès à certaines mémoires non-volatiles utilisées par le circuit (NVM en anglais pour "Non-Volatile Memory") comme la mémoire EEPROM (pour "Electrically-Erasable Programmable Read-Only Memory") ou la mémoire flash [SKOROBOGATOV 2009].

Les perturbations obtenues provoquent là encore des fautes globales sur l'exécution d'un schéma de chiffrement [HUTTER et al. 2013].

3.1.3 — Injection électromagnétique

L'apparition d'un champ électromagnétique sur un circuit intégré peut provoquer des courants sur certains fils électriques et ainsi perturber le fonctionnement du circuit [QUISQUATER et al. 2002].

Les courants ainsi provoqués sur certains fils du circuit intégré peuvent entraîner un changement d'état sur les fils, et provoquent là encore des fautes globales sur l'exécution d'un schéma de chiffrement [SCHMIDT et al. 2007].

L'utilisation d'un générateur d'impulsions de tension relié à une antenne d'injection, qui peut être un solénoïde avec un matériau ferromagnétique comme coeur du solénoïde ou sans rien d'autre, placée en champ proche du circuit intégré permet de générer un champ électromagnétique sur ce dernier [DEHBAOUI et al. 2012].

Un exemple de banc d'injection électromagnétique que nous avons utilisé en laboratoire est présenté sur la figure 3.1, où nous avons relié un générateur d'impulsions de tension à un solénoïde, placé en champ proche d'un FPGA, lui-même relié à un ordinateur de contrôle via un UART. Nous avons également utilisé un oscilloscope, relié au générateur d'impulsions de tension, afin d'afficher temporellement le pic de tension provoqué par ce dernier.

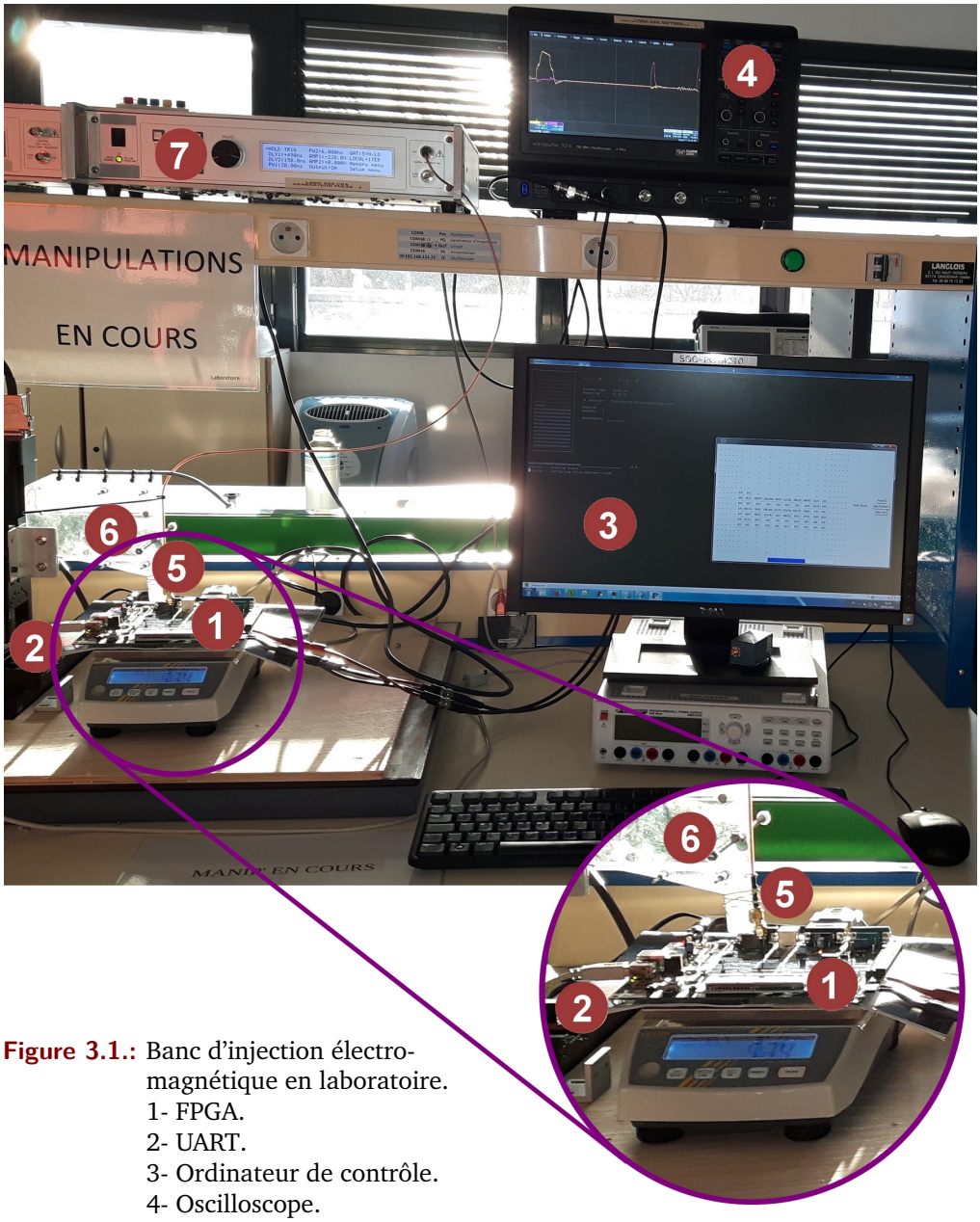


Figure 3.1.: Banc d'injection électromagnétique en laboratoire.
1- FPGA.
2- UART.
3- Ordinateur de contrôle.
4- Oscilloscope.
5- Antenne d'injection.
6- Support d'antenne.
7- Générateur d'impulsions de tension.

3.1.4 — Perturbation du signal d'horloge

Une augmentation de la fréquence d'horloge au dessus de la fréquence de fonctionnement maximale du circuit intégré modifie le signal d'horloge du circuit, ce qui peut provoquer une violation ponctuelle des contraintes temporelles du circuit intégré.

C'est notamment le cas lorsqu'un cycle d'horloge devient plus court qu'un accès à une mémoire ou lorsque le circuit exécute l'instruction $n+1$ avant que le microprocesseur n'ait terminé l'exécution de l'instruction n [AGOYAN et al. 2010].

L'effet obtenu est donc similaire à une brève modification de la tension d'alimentation du circuit intégré, ce qui provoque des fautes globales sur l'exécution d'un schéma de chiffrement [Zussa et al. 2013].

3.1.5 Injection par rayonnement lumineux

L'utilisation d'un laser ou d'une source lumineuse focalisée au contact du silicium du circuit intégré peut générer des paires d'électrons-trous le long du faisceau lumineux aboutissant à l'apparition d'un courant photoélectrique au niveau d'un transistor, ce qui peut entraîner un changement d'état en entrée ou en sortie de ce dernier [Skorobogatov et al. 2003].

Les changements d'état ainsi obtenus sur certains fils du circuit provoquent des fautes globales sur l'exécution d'un schéma de chiffrement, mais il est plus facile de ne cibler qu'un seul transistor avec cette technique, ce qui permet à un attaquant de ne modifier qu'un seul ou quelques bits de données contenus dans un registre ou une cellule mémoire avec beaucoup plus de précision [Roscian et al. 2013].

Bien que cette technique d'injection soit la plus précise, elle requiert un matériel souvent coûteux et demande de décapsuler le boîtier du circuit intégré afin d'être directement au contact du silicium [Dutertre et al. 2011].

3.2 Exploitation des fautes obtenues

Les différents effets qu'il est possible d'obtenir suite à une perturbation sur l'exécution d'un schéma de chiffrement sont les suivants [Verbauwhede et al. 2011] :

- Mise à un de n bits ("n-bit set" en anglais) : la faute force à 1 certains bits des données à un instant précis, seuls les bits à 0 sont modifiés.
- Mise à zéro de n bits ("n-bit reset" en anglais) : la faute force à 0 certains bits des données à un instant précis, seuls les bits à 1 sont modifiés.
- Inversion de n bits ("n-bit flip" en anglais) : la faute inverse la valeur de certains bits des données à un instant précis, tous les bits ciblés sont modifiés.
- Remplacement d'instructions : la faute entraîne l'exécution d'une autre instruction à un instant précis, la plus courante étant le saut d'instruction ("instruction skip" en anglais), qui correspond dans ce cas à un remplacement de l'instruction qui n'est pas exécutée par l'instruction NOP.

Par exemple, l'injection d'impulsions de tension peut produire une mise à un, une mise à zero ou une inversion d'un registre, et dans ce cas n est égal à la taille du registre, mais la faute la plus courante avec ce moyen d'injection sur une implémentation logicielle est le saut d'instruction [MORO 2014].

L'injection par rayonnement lumineux peut produire quant à elle, comme nous l'avons déjà mentionné, une mise à un, une mise à zéro ou une inversion d'un seul bit (dans ce cas $n = 1$) ou d'un seul octet (dans ce cas $n = 8$).

Il existe trois principales catégories d'attaques par perturbation : l'analyse simple de fautes, l'analyse différentielle de fautes et l'analyse d'erreurs sur l'exécution.

3.2.1 — Analyse simple de fautes

Une analyse simple de fautes consiste à directement examiner l'effet de chaque faute sur l'exécution du schéma de chiffrement.

L'analyse la plus répandue exploite le fait qu'une faute a ou n'a pas eu d'effet sur l'exécution (on parle de "safe-error analysis" en anglais) [YEN et al. 2000].

Par exemple, la mise à un d'un bit de la clé permet à l'attaquant de retrouver la valeur de ce bit : si le chiffré obtenu n'a pas été modifié alors la valeur du bit est 1, sinon la valeur du bit est 0.

Cependant, ce type de fautes est extrêmement difficile à obtenir en pratique, d'un côté parce que cela nécessite un moyen d'injection très précis comme un laser afin de ne cibler qu'un seul bit, et d'un autre parce qu'il faut être en mesure de connaître l'emplacement de la clé dans la mémoire du circuit intégré.

Un autre exemple est le fait de cibler une fonction qui est appliquée suivant la valeur d'un bit de la clé comme la fonction MULTIPLY du code de l'exponentiation modulaire sécurisé face aux analyses simples par observation décrit à la figure 2.10.

En effet, si le chiffré obtenu n'a pas été modifié malgré l'injection d'une faute dans la fonction MULTIPLY lors d'une itération donnée de l'exponentiation modulaire, alors la valeur du bit correspondant de l'exposant secret est égale à 0, sinon elle est égale à 1, puisque les données traitées sont modifiées par la sortie de la fonction MULTIPLY seulement si la valeur du bit de l'exposant secret est égale à 1.

3.2.2 Analyse différentielle de fautes

Une analyse différentielle de fautes (DFA en anglais pour “Differential Fault Analysis”) consiste à exécuter un schéma de chiffrement plusieurs fois sur un même message clair avec et sans injection de fautes, puis à comparer la valeur correcte C et la valeur corrompue C^* du chiffré pour chaque faute obtenue [Boneh et al. 1997].

Dans le cas des schémas de chiffrement à clé publique, il est possible d’exploiter directement la valeur de la différence $\Delta C = C - C^*$. Par exemple, le RSA est souvent effectué avec une implémentation de l’algorithme d’exponentiation modulaire basée sur le théorème des restes chinois (CRT en anglais pour “Chinese Remainder Theorem”) afin de calculer $M = C^d$ modulo n avec $n = pq$, où d, p et q sont secrets, C est le chiffré et M le message clair [Nozaki et al. 2001].

Pour cela, l’implémentation calcule $M_1 = C^d$ modulo p, $M_2 = C^d$ modulo q et $M = aM_1 + bM_2$ modulo n avec a et b deux valeurs fixées par le CRT telles que

$$\begin{cases} a \equiv 1 \text{ modulo } p \\ a \equiv 0 \text{ modulo } q \end{cases} \text{ et } \begin{cases} b \equiv 0 \text{ modulo } p \\ b \equiv 1 \text{ modulo } q \end{cases}$$

La DFA consiste alors à perturber le calcul de M_1 afin d’obtenir une valeur corrompue $M^* = aM_1^* + bM_2$, et à calculer $\Delta M = M - M^* = a(M_1 - M_1^*)$ [Boneh et al. 1997].

Alors, si $M_1 - M_1^*$ n’est pas divisible par p, l’attaquant peut utiliser le PGCD (pour “Plus Grand Commun Diviseur”) afin de calculer

$$\text{PGCD}(\Delta M, n) = \text{PGCD}(a(M_1 - M_1^*), n) = q$$

et ainsi retrouver la valeur du secret q, et d’en déduire facilement la valeur de p et d, puisque $ed \equiv 1$ modulo $(p\text{-}1)(q\text{-}1)$.

En ce qui concerne la cryptographie symétrique, il est possible d’exploiter directement la valeur de la différence $\Delta C = C \oplus C^*$ [Moro 2014]. Par exemple, la figure 3.2 illustre un saut d’instruction obtenu par un attaquant sur la dernière addition de clé d’un chiffrement itératif par blocs. Dans ce cas, l’attaque consiste simplement à calculer la valeur de la différence en sortie du schéma de chiffrement qui est directement égale à la valeur de la clé du dernier tour.

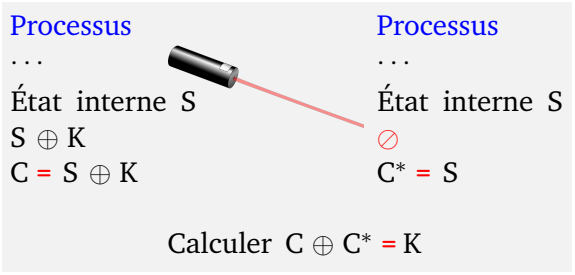


Figure 3.2.: Illustration d’un exemple d’analyse différentielle de fautes.

Il est également possible d'exploiter la répartition des fautes sur un mot donné, c'est-à-dire de calculer la probabilité d'apparition de chaque valeur de faute en fonction de la valeur du mot [LASHERMES et al. 2012].

Il est aussi possible de retrouver des informations à partir des différences en entrée et sortie de la couche de substitution lorsque le schéma de chiffrement utilise une ou plusieurs S-boxes, en exploitant la table de distribution des différences de chaque S-box [BIHAM et al. 1997]. En effet, cette dernière donne le nombre de candidats possibles pour x et y en fonction de la valeur de $x \oplus y$ en entrée d'une S-box $\mathcal{S}$ et de la valeur de $\mathcal{S}(x) \oplus \mathcal{S}(y)$ en sortie.

L'attaquant peut alors calculer les candidats pour une valeur x en entrée d'une S-box de n bits lorsqu'il obtient une différence en entrée et une différence en sortie de cette dernière (on parle de différentielle pour désigner un tel couple de différences) et si x peut s'exprimer avec pour seule inconnue une partie de n bits K^n de la clé secrète, il en déduit des candidats pour K^n.

Finalement, il peut réitérer l'attaque avec d'autres différentielles pour obtenir d'autres candidats pour K^n, et retrouver sa valeur par intersection, puis appliquer ces mêmes étapes sur les autres parties de la clé pour la retrouver entièrement.

C'est généralement la technique la plus employée sur les chiffrements par blocs, mais elle requiet la connaissance des différences autour d'une S-box, et nécessite souvent pour cela un modèle de faute très précis sur un seul bit [GIRAUD 2004] ou un seul octet [PIRET et al. 2003].

Par exemple, le tableau 3.1 donne les valeurs de la S-box utilisée par PRIDE et de sa table de distribution des différences définie par

$$T(i,j) = \#\{(x,y) \in \{0,1\}^4 \times \{0,1\}^4 \mid x \oplus y = i, \mathcal{S}(x) \oplus \mathcal{S}(y) = j\}.$$

Il est facile de vérifier qu'il y a effectivement 4 valeurs possibles pour x et y qui vérifient $x \oplus y = 1$ et $\mathcal{S}(x) \oplus \mathcal{S}(y) = 4$, c'est-à-dire $x, y \in \{0\text{x}0, 0\text{x}1, 0\text{x}4, 0\text{x}5\}$.

De ce fait, si un attaquant obtient une différentielle

$$(\Delta a, \Delta b) = (x \oplus y, \mathcal{S}(x) \oplus \mathcal{S}(y)) = (0\text{x}1, 0\text{x}4)$$

sur la S-box de PRIDE où x peut s'exprimer avec pour seule inconnue un quartet de la clé secrète, alors il en déduit que ce quartet est égal à 0x0, 0x1, 0x4 ou 0x5.

Tableau 3.1.: S-box de PRIDE **(a)** et sa table de distribution des différences **(b)**.

(a) S-box de PRIDE.

x	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
$\mathcal{S}(x)$	0x0	0x4	0x8	0xf	0x1	0x5	0xe	0x9	0x2	0x7	0xa	0xc	0xb	0xd	0x6	0x3

(b) Table de distribution des différences de la S-box de PRIDE.

T	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
0x0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0x1	0	0	0	0	4	4	4	4	0	0	0	0	0	0	0	0
0x2	0	0	0	0	0	0	0	0	4	0	0	4	2	2	2	2
0x3	0	0	0	0	0	0	0	0	4	0	0	4	2	2	2	2
0x4	0	4	0	0	0	0	4	0	0	2	2	0	2	0	0	2
0x5	0	4	0	0	0	4	0	0	0	2	2	0	2	0	0	2
0x6	0	4	0	0	4	0	0	0	0	2	2	0	0	2	2	0
0x7	0	4	0	0	0	0	0	4	0	2	2	0	0	2	2	0
0x8	0	0	4	4	0	0	0	0	4	0	4	0	0	0	0	0
0x9	0	0	0	0	2	2	2	2	0	0	0	0	2	2	2	2
0xa	0	0	0	0	2	2	2	2	4	0	4	0	0	0	0	0
0xb	0	0	4	4	0	0	0	0	0	0	0	0	2	2	2	2
0xc	0	0	2	2	2	2	0	0	0	2	0	2	2	0	2	0
0xd	0	0	2	2	0	0	2	2	0	2	0	2	0	2	0	2
0xe	0	0	2	2	0	0	2	2	0	2	0	2	2	0	2	0
0xf	0	0	2	2	2	2	0	0	0	2	0	2	0	2	0	2

3.2.3 — Analyse d'erreurs sur l'exécution

Certaines erreurs peuvent modifier l'exécution d'une fonction, et permettre à un attaquant d'exploiter directement la valeur corrompue du message, ou peuvent également empêcher l'exécution d'une fonction, et permettre de contourner des vérifications ou de provoquer un DoS [Van Woudenberg et al. 2011b].

Dans le cas de la cryptographie asymétrique, il est possible par exemple de modifier la valeur d'un paramètre utilisé par le chiffrement, comme la valeur de $n = pq$ dans le cas du RSA [Berzati et al. 2009] ou la valeur d'un des paramètres définissant la courbe sur laquelle sont effectuées les opérations dans le cas des courbes elliptiques [Biehl et al. 2000], afin de réduire le problème à résoudre à un cas plus simple.

Du côté des schémas de chiffrement à clé secrète, il est possible par exemple d'empêcher un chiffrement par blocs d'effectuer tous ses tours en réduisant l'indice de tour avec une faute, puis en appliquant des analyses mathématiques sur une version réduite du chiffrement [Dutertre et al. 2012].

3.3 DFA généralisée aux chiffrements SPN

Comme nous venons de le constater, la plupart des attaques par perturbation requièrent des fautes assez précises pour être mises en œuvre, comme une modification d'un seul bit, d'un seul octet ou un saut d'instruction d'une opération particulière.

Bien que ces modèles de faute soient réalistes au vu des techniques actuelles, notamment avec le laser ou l'injection électromagnétique, leur faisabilité en pratique dépend fortement du contexte dans lequel se place l'attaque.

Aujourd'hui, les chiffrements SPN légers favorisant le parallélisme par tranches de bits sont souvent proposés pour répondre aux besoins des objets connectés, notamment pour faciliter le masquage comme PRIDE [ALBRECHT et al. 2014b], les LS-Designs [GROSSO et al. 2015] ou les structures CUBE [BERGER et al. 2015].

En étudiant la structure de ces chiffrements, nous avons identifié un nouveau chemin d'attaque qui permet de réduire les contraintes liées aux modèles de faute requis pour la réalisation d'une attaque, très efficace sur ce type de structure et généralisable à la plupart des autres chiffrements SPN.

3.3.1 Application aux différentes structures SPN

Bien que certaines attaques par perturbation soient applicables aux chiffrements SPN, comme un saut d'instruction sur la dernière addition de clé, ou une modification d'un bit ou d'un octet avant la dernière couche de substitution, les modèles de faute requis sont généralement assez précis et ne laissent que peu de choix possibles à l'attaquant sur la technique d'injection de fautes à utiliser.

Le chemin d'attaque que nous proposons permet de réduire ces contraintes, en exploitant la diffusion de l'étage linéaire avant la dernière couche de substitution, et peut être réalisé à partir de fautes aléatoires sur des mots dont la taille dépend de la structure du schéma de chiffrement.

3.3.1.1 Structures favorisant le parallélisme par tranches de bits

Comme nous l'avons mentionné et constaté sur la figure 2.5, une structure facilitant le parallélisme par tranches de bits qui utilise une S-box sur m bits et prend en entrée un bloc de données de mn bits, fait correspondre pour tout $1 \leq i \leq n$ et pour tout $1 \leq j \leq m$, le bit i du mot j en entrée de la couche de substitution au bit j en entrée de la S-box i.

De ce fait, une modification d'un des mots de n bits en entrée de la dernière couche de substitution permet à un attaquant d'obtenir une différence sur un seul bit en entrée de chaque S-box. Pour chaque bit modifié, l'attaquant obtient ainsi une différentielle non nulle sur la S-box correspondante comme l'illustre la figure 3.3 dans le cas d'un bloc de 64 bits et d'une S-box $\mathcal{S}$ de 4 bits.

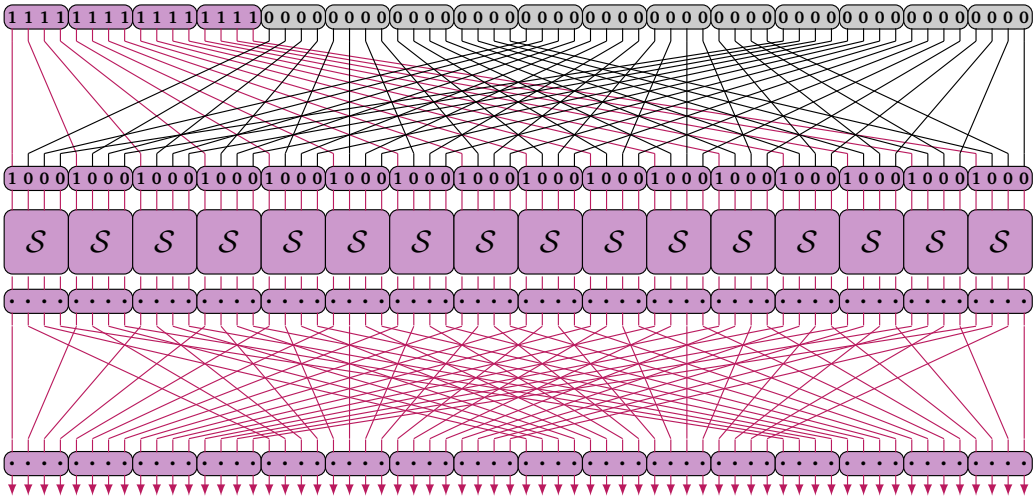


Figure 3.3.: Inversion de 16 bits en entrée d'une structure favorisant le parallélisme par tranches de bits sur un bloc de 64 bits avec une même S-box $\mathcal{S}$ de 4 bits.

L'étage linéaire d'une structure favorisant le parallélisme par tranches de bits consiste généralement à effectuer des opérations sur les mots de n bits qui sont ensuite diffusés par la couche de substitution.

De ce fait, un attaquant est souvent capable de corrompre un mot précis avant la dernière couche de substitution en perturbant l'exécution de l'étage linéaire, et obtient donc des différentielles connues sur chaque S-box.

En effet, il peut tout d'abord calculer la différence en sortie de la dernière couche de substitution, à partir de la différence entre la valeur correcte et la valeur corrompue du chiffré d'un même message, puisque les opérations effectuées après la dernière couche de substitution sont linéaires.

Ensuite, chaque S-box active, c'est-à-dire dont la différence en sortie est non nulle, lui permet de connaître la valeur de la différence en entrée, et donc de retrouver la valeur de la faute injectée.

Le fait d'être capable de modifier un mot précis à partir d'une injection de fautes sur l'étage linéaire dépend d'un côté de la structure du schéma de chiffrement, et d'un autre de l'implémentation utilisée.

3.3.1.1.a - Application à PRIDE

Les spécifications de PRIDE sont données dans la section 2.3.2.1.

Dans ce chapitre, nous utilisons les notations suivantes pour désigner les différentes valeurs intermédiaires de l'état interne lors d'un tour $1 \leq r \leq 19$ de PRIDE :

- I_{r} : l'entrée du tour r,
- X_{r} : l'état interne après l'addition de la clé du tour r,
- Y_{r} : l'état interne après la couche de substitution du tour r,
- Z_{r} : l'état interne après la permutation des bits du tour r,
- W_{r} : l'état interne après la couche de matrices du tour r,
- O_{r} : la sortie du tour r.

C'est-à-dire qu'un tour de PRIDE effectue les étapes suivantes :

- Appliquer la permutation inverse $\mathcal{P}^{-1}$ à $f_{\mathrm{r}}(\mathrm{K}_1)$ et appliquer un XOR entre le résultat obtenu et l'entrée du tour r : $X_{\mathrm{r}} = I_{\mathrm{r}} \oplus \mathcal{P}^{-1}(f_{\mathrm{r}}(\mathrm{K}_1))$,
- Appliquer la S-box $\mathcal{S}$ à chaque quartet de X_{r} (c'est-à-dire appliquer la couche de substitution S-layer à X_{r}) : $Y_{\mathrm{r}} = \text{S-layer}(X_{\mathrm{r}})$,
- Appliquer la permutation $\mathcal{P}$ à Y_{r} : $Z_{\mathrm{r}} = \mathcal{P}(Y_{\mathrm{r}})$,
- Multiplier le mot $0 \leq i \leq 3$ de 16 bits de Z_{r} par la matrice $\mathcal{L}_{\mathrm{i}}$ (c'est-à-dire appliquer la couche de matrices M-layer à Z_{r}) : $W_{\mathrm{r}} = \text{M-layer}(Z_{\mathrm{r}})$,
- Appliquer $\mathcal{P}^{-1}$ à W_{r} pour obtenir la sortie du tour r : $O_{\mathrm{r}} = \mathcal{P}^{-1}(W_{\mathrm{r}})$,

à part pour le dernier tour où seules les deux premières étapes sont appliquées.

Afin de mener la DFA, un attaquant cible dans un premier temps K_0 et peut le retrouver à partir de plusieurs fautes en sortie d'une matrice lors du dernier étage linéaire, ce qui est généralement assez facile à obtenir en pratique. Par exemple, l'implémentation de référence de PRIDE exécute les matrices séparément les unes des autres, il est donc possible de corrompre la sortie de l'une d'entre elles à partir de n'importe quelle injection de faute durant son exécution.

L'attaquant obtient alors une différentielle $(\Delta X_{20}, \Delta Y_{20})$ connue en entrée de la dernière couche de substitution. En effet, la valeur de ΔY_{20} peut être calculée à partir de la différence entre la valeur correcte C et la valeur corrompue C^* du chiffré, et la valeur de ΔX_{20} peut être retrouvée à partir de la valeur de ΔY_{20} en fonction du nombre de S-boxes actives et de la matrice ciblée : si le quartet i de ΔY_{20} est nul, alors le quartet i de ΔX_{20} est nul également. Par contre, si le quartet i de ΔY_{20} n'est pas nul, alors le quartet i de ΔX_{20} a un poids de Hamming égal à 1 dont la valeur est définie par la position du mot corrompu.

L'attaquant connaît de cette façon les valeurs suivantes :

$$\Delta X_{20} \quad = \quad \text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}^*) \oplus \text{K}_0)$$

$$\text{et} \quad \Delta Y_{20} \quad = \quad (\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus (\mathcal{P}^{-1}(\text{C}^*) \oplus \text{K}_0)$$

Ensuite, il peut utiliser cette différentielle pour obtenir des informations sur les quartets $0 \leq i \leq 15$ de K_0 pour chaque S-box active :

$$x = (\mathcal{P}^{-1}(\text{C}))_i \oplus (\text{K}_0)_i \text{ et } y = (\mathcal{P}^{-1}(\text{C}^*))_i \oplus (\text{K}_0)_i$$

$$\text{satisfont} \quad x \oplus y = (\Delta Y_{20})_i \text{ et } \mathcal{S}^{-1}(x) \oplus \mathcal{S}^{-1}(y) = (\Delta X_{20})_i.$$

La figure 3.4 illustre la diffusion d'une faute qui a eu pour effet d'inverser les 16 bits en sortie de la première matrice du dernier étage linéaire.

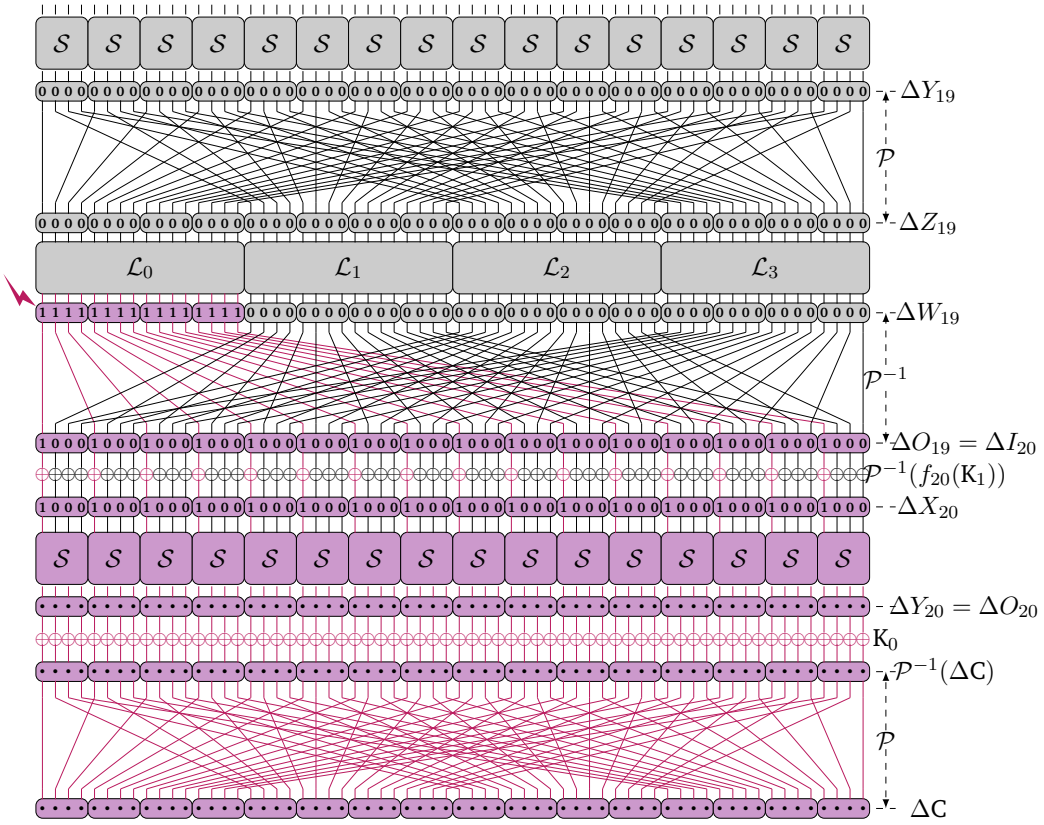


Figure 3.4.: Diffusion d'une faute obtenue lors de l'exécution de PRIDE par une inversion des 16 premiers bits de W_{19}.

L'attaquant obtient ainsi des candidats possibles pour chaque quartet de K_0, et peut réitérer l'attaque sur les autres mots de W_{19} afin de retrouver entièrement K_0 par intersection des ensembles obtenus.

Finalement, pour retrouver K_1, il peut perturber de la même façon l'application d'une matrice lors de l'avant-dernier étage linéaire.

Il obtient alors une différentielle $(\Delta X_{19}, \Delta Y_{19})$ connue en entrée de l'avant-dernière couche de substitution. En effet, la valeur de ΔY_{19} peut être calculée à partir de la connaissance de K_0 et de la différence entre la valeur correcte C et la valeur corrompue C* du chiffré, et la valeur de ΔX_{19} peut être retrouvée à partir de la valeur de ΔY_{19} en fonction du nombre de S-boxes actives et de la matrice ciblée.

L'attaquant connaît de cette façon les valeurs suivantes :

$$\begin{aligned}\Delta X_{19} \quad = \quad & \text{S-layer}^{-1}(\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1)))) \\ & \oplus \text{S-layer}^{-1}(\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}^*) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1))))\end{aligned}$$

$$\text{et} \quad \Delta Y_{19} \quad = \quad \text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C} \oplus \text{K}_0)) \oplus \text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}^* \oplus \text{K}_0)))$$

Ensuite, il peut utiliser cette différentielle pour obtenir des informations sur chaque quartet $0 \leq i \leq 15$ de K_1 :

$$\begin{aligned} & x = (\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1))))_i \\ \text{et} \quad & y = (\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}^*) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1))))_i \end{aligned}$$

$$\text{satisfont} \quad x \oplus y = (\Delta Y_{19})_i \text{ et } \mathcal{S}^{-1}(x) \oplus \mathcal{S}^{-1}(y) = (\Delta X_{19})_i.$$

Il peut enfin retrouver entièrement la clé secrète en réitérant l'attaque sur les autres mots de W_{18} à partir d'injection de fautes sur l'exécution des autres matrices.

3.3.1.1.b - Application aux LS-Designs

Un LS-Design [GROSSO et al. 2015] prend en entrée un bloc représenté par une matrice binaire de ω lignes et c colonnes appelée l'état interne. La couche de substitution consiste alors à appliquer une S-box $\mathcal{S}$ sur chaque colonne de l'état interne, et l'étage linéaire à appliquer une matrice M sur chaque ligne. La figure 3.5 illustre l'état interne d'un LS-Design, et donne un exemple de l'entrée d'une S-box et d'une matrice. Le bit de l'état interne en ligne $1 \leq i \leq \omega$ et colonne $1 \leq j \leq c$ est noté $b_{i,j}$.

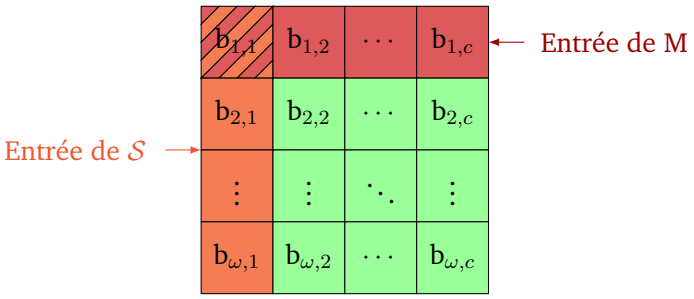


Figure 3.5.: L'état interne d'un LS-Design.

La structure d'un LS-Design est similaire à une structure entrelacée comme PRIDE, à la différence qu'un LS-Design applique la même matrice sur chaque ligne de l'état interne pour son étage linéaire. Appliquer les S-boxes sur les colonnes permet en effet de faciliter le parallélisme par tranches de bits, puisque chaque ligne est diffusée en entrée de chaque S-box.

La plupart des implémentations de LS-Designs traitent les lignes séparément pour appliquer les matrices. De ce fait, une perturbation durant l'application d'une matrice lors du dernier étage linéaire permet à un attaquant d'obtenir des différentielles connues sur chaque S-box de la dernière couche de substitution.

Il peut en effet calculer la valeur de la différence en sortie de la dernière couche de substitution à partir de la différence entre la valeur correcte C et la valeur corrompue C^* du chiffré, et retrouver la valeur de la différence en entrée à partir du nombre de S-boxes actives, comme nous l'avons vu sur PRIDE.

La figure 3.6 illustre la diffusion d'une faute obtenue durant l'application du dernier étage linéaire, où ΔZ_j^i (resp. ΔW_j^i) est la différence en entrée (resp. sortie) de la ligne $1 \leq i \leq \omega$, colonne $1 \leq j \leq c$ du dernier étage linéaire.

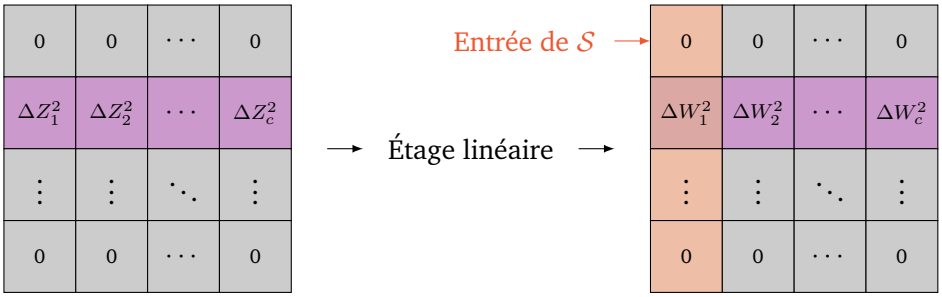


Figure 3.6.: Diffusion d'une faute obtenue lors de l'exécution d'un LS-Design.

L'attaquant obtient ainsi des candidats possibles pour chaque mot de ω bits de la clé secrète, qui est appliquée par un XOR en sortie pour produire le chiffré, et peut réitérer l'attaque sur les autres lignes de l'état interne afin de retrouver entièrement la clé par intersection des ensembles obtenus.

3.3.1.1.c - Application aux structures CUBE

Une structure CUBE [BERGER et al. 2015] prend en entrée un bloc représenté par un cube binaire de $\omega \times \omega \times \omega$ bits. La couche de substitution consiste alors à appliquer une S-box $\mathcal{S}$ sur chaque ligne du cube, et l'étage linéaire à appliquer une matrice M sur les plans horizontaux puis à effectuer la rotation d'axe (X,Y,Z) $\rightarrow$ (Z,X,Y).

La figure 3.7 illustre un bloc de données manipulé par une structure CUBE avec $\omega = 4$, et donne un exemple de l'entrée d'une S-box et d'une matrice.

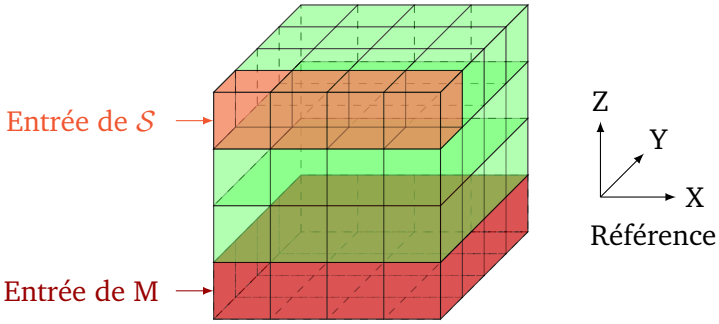


Figure 3.7.: Bloc de données manipulé par une structure CUBE.

Appliquer les S-boxes sur les lignes permet une fois de plus de faciliter le parallélisme par tranches de bits, puisque chaque plan est diffusé en entrée de chaque S-box grâce à la rotation (X,Y,Z) $\rightarrow$ (Z,X,Y) sur le cube.

Là encore, une implémentation traitera généralement les plans séparément les uns des autres lors de l'étage linéaire, ce qui permet à un attaquant d'obtenir facilement des différentielles connues sur chaque S-box de la dernière couche de substitution.

La figure 3.8 illustre la diffusion d'une faute obtenue sur un plan du cube, où ΔZ (resp. ΔW) est la différence en entrée (resp. sortie) du dernier étage linéaire.

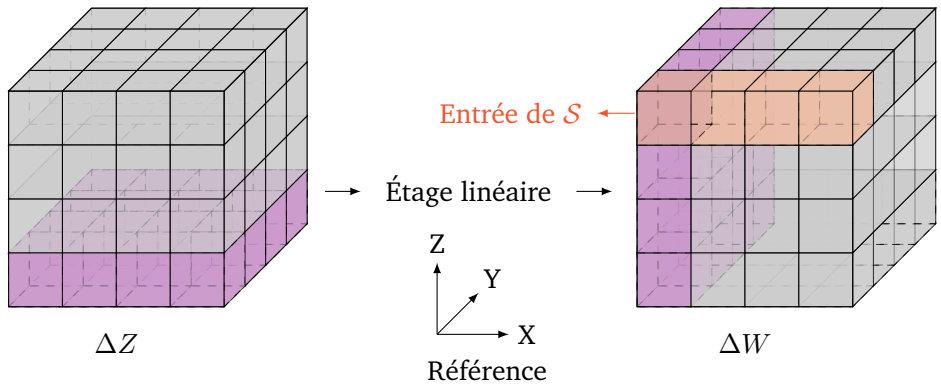


Figure 3.8.: Diffusion d'une faute obtenue lors de l'exécution d'une structure CUBE.

Une fois de plus, l'attaquant obtient des candidats possibles pour chaque mot de ω bits de la clé, qui est appliquée par un XOR en sortie pour obtenir le chiffré, et peut réitérer l'attaque sur les autres plans du cube afin de la retrouver entièrement.

3.3.1.2 – Autres structures SPN

L'application de la DFA que nous avons proposée aux autres structures SPN dépend principalement de la structure de l'étage linéaire du schéma de chiffrement ciblé. En général, l'étage linéaire diffuse au maximum des mots du bloc en entrée de chaque S-box, et c'est cette diffusion qui permet d'appliquer l'attaque. Cependant, certaines structures utilisent un étage linéaire qui diffuse plus que d'autres.

Par exemple, PRESENT [BOGDANOV et al. 2007] est un chiffrement SPN dont la fonction de chiffrement applique la même permutation $\mathcal{P}$ que PRIDE pour son étage linéaire, et utilise une S-box de 4 bits en parallèle pour sa couche de substitution.

La DFA consiste alors comme précédemment à cibler un mot, avant le dernier étage linéaire, qui est diffusé en entrée de plusieurs S-boxes de la dernière couche de substitution, et qui n'affecte qu'un seul bit sur chacune.

Cela revient donc à cibler la sortie, ou l'application, d'une S-box lors de l'avant-dernière couche de substitution. La figure 3.9 illustre une faute obtenue sur le premier quartet en entrée du dernier étage linéaire lors de l'exécution de la fonction de chiffrement de PRESENT.

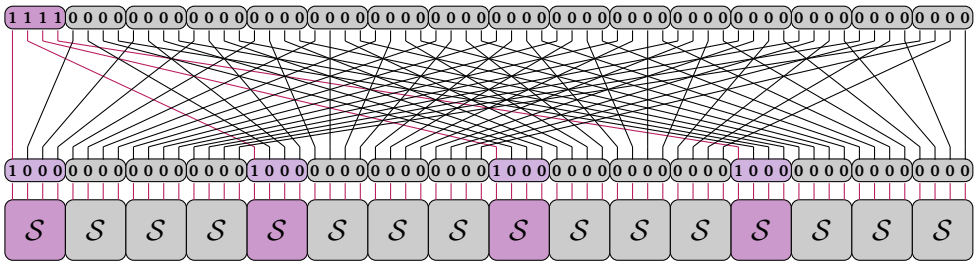


Figure 3.9.: Diffusion d'une faute obtenue lors de l'exécution de la fonction de chiffrement de PRESENT.

On constate immédiatement que l'attaque est moins efficace, puisqu'une faute ne permet pas d'activer toutes les S-boxes de la dernière couche de substitution.

De plus, le modèle de faute nécessaire peut être plus complexe puisque l'attaquant ne doit affecter qu'un quartet du bloc de données, bien que n'importe quelle faute durant l'exécution d'une S-box lors de l'avant-dernière couche de substitution permet à un attaquant d'obtenir l'effet désiré.

Il est à noter que dans le cas de PRESENT, l'attaque est plus efficace sur sa fonction de déchiffrement. En effet, les 16 premiers bits du bloc de données avant l'étage linéaire sont diffusés en entrée de chaque S-box.

De ce fait, un attaquant peut activer toutes les S-boxes de la dernière couche de substitution avec une seule faute lors de l'exécution des quatre premières S-boxes de l'avant-dernière couche de substitution. La figure 3.10 illustre la diffusion d'une faute obtenue sur le premier mot de 16 bits en entrée du dernier étage linéaire lors de l'exécution de la fonction de déchiffrement de PRESENT.

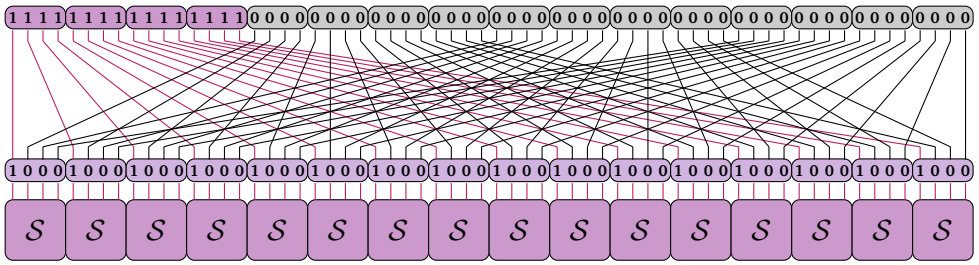


Figure 3.10.: Diffusion d'une faute obtenue lors de l'exécution de la fonction de déchiffrement de PRESENT.

D'autres structures, comme l'AES, appliquent des opérations directement entre les sorties des S-boxes lors de l'étage linéaire.

La DFA comme nous l'avons introduite est dans ce cas plus difficile à mener, bien que perturber l'exécution d'une S-box lors de l'avant-dernière couche de substitution affecte dans ce cas plusieurs S-boxes de la dernière couche de substitution, et l'attaquant peut parfois de cette façon éliminer certaines valeurs possibles pour les différences en entrée en fonction des différences obtenues en sortie.

La complexité de l'attaque est donc beaucoup plus élevée sur ce type de structure, et il est souvent préférable d'appliquer une autre attaque avec des contraintes plus fortes sur les fautes requises si la technique d'injection utilisée le permet.

3.3.2 — Estimation du nombre de fautes requises

Comme nous avons pu le constater, la DFA que nous avons proposée est applicable à la plupart des structures SPN, et est particulièrement efficace sur les structures facilitant le parallélisme par tranches de bits sur la couche de substitution.

Il reste néanmoins un point que nous n'avons pas abordé jusque là : le nombre de fautes requises pour retrouver entièrement la clé secrète qui, comme nous l'avons constaté, dépend à la fois de la structure du chiffrement SPN, des ensembles de candidats obtenus pour chaque mot de la clé secrète et du type de faute que l'attaquant est capable d'obtenir.

L'intersection de deux ensembles obtenus à partir de deux différentielles ne réduit pas systématiquement à un seul candidat possible la partie ciblée de la clé secrète.

Un attaquant peut néanmoins exploiter la table de distribution des différences avec la proposition 1, que nous avons énoncée et démontrée afin de trouver deux différences a_1 et a_2 en entrée d'une S-box qui permettent de réduire le nombre d'entrées à un unique candidat quelles que soit les valeurs des différences b_1 et b_2 en sortie.

Proposition 1. *Soit $\mathcal{S}$ une S-box de n bits. Soit (a_1, b_1) et (a_2, b_2) deux différentielles avec $a_1 \neq a_2$ telles que le système de deux équations*

$$\begin{cases} \mathcal{S}(x \oplus a_1) \oplus \mathcal{S}(x) = b_1 & (3.1) \\ \mathcal{S}(x \oplus a_2) \oplus \mathcal{S}(x) = b_2 & (3.2) \end{cases}$$

a au moins deux solutions. Alors, chacune des trois équations (3.1)*,* (3.2) *et*

$$\mathcal{S}(x \oplus a_1 \oplus a_2) \oplus \mathcal{S}(x) = b_1 \oplus b_2$$

a au moins quatre solutions.

Preuve de la proposition 1. Soit $\mathcal{D}(a, b)$ l'ensemble des solutions de l'équation

$$\mathcal{S}(x \oplus a) \oplus \mathcal{S}(x) = b.$$

Soit (a_1, b_1) et (a_2, b_2) deux différentielles avec $a_1 \neq a_2$ telles que

$$\#\mathcal{D}(a_1, b_1) \cap \mathcal{D}(a_2, b_2) \geq 2.$$

Il est clair que n'importe quel élément x dans $\mathcal{D}(a_1, b_1) \cap \mathcal{D}(a_2, b_2)$ est solution de

$$\mathcal{S}(x \oplus a_2) \oplus \mathcal{S}(x \oplus a_1) = b_1 \oplus b_2,$$

c'est-à-dire, $x \oplus a_1 \in \mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2)$ et $x \oplus a_2 \in \mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2)$.

Soit $\{x, x \oplus a_4\} \subseteq \mathcal{D}(a_1, b_1) \cap \mathcal{D}(a_2, b_2)$ pour un certain $a_4 \neq 0$. Alors

$$\{x, x \oplus a_1, x \oplus a_4, x \oplus a_1 \oplus a_4\} \subseteq \mathcal{D}(a_1, b_1),$$

$$\{x, x \oplus a_2, x \oplus a_4, x \oplus a_2 \oplus a_4\} \subseteq \mathcal{D}(a_2, b_2),$$

$$\{x \oplus a_1, x \oplus a_2, x \oplus a_1 \oplus a_4, x \oplus a_2 \oplus a_4\} \subseteq \mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2).$$

Puisque $a_1 \neq a_2$, nous avons juste à montrer que si $a_4 = a_1$, $a_4 = a_2$ ou $a_4 = a_1 \oplus a_2$ alors $\mathcal{D}(a_1, b_1)$, $\mathcal{D}(a_2, b_2)$ et $\mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2)$ ont chacun au moins 4 éléments.

Si $a_4 = a_1$ alors $x \oplus a_1 \in \mathcal{D}(a_2, b_2)$ implique

$$\mathcal{S}(x \oplus a_1 \oplus a_2) \oplus \mathcal{S}(x \oplus a_1) = b_2 = \mathcal{S}(x \oplus a_2) \oplus \mathcal{S}(x)$$

ce qui implique que

$$\mathcal{S}(x \oplus a_1) \oplus \mathcal{S}(x) = \mathcal{S}(x \oplus a_2 \oplus a_1) \oplus \mathcal{S}(x \oplus a_2).$$

Ainsi $x \oplus a_2 \in \mathcal{D}(a_1, b_1)$ et $\mathcal{D}(a_1, b_1)$, $\mathcal{D}(a_2, b_2)$ et $\mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2)$ contiennent $\{x, x \oplus a_1, x \oplus a_2, x \oplus a_1 \oplus a_2\}$. Le raisonnement est identique si $a_4 = a_2$.

Maintenant, $a_4 = a_1 \oplus a_2$ implique que $x \oplus a_2 \oplus a_4 = x \oplus a_1$ appartient à $\mathcal{D}(a_2, b_2)$, c'est-à-dire, $x \oplus a_1 \in \mathcal{D}(a_1, b_1) \cap \mathcal{D}(a_2, b_2)$.

En conclusion, $x \oplus a_1$, $x \oplus a_2$, x et $x \oplus a_1 \oplus a_2$ appartiennent tous à $\mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2)$ et $\#\mathcal{D}(a_1 \oplus a_2, b_1 \oplus b_2) \geq 4$. □

En d'autres termes, si un attaquant trouve deux différentielles (a_1, b_1) et (a_2, b_2) telles que l'une des trois entrées (a_1, b_1), (a_2, b_2) ou $(a_1 \oplus a_2, b_1 \oplus b_2)$ est égale à 2 dans la table de distribution des différences, alors la valeur de x est unique.

Il est à noter que si une des trois équations n'a pas de solution, alors le système formé par les deux équations (3.1), (3.2) n'a pas de solution.

Un attaquant peut alors appliquer la proposition 1 à la S-box inverse du chiffrement SPN, dont la table de distribution des différences est la transposée de celle de la S-box, afin de retrouver la partie ciblée de la clé secrète comme nous l'avons montré dans la section 3.3.1. Il est de plus généralement assez facile de trouver un couple de différentielles qui vérifie cette proposition. Le tableau 3.2 donne des exemples de couples exploitables pour quelques S-boxes inverses.

Tableau 3.2.: Couples de différentielles (Δ_{in},Δ_{out}) exploitables.

S-box^{-1}	Couple exploitable
Fantomas	(0x01,a_1) - (0x80,a_2)
PRIDE	(0x1,a_1) - (0x8,a_2)
Robin	(0x01,a_1) - (0x40,a_2)
Scream	(0x01,a_1) - (0x02,a_2)
iScream	(0x02,a_1) - (0x80,a_2)

Pour ces chiffrements SPN, les différences Δ_{in} en entrée de la S-box inverse données à la table 3.2 ont un poids de Hamming égal à un, et peuvent donc être obtenues à partir d'une inversion des bits d'un mot en sortie de l'étage linéaire.

Ainsi, deux différentielles seulement sont nécessaires à un attaquant pour retrouver la valeur en sortie de la S-box, et le nombre total de fautes requises ne dépend que de l'implémentation ciblée et du type de faute que l'attaquant est en mesure de produire.

3.3.2.1 – Modèle de faute idéal

Le modèle de faute idéal pour un attaquant est une inversion de tous les bits du mot ciblé, afin que chaque S-box sur laquelle se propage la faute soit active.

Pour les chiffrements SPN facilitant le parallélisme par tranches de bits, pour lesquels les deux mots ciblés se diffusent sur toutes les S-boxes, deux fautes sont donc suffisantes pour retrouver entièrement la dernière clé utilisée.

Cela signifie que deux fautes théoriques seulement sont suffisantes dans le cas des LS-Designs et des structures CUBE, et quatre fautes dans le cas de PRIDE.

Pour les autres structures SPN, il faut donc au maximum $2e$ fautes pour retrouver entièrement la dernière clé utilisée, où e est le nombre de fautes nécessaires afin que chaque S-box soit affectée au moins une fois par une faute.

3.3.2.2 – Modèle de faute aléatoire

Comme nous l'avons mentionné, l'effet d'une faute dépend de la technique d'injection utilisée, mais qu'il s'agisse d'un saut d'instruction, d'une mise à zéro, d'une mise à un ou d'une inversion d'un ou plusieurs bits, le type de faute qu'un attaquant est capable d'obtenir sur un mot quelconque des données dépend également de la valeur des données à corrompre [LASHERMES et al. 2012].

La probabilité d'apparition de chaque faute sur un mot ciblé varie donc en fonction des données, et on peut considérer qu'un attaquant est capable d'inverser chaque bit avec une probabilité de 50%, que l'on désigne par modèle de faute aléatoire, à partir de messages clairs quelconques (il est à noter que changer de message clair est parfois nécessaire, en fonction de l'application ciblée, et demande alors une exécution supplémentaire sans perturbation afin de pouvoir faire la comparaison avec les autres exécutions corrompues sur ce même message).

Bien que la probabilité d'inverser chaque bit ne soit jamais exactement de 50%, ce modèle de faute aléatoire est assez représentatif de ce qu'un attaquant est capable d'obtenir en pratique avec la plupart des techniques d'injection.

L'attaque consiste alors à cibler deux mots qui permettent d'obtenir deux différentielles vérifiant la proposition 1.

Dans le cas où les deux mots ciblés se diffusent sur toutes les S-boxes, le nombre de candidats restant en moyenne pour la dernière clé utilisée va dépendre du nombre A_1 (resp. A_2) de candidats restant en moyenne sur un mot de ω bits de la clé avec une faute sur le premier mot (resp. le second) activant la S-box correspondante.

Ainsi, après m_1 fautes sur le premier mot et m_2 fautes sur le second, le nombre moyen N de candidats restants pour la dernière clé utilisée est :

$$N = \left(\frac{2^\omega}{2^{m_1+m_2}} + \sum_{i=1}^{m_1} \frac{A_1}{2^{i+m_2}} + \sum_{i=1}^{m_2} \frac{A_2}{2^{i+m_1}} + \left(\sum_{i=1}^{m_1} \frac{1}{2^i} \right) \left(\sum_{i=1}^{m_2} \frac{1}{2^i} \right) \right)^c$$

où c est le nombre de S-boxes utilisées.

En effet, lorsque m fautes ont été injectées, la probabilité de ne pas avoir activé une S-box est égale à $1/2^m$ (puisqu'elle est égale à $1/2$ avec une faute), et la probabilité de l'avoir activée au moins une fois est égale à $\sum_{i=1}^{m} 1/2^i = 1 - 1/2^m = (2^m - 1)/2^m$.

De plus, si la S-box n'a été activée avec aucune faute alors le nombre de candidats restant sur le mot de ω bits correspondant est 2^ω. Dans le cas où elle a été activée à partir de fautes sur un seul des deux mots ciblés alors l'attaquant obtient A_1 ou A_2 candidats. Finalement, si elle a été activée à partir de fautes sur chacun des mots l'attaquant retrouve la valeur correcte.

On en déduit donc que

$$\begin{aligned} N &= \left(\frac{2^\omega + A_1(2^{m_1} - 1) + A_2(2^{m_2} - 1) + (2^{m_1} - 1)(2^{m_2} - 1)}{2^{m_1+m_2}} \right)^c \\ &= \left(\frac{2^\omega - A_1 - A_2 + 1}{2^{m_1+m_2}} + \frac{A_1 - 1}{2^{m_2}} + \frac{A_2 - 1}{2^{m_1}} + 1 \right)^c . \end{aligned}$$

Par exemple, dans le cas de PRIDE où $\omega = 4$, $c = 16$ et $A_1 = A_2 = 4$, l'équation devient

$$N = \left(\frac{9}{2^{m_1+m_2}} + \frac{3}{2^{m_2}} + \frac{3}{2^{m_1}} + 1 \right)^{16} .$$

Ainsi, une dizaine de fautes sur chaque mot sont suffisantes pour retrouver une des clés secrètes (K_0 ou K_1) utilisées par PRIDE.

Pour les autres structures SPN, il faut également prendre en compte le nombre de fautes nécessaires pour affecter toutes les S-boxes, et le nombre de fois où chaque S-box est affectée par ces fautes.

Enfin, dans le cas d'une implémentation logicielle qui manipule un mot plus grand que la partie P à corrompre, il est à noter que l'attaquant est généralement en mesure de savoir à partir des différences en sortie de chaque S-box si la faute obtenue a bien affecté un seul bit en entrée de chacune, c'est-à-dire qu'il est capable de n'exploiter que les fautes ayant corrompu la partie P du mot manipulé.

Il peut également choisir d'exploiter toutes les fautes si le mot manipulé se diffuse sur peu de bits en entrée des S-boxes, par exemple s'il se diffuse sur deux bits seulement, la différence en entrée ne peut prendre que trois valeurs non nulles, ce qui est parfois une information suffisante pour mener l'attaque.

3.3.3 Simulation avec un modèle de faute idéal

Afin de valider que le chemin d'attaque que nous proposons permet bien de retrouver la clé secrète d'un chiffrement SPN, nous détaillons dans un premier temps une simulation de la DFA sur PRIDE avec un modèle de faute idéal.

Nous supposons donc qu'un dispositif exécute PRIDE avec une clé secrète $\mathrm{K} = \mathrm{K}_0||\mathrm{K}_1$ où K_0 = 0xefcdab8967452301 et K_1 = 0x0123456789abcdef.

Nous supposons également qu'un attaquant arrive à inverser successivement tous les 16 premiers et les 16 derniers bits de W_{19} (voir figure 3.4), puis de W_{18}.

L'attaquant obtient alors les chiffrés suivants à partir de 5 exécutions du même message clair 0xfedcba9876543210 :

- 0xc40f2551f39c63a9 la valeur correcte du chiffré,
- 0xe7f325510dc3b7a8 (resp. 0xc40fdaaec89376f7) à partir d'une inversion des 16 premiers (resp. derniers) bits de W_{19},
- 0x2857589433cbdead (resp. 0x461720d9729c1956) à partir d'une inversion des 16 premiers (resp. derniers) bits de W_{18}.

Il peut ainsi calculer les différentielles sur la dernière couche de substitution :

- $(\Delta X_{20}, \Delta Y_{20})_1$ = (0x8888888888888888, 0x33a323a88a8aaa23),
- $(\Delta X_{20}, \Delta Y_{20})_2$ = (0x1111111111111111, 0x4467656745457776).

À partir de la première différentielle, il obtient un ensemble de candidats pour chaque quartet de $\mathcal{P}^{-1}(\mathrm{C}) \oplus \mathrm{K}_0$ où C est la valeur correcte du chiffré. Il peut alors calculer l'ensemble de candidats pour chaque quartet de K_0, noté Q_i pour $0 \leq i \leq 15$, à partir de $\mathcal{P}^{-1}(\mathrm{C})$ = 0xab720c373416ba8d. Le tableau 3.3 donne les ensembles de candidats ainsi calculés en notation hexadécimale.

Tableau 3.3.: Candidats obtenus à partir de $(\Delta X_{20}, \Delta Y_{20})_1$ en notation hexadécimale.

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
5	4	4	5	0	0	0	1	5	5	4	5	0	1	0	1
6	7	6	6	2	3	2	2	6	7	7	7	2	3	2	2
d	c	c	d	8	8	8	9	d	d	c	d	8	9	8	9
e	f	e	e	a	b	a	a	e	f	f	f	a	b	a	a

Ensuite, à partir de la seconde différentielle, l'attaquant obtient un autre ensemble de candidats pour chaque quartet de K_0 donnés dans le tableau 3.4.

Tableau 3.4.: Candidats obtenus à partir de $(\Delta X_{20}, \Delta Y_{20})_2$ en notation hexadécimale.

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
a	a	a	a	a	a	8	8	2	2	0	0	2	2	0	0
b	b	b	b	b	b	9	9	3	3	1	1	3	3	1	1
e	e	c	c	c	e	e	e	6	6	4	4	4	4	6	6
f	f	d	d	d	f	f	f	7	7	5	5	5	5	7	7

L'attaquant retrouve ainsi la valeur de chaque quartet de K_0 par intersection des deux ensembles obtenus.

Ensuite, à partir de la connaissance de K_0 et des valeurs corrompues du chiffré obtenues à partir des fautes sur W_{18}, il peut calculer la valeur des différentielles sur l'avant-dernière couche de substitution :

- $(\Delta X_{19}, \Delta Y_{19})_1$ = (`0x8888888888888888`, `0x23a2288338832828`),
- $(\Delta X_{19}, \Delta Y_{19})_2$ = (`0x1111111111111111`, `0x7777456474776476`).

En exploitant la première différentielle, il obtient un ensemble de candidats pour chaque quartet Q_i de M-layer^{-1}(S-layer^{-1}($\mathcal{P}^{-1}$(C) $\oplus$ K_0) $\oplus$ $\mathcal{P}^{-1}(f_{20}(K_1))$) pour $0 \leq i \leq 15$. Le tableau 3.5 donne les candidats obtenus en notation hexadécimale.

Tableau 3.5.: Candidats obtenus à partir de $(\Delta X_{19}, \Delta Y_{19})_1$ en notation hexadécimale.

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
0	4	1	0	0	5	5	4	4	5	5	4	0	5	0	5
2	7	3	2	2	6	6	7	7	6	6	7	2	6	2	6
8	c	9	8	8	d	d	c	c	d	d	c	8	d	8	d
a	f	b	a	a	e	e	f	f	e	e	f	a	e	a	e

Ensuite, à partir de la seconde différentielle, il obtient d'autres candidats qui sont donnés dans le tableau 3.6 en notation hexadécimale.

Tableau 3.6.: Candidats obtenus à partir de $(\Delta X_{19}, \Delta Y_{19})_2$ en notation hexadécimale.

Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
8	8	8	8	0	2	a	0	8	0	8	8	a	0	8	a
9	9	9	9	1	3	b	1	9	1	9	9	b	1	9	b
e	e	e	e	4	6	c	4	e	4	e	e	c	4	e	c
f	f	f	f	5	7	d	5	f	5	f	f	d	5	f	d

Par intersection des ensembles obtenus, l'attaquant retrouve

$$\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1))) = \text{0x8f9806d4f5efa58d}.$$

Puis il calcule

$$\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1)) = \text{0x24c39cc978f41dd4}$$

et à partir de $\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) = \text{0x11c3a9c65f5f772b}$, il obtient

$$\mathcal{P}^{-1}(f_{20}(\text{K}_1)) = \text{0x3500350f27ab6aff}.$$

Finalement, il déduit que $f_{20}(\text{K}_1) = \text{0x0137454b89ffcd53}$, et retrouve K_1, et donc la clé complète, à partir de l'algorithme de cadencement de clés de PRIDE.

Ainsi, quatre fautes théoriques seulement ont bien suffi à retrouver entièrement la clé secrète manipulée par PRIDE.

3.3.4 Exemples de DFA conduite en laboratoire

Afin de montrer la faisabilité de notre chemin d'attaque en pratique, nous allons détailler les expérimentations de la DFA que nous avons menées en laboratoire sur deux implémentations logicielles de PRIDE, et sur une implémentation matérielle de SCREAM. Nous avons choisi le schéma d'authentification SCREAM, qui a été soumis à la compétition Caesar [Bernstein 2014] et qui utilise le LS-Design Scream [Grosso et al. 2014], car son implémentation matérielle de référence est très efficace.

3.3.4.1 Exemple de DFA conduite sur un microcontrôleur

Afin de tester la faisabilité de l'attaque sur PRIDE, nous avons implémenté et exécuté ce dernier sur un micro-contrôleur ARM Cortex-M3 puisqu'il s'agit d'une architecture 32 bits assez représentative des dispositifs utilisés pour l'IoT.

Nous avons dans un premier temps utilisé l'implémentation 8 bits de référence de PRIDE, que nous avons implémentée en langage C, dont le code source est donné en annexe A.4.3, et qui manipule les lignes de l'état interne indépendamment les unes des autres lors de l'exécution de l'étage linéaire.

Nous avons ensuite proposé et utilisé une implémentation 32 bits de PRIDE afin d'exploiter entièrement les mots 32 bits de l'architecture, dont le code source est donné en annexe A.4.1, qui manipule cette fois deux lignes de l'état interne simultanément lors de l'exécution de l'étage linéaire.

3.3.4.1.a - DFA conduite sur l'implémentation 8 bits de PRIDE

Tout d'abord, nous avons utilisé l'analyse des rayonnements électromagnétiques de PRIDE décrite à la figure 2.8 afin de situer temporellement toutes les étapes du chiffrement, en particulier le dernier étage linéaire.

Nous avons ensuite exécuté PRIDE avec la clé secrète K_0 = 0xa371b246f90cf582, K_1 = 0xe417d148e239ca5d et le message clair 0xe8d3157f246e80cb en entrée, et nous avons utilisé le banc d'injection d'impulsions de tension décrit à la figure 3.1 afin de perturber son exécution.

Le valeur correcte du chiffré est C = 0x0b735baaf63aac9e et nous avons obtenu 13 fautes (resp. 19 fautes) exploitables sur le dernier (resp. avant-dernier) étage linéaire. Afin de vérifier qu'une faute est exploitable, nous avons calculé la différence en sortie du dernier étage linéaire, ou l'avant-dernier une fois K_0 retrouvé, et nous avons vérifié que toutes les différences en sortie des S-boxes actives pouvaient être obtenues à partir d'une même différence égale à 2^i avec $0 \leq i \leq 3$ en entrée.

Lorsqu'un chiffré corrompu vérifie la propriété précédente, cela signifie qu'il a été obtenu à partir d'une faute n'ayant affecté qu'une seule ligne de l'état interne.

Le tableau 3.7 fournit les chiffrés corrompus exploitables que nous avons obtenus à partir de fautes sur le dernier étage linéaire, ainsi que notre connaissance sur les différences autour de la dernière couche de substitution pour chacun, c'est-à-dire la valeur de la différence ΔY_{20} en sortie et la valeur δ_{in} de chaque quartet non nul de la différence ΔX_{20} en entrée (identique pour tous les quartets).

Tableau 3.7.: Chiffrés corrompus obtenus à partir de fautes sur le dernier étage linéaire lors de l'exécution de l'implémentation 8 bits de PRIDE.

Chiffrés corrompus	ΔY_{20}	δ_{in}
0x83735baa7632ac9e	0xa000800000002000	0x8
0x03f3d30276128c9e	0x6010c000c0606000	0x4
0xcb339beaf67aacde	0xcc0000000f000000	0x2
0xc47397aaf23aa09e	0xcc00df8800000000	0x2
0xcb329beaf67aacde	0xcc0000000f000008	0x2
0xadd5df8ad21c88b8	0xc0b00f8080f00bb0	0x2
0x0b739f2276b22c96	0x7400040060007000	0x1
0x0b730e41f793bcb4	0x0405040664707056	0x1
0x0b73d3a276322496	0x7000500000007000	0x1
0x0b73c23377b33486	0x7005500660057006	0x1
0x0b73a40f759b34be	0x7445546660700406	0x1
0x0b737b88f61aacbc	0x0040000000700050	0x1
0x0b73eb1176933ca4	0x7045000060757056	0x1

Nous pouvons alors rassembler l'information que nous avons obtenue sous la forme de quatre différentielles sur la dernière couche de substitution :

- $(\Delta X_{20}, \Delta Y_{20})_1$ = (0x1111111111111111, 0x7445546664757456),
- $(\Delta X_{20}, \Delta Y_{20})_2$ = (0x2220222222200222, 0xccb0df888ff00bb8),
- $(\Delta X_{20}, \Delta Y_{20})_3$ = (0x4040400040404000, 0x6010c000c0606000),
- $(\Delta X_{20}, \Delta Y_{20})_4$ = (0x8000800000008000, 0xa000800000002000).

Le tableau 3.8 donne les candidats, en notation hexadécimale, que nous avons obtenus sur chaque quartet Q_i de K_0 pour $0 \leq i \leq 15$ à partir de ces différentielles et de la connaissance de $\mathcal{P}^{-1}(C)$ = 0x3636d3ec58eb71f8, en fonction de la valeur δ_{in} en entrée de chaque S-box. Il donne également l'intersection, noté $\cap$, des ensembles de candidats obtenus sur chaque quartet. Le symbole $*$ désigne l'ensemble des 16 valeurs possibles, c'est-à-dire que la S-box correspondante n'a pas été activée.

Tableau 3.8.: Connaissance à partir du tableau 3.7 sur chaque quartet Q_i pour $0 \leq i \leq 15$ en notation hexadécimale, où $*$ désigne l'ensemble des 16 valeurs possibles.

δ_{in}	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
1	a b c d	2 3 6 7	2 3 6 7	0 1 4 5	a b e f	2 3 6 7	2 3 4 5	0 1 6 7	8 9 e f	8 9 c d	0 1 6 7	8 9 c d	8 9 e f	0 1 4 5	8 9 c d	2 3 4 5
2	6 a	3 f	4 7 c f	*	6 b	2 d	4 6 c e	4 6 c e	5 7 d f	6 9	0 f	*	*	5 6 d e	0 3 8 b	0 2 8 a
4	a b c d	*	2 3 6 7	*	7 b	*	*	*	3 f	*	0 1 6 7	*	8 9 e f	*	*	*
8	0 2 8 a	*	*	*	0 3 8 b	*	*	*	*	*	*	*	5 7 d f	*	*	*
$\cap$	a	3	7	0 1 4 5	b	2	4	6	f	9	0	8 9 c d	f	5	8	2

Les 13 chiffrés corrompus du tableau 3.7 nous ont donc permis de réduire les valeurs possibles pour K_0 à seulement 16 candidats, et nous avons ensuite exploité certaines fautes obtenues sur l'avant-dernier étage linéaire pour retrouver K_0.

En effet, à partir du message corrompu 0xc42ec0dbb65e18db issu d'une faute sur l'avant-dernier étage linéaire, nous avons retrouvé la valeur du 11-ième quartet et ainsi éliminé 12 candidats pour K_0 puisque les différences en sortie de l'avant-dernière couche de substitution n'étaient pas possibles dans ce cas, et les 4 candidats restant étaient donc tels que $Q_3 \in \{0x0,0x1,0x4,0x5\}$.

Nous avons alors de façon similaire éliminé 3 candidats à partir du message corrompu 0x3165d7eea5f5f4dc, et nous avons ainsi retrouvé la valeur correcte de K_0.

Une fois la valeur de K_0 obtenue, nous avons pu calculer les différences en sortie de l'avant-dernière couche de substitution. Le tableau 3.9 donne les chiffrés corrompus et les différentielles obtenus à partir de fautes sur l'avant-dernier étage linéaire, avec les mêmes notations que précédemment.

Tableau 3.9.: Chiffrés corrompus obtenus à partir de fautes sur l'avant-dernier étage linéaire lors de l'exécution de l'implémentation 8 bits de PRIDE.

Chiffrés corrompus	ΔY_{19}	δ_{in}
0xb3035fae64aabc8e	0x0000000003208080	0x8
0x3f6713aecea2948e	0x8300000200000000	0x8
0x1bdad38aff8aa4ae	0x0000000000022800a	0x8
0x3165d7eea5f5f4dc	0x03a88a8200000000	0x8
0x16fdd78aea9ca890	0x000000000000a066	0x4
0x077fdeba72a7d9da	0xa60c000100000000	0x4
0x12f193ceee10a898	0x00000000c0000166	0x4
0x92f9c2927701dcdc	0xa60c00010000a066	0x4
0x81791f6e017bd89e	0x000000000088eb00	0x2
0x827873a04d02ac8c	0x00000000800000fc	0x2
0xb05e37e04c63acec	0x000000008b080bfc	0x2
0x411737ca9638aeba	0x00000dd000000000	0x2
0x08bf2c2551e6f6bf	0x0bedf0d000000000	0x2
0x303fbc2c4076debe	0xebe000d000000000	0x2
0xd4bfe13bb63fa8e8	0x0000000064006077	0x1
0x91f0e1b0f632ada9	0x0000000004400077	0x1
0xc42ec0dbb65e18db	0x0000000064446407	0x1
0x4cbfd8ca365e88d2	0x0000000064446400	0x1
0x856cc59ff218d813	0x0000000004006407	0x1

Là encore, nous pouvons rassembler l'information que nous avons obtenue sous la forme de quatre différentielles sur la dernière couche de substitution :

- $(\Delta X_{19}, \Delta Y_{19})_1 = (\texttt{0x0000000011111111}, \texttt{0x0000000064446477})$,
- $(\Delta X_{19}, \Delta Y_{19})_2 = (\texttt{0x2222222022222222}, \texttt{0xebedfdd08b88ebfc})$,
- $(\Delta X_{19}, \Delta Y_{19})_3 = (\texttt{0x4404000440004444}, \texttt{0xa60c0001c000a166})$,
- $(\Delta X_{19}, \Delta Y_{19})_4 = (\texttt{0x8888888808888088}, \texttt{0x83a88a820322808a})$.

Le tableau 3.10 donne les candidats, en notation hexadécimale, que nous avons obtenus sur chaque quartet Q_i de

$$\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(\text{C}) \oplus \text{K}_0) \oplus \mathcal{P}^{-1}(f_{20}(\text{K}_1))) = \text{0xdf36eb60a400d4e9}$$

pour $0 \leq i \leq 15$ en fonction de la valeur δ_{in} en entrée de chaque S-box, ainsi que l'intersection des ensembles obtenus avec les mêmes notations que précédemment.

Tableau 3.10.: Connaissance à partir du tableau 3.9 sur chaque quartet Q_i pour $0 \leq i \leq 15$ en notation hexadécimale, où $*$ désigne l'ensemble des 16 valeurs possibles.

δ_{in}	Q_0	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_{15}
1	*	*	*	*	*	*	*	*	a b c d	0 1 4 5	0 1 4 5	0 1 4 5	a b c d	0 1 4 5	8 9 e f	8 9 e f
2	3 d	4 7 c f	3 d	6 b	1 e	6 b	6 b	*	0 2 8 a	4 7 c f	0 2 8 a	0 2 8 a	3 d	4 7 c f	1 e	5 9
4	7 d	8 9 e f	*	6 a	*	*	*	0 1 4 5	6 a	*	*	*	7 d	0 1 4 9	8 9 e f	8 9 e f
8	5 6 d e	4 7 c f	1 3 9 b	5 6 d e	5 6 d e	1 3 9 b	5 6 d e	0 2 8 a	*	4 7 c f	0 2 8 a	0 2 8 a	5 6 d e	*	5 0 d e	1 3 9 b
∩	d	f	3	6	e	b	6	0	a	4	0	0	d	4	e	9

Les 19 chiffrés corrompus du tableau 3.9 nous ont donc permis de retrouver la valeur de M-layer^{-1}(S-layer^{-1}($\mathcal{P}^{-1}$(C) $\oplus$ K_0) $\oplus$ $\mathcal{P}^{-1}(f_{20}(K_1))$) et d'obtenir ainsi K_1, et donc la clé complète, à partir de C et K_0.

Il est à noter que nous avons eu besoin de plus de fautes pour n'obtenir qu'un seul candidat pour la clé secrète entière comparé à l'analyse que nous avons faite à la section 3.3.2.2 sur le nombre de fautes requises pour PRIDE.

Cela vient certainement du fait que nous avons utilisé le même message clair pour toutes les exécutions, ce qui implique que le modèle de faute n'était pas complètement aléatoire, c'est-à-dire que certaines fautes avaient une probabilité plus importante d'apparaître que d'autres. La connaissance du modèle de faute n'a cependant pas été nécessaire pour mener l'attaque.

3.3.4.1.b - DFA conduite sur l'implémentation 32 bits de PRIDE

L'implémentation 32 bits que nous avons proposée manipule les lignes de l'état interne deux par deux. De ce fait, perturber l'exécution de l'étage linéaire implique que la valeur de toutes les différences en entrée des S-boxes doivent appartenir à un seul des deux ensembles $E_1 = \{0x0, 0x1, 0x2, 0x3\}$ ou $E_2 = \{0x0, 0x4, 0x8, 0xc\}$.

Afin de mener l'attaque, nous avons une fois de plus utilisé une analyse des rayonnements électromagnétiques pour situer temporellement le dernier étage linéaire.

Nous avons ensuite exécuté PRIDE avec la clé secrète K_0 = 0xf3f721cb1c882658, K_1 = 0xe417d148e239ca5d et le message clair 0x0132546798badcfe en entrée, et nous avons perturbé son exécution avec le banc d'injection d'impulsions de tension décrit à la figure 3.1.

Le valeur correcte du chiffré était 0x9aecb37ea45a6c89 et nous avons obtenu 12 (resp. 13) fautes exploitables sur le dernier (resp. l'avant-dernier) étage linéaire.

Afin de vérifier qu'une faute est exploitable, nous nous sommes cette fois assuré que toutes les différences en sortie des S-boxes actives pouvaient provenir de différences en entrée toutes contenues dans un des deux ensembles E_1 ou E_2.

Le tableau 3.11 donne les chiffrés corrompus et les différentielles obtenus à partir de fautes sur le dernier étage linéaire, avec les mêmes notations que le paragraphe précédent, et avec θ, β, γ et σ qui désignent respectivement les ensembles {0x2,0x3}, {0x4,0x8}, {0x4,0xc} et {0x8,0xc} des valeurs possibles pour δ_{in}.

Tableau 3.11.: Chiffrés corrompus obtenus à partir de fautes sur le dernier étage linéaire lors de l'exécution de l'implémentation 32 bits de PRIDE.

Chiffrés corrompus	ΔY_{20}	ΔX_{20}
0x1aad3b972c92ec09	0xf00060007e40600c	0xθ0001000$1\theta$10100θ
0x7b4c93dea55a6d89	0x88c0000bc0c00000	0x$\theta\theta\theta$0000$\theta\theta$0θ00000
0x1b6c733e255aadc9	0xf500000b85000000	0xθ100000$\theta\theta$1000000
0x71ecd27ee55a6d89	0x8ec0808f00000000	0x$\theta\theta\theta$0θ0$\theta\theta$00000000
0x9aecb324a4426cdb	0x0000000005076050	0x0000000001011010
0x9a57b33fa4626cf1	0x0000000085bbb08c	0x0000000θ1$\theta\theta\theta$0$\theta\theta$
0x9a57b365a4606cb9	0x0000000080bfe0ec	0x0000000θ0$\theta\theta\theta$0$\theta\theta$
0x77aa24313111ed8c	0xf8868e4f0e006de7	0x$\theta\theta$1$\theta\theta$1θ0θ001$\theta\theta$1
0x9ae8b37ac15a6989	0x0220030300000c00	0x0$\sigma\sigma$00σ0σ00000γ00
0x8aecb27e415abc89	0x3329020600000000	0x$\sigma\sigma\sigma\gamma$0σ0400000000
0xa3e692ed909ee688	0x10ea921c620482c5	0x40c$\beta\gamma\sigma$4γ4σ0c8$\sigma\gamma$c
0x05ecb27e565a7289	0xa22b99bc00000000	0x$\beta\sigma\sigma$c$\gamma\gamma$cγ00000000

Il est à noter que sur 2 000 impulsions de tension, une fois les paramètres du générateur d'impulsions de tension réglés, nous n'avons reçu aucun chiffré dans 1 219 des cas, et nous avons obtenu 247 chiffrés corrompus dont 13 exploitables. Les fautes non exploitables que nous avons obtenues sont dues à un dysfonctionnement de l'UART à cause du rayonnement électromagnétique.

Le tableau 3.12 donne les différentielles obtenues sur $\mathcal{S}^{-1}$, et les candidats correspondants en notation hexadécimale, qui nous ont permis de retrouver autant d'information sur chaque quartet Q_i de K_0 pour $0 \leq i \leq 15$ qu'avec toutes les fautes, à partir de la connaissance de $\mathcal{P}^{-1}(C)$ = 0xe17c93c49ec6fc61. Il donne également l'intersection, noté $\cap$, des ensembles de candidats obtenus sur chaque quartet.

Tableau 3.12.: Connaissance à partir du tableau 3.11 sur chaque quartet Q_i pour $0 \leq i \leq 15$ en notation hexadécimale.

	Différentielles obtenues sur $\mathcal{S}^{-1}$ et candidats correspondants	∩
Q_0	(f,θ) → {0,1,e,f} et (a,β) → {3,5,7,9,d,f}	f
Q_1	(5,1) → {2,3,6,7}, (8,θ) → {0,1,2,3,8,9,a,b} et (2,σ) → {1,3,4,6,9,b}	3
Q_2	(e,c) → {1,f} et (2,σ) → {0,2,5,7,d,f}	f
Q_3	(6,1) → {0,1,6,7} et (b,c) → {7,c}	7
Q_4	(6,1) → {2,3,4,5}, (8,θ) → {0,1,2,3,8,9,a,b} et (9,γ) → {2,4,b,d	2
Q_5	(e,θ) → {0,1,e,f} et (2,σ) → {1,3,4,6,e,b}	1
Q_6	(1,4) → {8,9,c,d} et (b,c) → {7,c}	c
Q_7	(f,θ) → {4,5,a,b} et (c,γ) → {2,7,b,e}	b
Q_8	(7,1) → {0,1,6,7} et (c,θ) → {0,1,2,3,8,9,a,b}	{0,1}
Q_9	(e,θ) → {2,3,c,d} et (2,σ) → {4,6,9,b,c,e}	c
Q_{10}	(4,1) → {8,9,c,d} et (b,θ) → {0,1,2,3,8,9,a,b}	{8,9}
Q_{11}	(b,θ) → {0,1,2,3,8,9,a,b} et (4,c) → {8,c}	8
Q_{12}	(6,1) → {2,3,4,5} et (8,8) → {1,2,9,a}	2
Q_{13}	(d,θ) → {6,7,a,b}, (c,γ) → {3,6,a,f} et (2,σ) → {4,6,9,b,c,e}	6
Q_{14}	(e,θ) → {4,5,a,b} et (c,γ) → {0,5,9,c}	5
Q_{15}	(c,θ) → {4,5,8,9} et (5,c) → {8,d}	8

Les 12 chiffrés corrompus du tableau 3.11 nous ont ainsi permis d'obtenir 4 candidats pour K_0, et nous avons ensuite exploité le chiffré corrompu 0xf24690de8df8cc89, issu d'une faute sur l'avant-dernier étage linéaire, afin d'éliminer 3 des candidats puisque les différences en sortie de l'avant-dernière couche de substitution n'étaient pas possibles dans ce cas, et nous avons ainsi retrouvé la valeur correcte de K_0.

Une fois la valeur de K_0 obtenue, nous avons pu calculer les différences en sortie de l'avant-dernière couche de substitution. Le tableau 3.13 donne les chiffrés corrompus et les différentielles obtenus à partir de fautes sur l'avant-dernier étage linéaire, avec les mêmes notations que précédemment.

Tableau 3.13.: Chiffrés corrompus obtenus à partir de fautes sur l'avant-dernier étage linéaire lors de l'exécution de l'implémentation 32 bits de PRIDE.

Chiffrés corrompus	ΔY_{19}	ΔX_{19}
0xf24690de8df8cc89	0xc00000b000000000	0xθ0000θ000000000
0x2df93aebf5935009	0x7807000bd8050000	0x1θ01000$\theta\theta\theta$10000
0xa9a4a34f84604dde	0x000004cd0000065c	0x000001$\theta\theta$00000011θ
0x52c367c49a9b8786	0x05077000b6d84808	0x01011000θ1$\theta\theta$1θ0θ
0x00632c247f18e99e	0x0e0bb0000d0ef000	0x0θ0$\theta\theta$0000θ0$\theta\theta$000
0xecbc98d50864ad3a	0xc0f008bbb0d00888	0xθ0θ00$\theta\theta\theta\theta$0θ00$\theta\theta\theta$
0x43b733ec34c1ec11	0x00000000300a0022	0x00000000δ00β00$\delta\delta$
0xcabdf870ee423736	0x0c8c0b123baf049e	0x0γ8γ0c4$\delta\delta$cβ40cγc
0x46eb59132610ef55	0x6f0001133aa00006	0x4400044$\delta\delta\beta\beta$00004
0x9d13b57cf2211618	0x0f036133290c0422	0x040δ44$\delta\delta\delta\gamma$0γ0c$\delta\delta$
0x1247352b2400c0ed	0x0000000009900c96	0x000000000$\gamma\gamma$00$\gamma\gamma$4
0x770a084c5528c599	0x0a8000330aa00022	0x0β8000$\delta\delta$0$\beta\beta$000$\delta\delta$
0xc80ca16eb67b9711	0x6043623a00000000	0x40cδ4$\delta\delta\beta$00000000

Le tableau 3.14 donne alors les différentielles obtenues sur $\mathcal{S}^{-1}$, et les candidats correspondants en notation hexadécimale, qui nous ont permis de retrouver autant d'information sur chaque quartet Q_i de

$$\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(C) \oplus K_0) \oplus \mathcal{P}^{-1}(f_{20}(K_1))) = \text{0x93eff0477bb31a28}$$

pour $0 \leq i \leq 15$ qu'avec toutes les fautes. Il donne également l'intersection, noté $\cap$, des ensembles de candidats obtenus sur chaque quartet.

Tableau 3.14.: Connaissance à partir du tableau 3.13 sur chaque quartet Q_i pour $0 \leq i \leq 15$ en notation hexadécimale.

	Différentielles obtenues sur $\mathcal{S}^{-1}$ et candidats correspondants	$\cap$
Q_0	(c,θ) $\rightarrow$ {4,5,8,9} et (6,4) $\rightarrow$ {8,9,e,f}	9
Q_1	(5,1) $\rightarrow$ {2,3,6,7} et (f,4) $\rightarrow$ {3,c}	3
Q_2	(f,θ) $\rightarrow$ {0,1,e,f} et (4,c) $\rightarrow$ {a,e}	e
Q_3	(7,1) $\rightarrow$ {8,9,e,f} et (c,γ) $\rightarrow$ {3,6,a,f}	f
Q_4	(7,1) $\rightarrow$ {8,9,e,f} et (b,θ) $\rightarrow$ {4,5,6,7,c,d,e,f}	{e,f}
Q_5	(b,c) $\rightarrow$ {0,b} et (1,4) $\rightarrow$ {0,1,4,5}	0
Q_6	(b,θ) $\rightarrow$ {4,5,6,7,c,d,e,f}, (1,4) $\rightarrow$ {0,1,4,5} et (3,σ) $\rightarrow$ {1,2,4,7,c,f}	4
Q_7	(2,σ) $\rightarrow$ {0,2,5,7,8,a} et (a,β) $\rightarrow$ {1,3,7,9,b,d}	7
Q_8	(b,θ) $\rightarrow$ {4,5,6,7,c,d,e,f}, (2,σ) $\rightarrow$ {0,2,5,7,8,a} et (3,σ) $\rightarrow$ {1,2,4,7,c,f}	7
Q_9	(6,1) $\rightarrow$ {a,b,c,d} et (b,c) $\rightarrow$ {0,b}	b
Q_{10}	(d,θ) $\rightarrow$ {6,7,a,b} et (9,σ) $\rightarrow$ {2,4,b,d}	b
Q_{11}	(e,θ) $\rightarrow$ {2,3,a,b} et (f,4) $\rightarrow$ {3,c}	3
Q_{12}	(4,1) $\rightarrow$ {0,1,4,5} et (f,θ) $\rightarrow$ {0,1,e,f}	{0,1}
Q_{13}	(4,c) $\rightarrow$ {a,e} et (c,γ) $\rightarrow$ {3,6,a,f}	a
Q_{14}	(9,γ) $\rightarrow$ {2,4,b,d} et (2,σ) $\rightarrow$ {0,2,5,7,8,a}	2
Q_{15}	(e,c) $\rightarrow$ {6,8} et (6,4) $\rightarrow$ {8,9,e,f}	8

Les 13 chiffrés corrompus du tableau 3.13 nous ont ainsi permis d'obtenir 4 candidats pour $\text{M-layer}^{-1}(\text{S-layer}^{-1}(\mathcal{P}^{-1}(C) \oplus K_0) \oplus \mathcal{P}^{-1}(f_{20}(K_1)))$ à partir de C et K_0 et de retrouver ainsi K_1, et donc la clé complète, en les testant tous.

De ce fait, malgré que l'implémentation 32 bits que nous avons proposée manipule deux lignes de l'état interne simultanément, nous avons montré que la DFA que nous avons introduite reste valide et permet de retrouver la clé secrète.

3.3.4.2 – Exemple de DFA conduite sur un FPGA

Afin de tester la faisabilité de l'attaque sur une implémentation matérielle, nous avons implémenté et exécuté l'implémentation VHDL de référence de SCREAM [Grosso et al. 2014] sur un FPGA Xilinx Spartan-3E 1600E fabriqué avec une technologie 90 nm et cadencé par une fréquence de 50 MHz. Avant de décrire notre expérimentation, nous allons donner les spécifications de SCREAM.

3.3.4.2.a - Le schéma d'authentification SCREAM

SCREAM est un schéma d'authentification qui utilise le LS-Design Scream avec le mode opératoire TAE (pour "Tweakable Authenticated Encryption" en anglais). Afin de chiffrer un message M, le mode opératoire TAE [LISKOV et al. 2002] découpe M en m blocs de n bits, et applique sur chaque bloc un chiffrement par blocs adaptable

$$\mathcal{E}_{\mathrm{K}} : \{0,1\}^t \times \{0,1\}^n \to \{0,1\}^n$$

en utilisant une même clé secrète K de k bits pour chiffrer chaque bloc et un bloc T de t bits, appelé le Tweak, différent sur chaque bloc produit à partir d'un nonce N (dans le cas de SCREAM, la taille recommandée pour le nonce est de 11 octets).

Le dernier bloc est complété par un 1 suivi par des 0 si nécessaire, et dans ce cas il est ajouté par un XOR à sa sortie. Enfin, il produit un "tag" à partir du XOR de tous les blocs chiffrés. La figure 3.11 illustre le mode TAE.

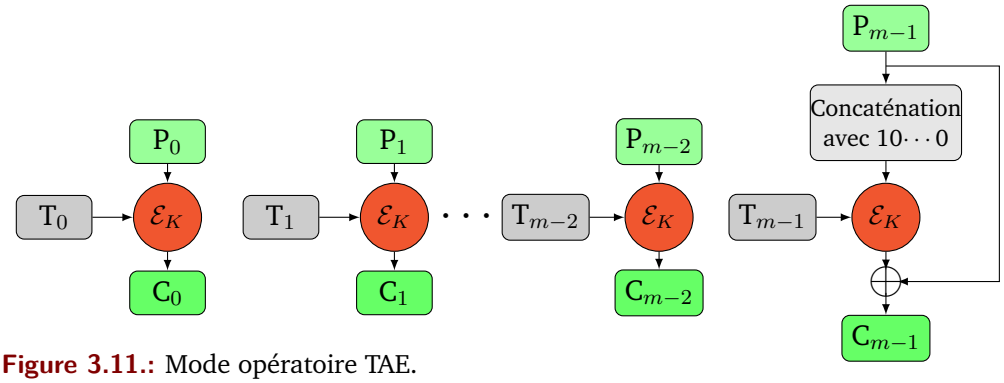


Figure 3.11.: Mode opératoire TAE.

SCREAM propose facultativement d'authentifier des blocs de données associées avec le message. Il utilise le LS-Design Scream et produit les Tweaks à partir de la valeur du nonce concaténée avec un compteur.

Scream est composé de 12 étapes de 2 tours chacune, dont les spécifications sont données dans [GROSSO et al. 2014]. Il utilise le Tweak T = $t_0||t_1$ pour générer la clé de tour utilisée à l'étape s qui est alors égale à :

$$\begin{aligned} &K \oplus (t_0||t_1) && \text{si } s = 3i, \\ &K \oplus (t_0 \oplus t_1||t_0) && \text{si } s = 3i+1, \\ &K \oplus (t_1||t_0 \oplus t_1) && \text{si } s = 3i+2. \end{aligned}$$

Le Tweak agit comme une contremesure contre certaines attaques par perturbation comme la DFA que nous avons proposée. En effet, le Tweak doit être changé à chaque exécution, et il n'est donc pas possible d'exécuter le schéma de chiffrement deux fois avec les mêmes paramètres (K,T), ce qui empêche l'attaquant d'obtenir une valeur correcte et une valeur corrompue du chiffré d'un même message clair.

Cependant, attaquer SCREAM avec un Tweak fixe est équivalent à attaquer le LS-Design Scream, ce qui nous intéresse pour valider la DFA.

3.3.4.2.b - DFA conduite sur Scream

Le modèle matériel embarqué dans le FPGA était composé du code de référence de SCREAM, d'un UART afin d'échanger des données avec l'ordinateur de contrôle et d'une FSM (pour "Finite State Machine") pour définir tous les états internes.

Les paramètres en entrée de chaque exécution étaient :

- Nonce (11 octets) : 0xe9e6f9281b86c8470ba120,
- Clé secrète : 0x2ff6963dd72462ab67d5da22c0e264ae,
- Données associées (2 blocs) : 0x5c0e6a47bc146679d2d64aca5774636797829 5340157eb9d2581bfbb14a0cb39,
- Données (3 blocs) : 0x6b36f33ff882e432861448a61183583b0df1f908593481 535b6ebbc6abfc07ae22cd50a331678301fd8535690335dcbe.

La valeur correcte du chiffré est 0xc9018ef2804f85e0de4d6519593a3e5ed83c22 bdc8b2db2229e6801071cdea6785856feac83bbe335c6bcb2f5f6d81a6.

Nous avons une fois de plus utilisé une analyse des rayonnements électromagnétiques afin de situer temporellement le dernier étage linéaire.

La figure 3.12 décrit les courbes que nous avons obtenues qui nous ont permis d'identifier toutes les étapes de l'exécution de SCREAM.

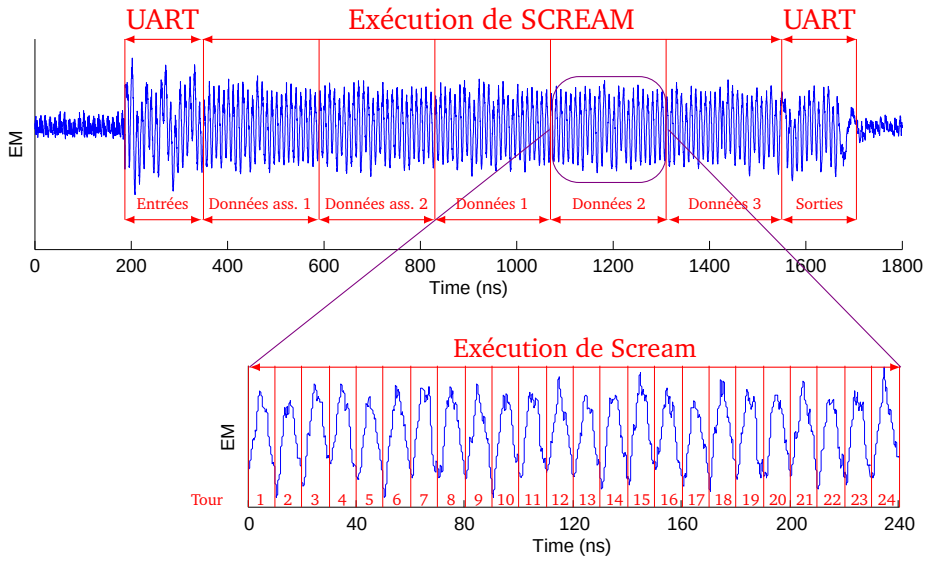


Figure 3.12.: Analyse des rayonnements électromagnétiques de SCREAM.

En pratique, un attaquant peut cibler n'importe quel bloc pour mener la DFA, il lui suffit de perturber le dernier étage linéaire, à condition bien sûr que le nonce ne soit pas changé à chaque exécution (ce qui ne respecte donc pas les spécifications de SCREAM). Dans notre expérimentation, nous avons ciblé le dernier bloc afin de mener l'attaque. De ce fait, nous ne donnerons que la valeur du dernier chiffré corrompu dont la valeur correcte était 0x85856feac83bbe335c6bcb2f5f6d81a6.

Nous avons alors utilisé le banc d'injection d'impulsions de tension décrit à la figure 3.1 afin de perturber l'exécution du dernier étage linéaire de Scream appliqué sur le dernier bloc, avec un nonce fixe pour notre expérimentation.

Toutefois, la durée d'une impulsion étant de 6 ns minimum, elle ne permet pas une précision suffisante pour cibler uniquement le dernier étage linéaire puisque la durée d'un tour est d'environ 10 ns comme nous pouvons le constater à la figure 3.12. Nous avons donc perturbé aléatoirement le dernier tour du chiffrement, contrairement à une implémentation logicielle pour laquelle il est possible de cibler dans le temps précisément une instruction donnée.

Tout d'abord, nous avons effectué des injections sur 100 positions spatiales distribuées sur une grille 10×10 sur le boîtier du circuit intégré de taille 19×19 mm^2. Pour chaque position spatiale, nous avons testé 11 positions temporelles, 4 tensions et nous avons effectué 2 injections, c'est-à-dire 88 injections par position.

Sur le total des 8 800 injections effectuées, nous avons obtenu 465 chiffrés corrompus avec au maximum 88 sur une position spatiale. La figure 3.13 donne la distribution des fautes que nous avons observées sur le boîtier du circuit intégré.

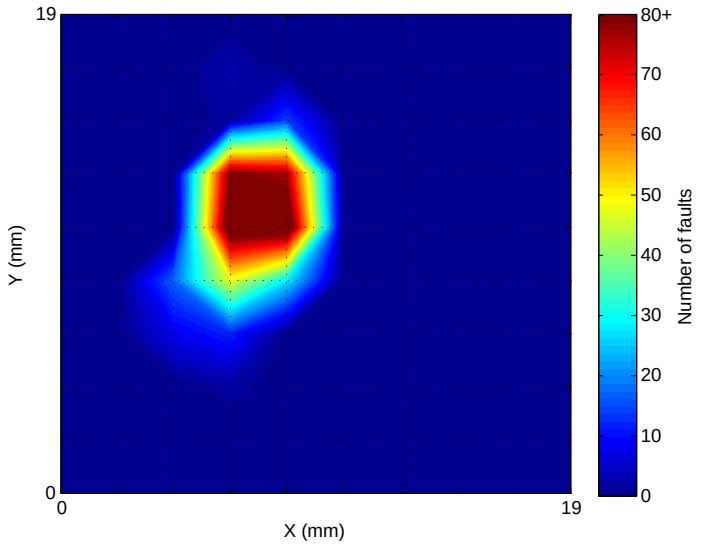


Figure 3.13.: Distribution des fautes obtenues sur le boîtier du circuit intégré.

Nous avons ensuite ciblé la zone sensible, qui correspond certainement au circuit intégré, et nous avons effectué un total de 69 250 injections.

Nous avons obtenu un total de 2 482 chiffrés corrompus sur Scream, dont 937 de valeurs distinctes. Pour chacun, nous avons calculé la valeur de la différence en sortie de la dernière couche de substitution et nous avons vérifié que la différence en sortie de chaque S-box active pouvait être obtenue à partir d'une même différence égale à 2^i avec $0 \leq i \leq 7$ en entrée.

Au total, nous avons obtenu 36 chiffrés corrompus différents vérifiant cette propriété, dont seulement 7 nous ont permis de retrouver autant d'information sur la clé secrète qu'avec toutes les fautes.

Les valeurs de ces 7 chiffrés corrompus sont données dans le tableau 3.15, ainsi que la valeur de la différence Δ_{out} en sortie de la dernière couche de substitution, et la valeur de la différence δ_{in} en entrée de chaque S-box, où λ désigne l'ensemble {0x01,0x02} des valeurs possibles pour δ_{in}.

Tableau 3.15.: Chiffrés corrompus obtenus à partir de fautes lors de l'exécution de l'implémentation matérielle de SCREAM avec un nonce fixe.

Chiffrés corrompus	Δ_{out}	δ_{in}
0x04eb0f2430df5f9301047fae10109f6d	0x003347ca19002a00d95d000000548a00	0x01
0xdb31fb5345d645d102dff1f33921e7ea	0x003300ca00000000d900fdcc2a0000c2	0x01
0xdbe8c81128d80ce36d7cf1f308179d43	0x003347ca000000d5d95dfd002a000000	0x01
0xea8ab50cc83bba5082eea0435f6d302c	0xb8330000190000000000000002a000000	λ
0xeaff0090da5cd1494e0ccb2f227081a6	0x00004b540000000000000000000000000	0x04
0x880d56417b40b585a3ee75dc74f12425	0xb98800000000001a0058f9001aee00e9	0x08
0xd9bdd4df0cbe9dbb23dbe8a7aa07b049	0x0000003a00000000000000bc000046e9	0x08

Le tableau 3.16 donne les différentielles sur $\mathcal{S}^{-1}$ qui nous ont permis de retrouver autant d'information qu'avec toutes les fautes, et donne les candidats en notation hexadécimale obtenus par intersection, notée $\cap$, pour chaque octet B_i de

$$\text{L-layer}^{-1}(\text{C} \oplus \text{K} \oplus \text{T}) \oplus \mathcal{C}t_{23}$$

pour $0 \leq i \leq 15$, où C, T et $\mathcal{C}t_{23}$ sont respectivement le chiffré, le Tweak (fixe) et la dernière constante de tour.

Finalement, nous avons obtenu $6144 \approx 2^{12,6}$ candidats pour

$$\text{L-layer}^{-1}(\text{C} \oplus \text{K} \oplus \text{T}) \oplus \mathcal{C}t_{23}$$

et nous avons retrouvé la clé secrète, à partir des valeurs du chiffré C, du Tweak T et de la dernière constante de tour $\mathcal{C}t_{23}$, en testant toutes les possibilités.

Tableau 3.16.: Connaissance à partir du tableau 3.15 sur chaque octet B_i pour $0 \leq i \leq 15$ en notation hexadécimale, où $*$ désigne l'ensemble des 256 valeurs possibles.

	Différentielles obtenues sur $\mathcal{S}^{-1}$	$\cap$
B_0	(b8,01) et (b9,08)	03
B_1	(33,01) et (88,08)	0e
B_2	(47,01) et (4b,04)	2e
B_3	(ca,01), (54,02) et (3a,08)	ef
B_4	(19,01)	{2b,32,4f,56,65,7c}
B_5	Aucune	$*$
B_6	(2a,01)	{d1,fb}
B_7	(d5,01) et (1a,08)	cb
B_8	(d9,01)	{02,db}
B_9	(5d,01) et (58,08)	3f
B_{10}	(fd,01) et (f9,08)	48
B_{11}	(cc,01) et (bc,08)	59
B_{12}	(2a,01) et (1a,08)	d1
B_{13}	(54,01) et (ee,08)	e4
B_{14}	(8a,01) et (46,08)	9e
B_{15}	(c2,01) et (e9,08)	97

Il est à noter que la mise en œuvre de l'attaque a demandé beaucoup plus de fautes sur une implémentation matérielle que sur une implémentation logicielle, car la technique d'injection que nous avons utilisée n'affecte pas les différentes architectures de la même façon. Néanmoins, l'expérimentation que nous avons proposée a montré la faisabilité de la DFA sur ces deux types d'implémentation.

3.4 Contre-mesures

Les chiffrements SPN ne sont pas les seuls à être vulnérables aux attaques par perturbation. En effet, on retrouve ce type d'attaque sur la plupart des structures existant aujourd'hui, que ce soit sur les réseaux de Feistel [Le Bouder et al. 2014], sur les structures ARX [Kumar et al. 2017], sur les chiffrements à flot [Barenghi et al. 2012] ou sur les schémas de chiffrement à clé publique [Berzati et al. 2009].

Il est donc très important de proposer des contre-mesures matérielles ou logicielles efficaces pour se prémunir de ces attaques, dont l'objectif consiste à empêcher un attaquant d'obtenir une faute exploitable, en modifiant la valeur de la faute ou en empêchant directement l'injection de fautes.

Dans cette section, nous décrivons dans un premier temps les différentes possibilités de contre-mesures matérielles et logicielles existant, et nous évaluons ensuite leur efficacité pour se prémunir de la DFA que nous avons introduite.

3.4.1 — Contre-mesures matérielles

Du côté des contre-mesures matérielles, il est possible d'utiliser une couche de métal au dessus du circuit intégré afin de se prémunir des injections par rayonnement lumineux, on parle alors d'un bouclier passif [Van Woudenberg et al. 2011b].

Cependant, il est possible d'enlever un bouclier passif en utilisant des moyens chimiques, et ce type de protection ne permet pas de se prémunir de l'injection électromagnétique. Plutôt qu'une simple couche de métal, il est possible d'utiliser des mailles métalliques qui transmettent des signaux sur la surface du circuit intégré et détectent toute interruption sur un fil afin de se prémunir cette fois de la plupart des injections de fautes, on parle alors d'un bouclier actif [Karaklajic et al. 2013].

L'utilisation de capteurs [El-Baze et al. 2016] peut également permettre de détecter des anomalies dans le comportement du circuit intégré. Certains circuits asynchrones, qui n'utilisent pas de signal d'horloge, peuvent aussi permettre d'appliquer des techniques de codage pour empêcher certaines injections de fautes [Moore et al. 2002]. Enfin, l'insertion de contrôles de parité sur les chemins de données peut également permettre de détecter d'éventuelles erreurs introduites par les injections de fautes [Bertoni et al. 2002], [Karri et al. 2003].

Le problème avec les contre-mesures matérielles est d'une part leur coût de mise en œuvre généralement assez élevé, et d'autre part même si les simulations montrent qu'elles sont efficaces, il n'y a aucune garantie absolue que ce sera le cas sur le produit final lors de son déploiement où d'autres considérations comme le routage, les processus de fabrication ou la découverte de nouveaux chemins d'attaques peuvent avoir un impact sur l'efficacité des contre-mesures.

Et lorsque le produit final est obtenu, ou pire déjà déployé, il serait coûteux d'avoir une autre itération du cycle de conception pour le sécuriser si une faille de sécurité était identifiée. Il est donc fortement recommandé d'utiliser ces contre-mesures matérielles avec des contre-mesures logicielles.

3.4.2 — Contre-mesures logicielles

Tout d'abord, nous allons analyser la résistance aux attaques par perturbation des contre-mesures logicielles présentées en section 2.4, qui permettent principalement de contrecarrer certaines attaques par observation, afin de savoir si certaines permettent de se prémunir de ces deux catégories d'attaques physiques.

Diminuer les fuites en équilibrant la consommation ou le traitement des calculs cryptographiques n'empêche en rien l'injection de fautes et leur exploitation. Il en est de même pour l'ajout de délais aléatoires ou le "shuffling", qui peuvent néanmoins complexifier certaines attaques puisque cibler une instruction précise devient plus difficile. Dans le cas de la DFA que nous avons proposée, qui permet de savoir si une faute est exploitable ou non à partir de la valeur du chiffré corrompu, il suffit à un attaquant d'injecter des fautes jusqu'à obtenir l'effet désiré.

La technique de masquage que nous avons présentée dans la section 2.4 permet également de complexifier certaines attaques par perturbation. En effet, si un attaquant obtient une faute e sur un mot masqué S_a des données, c'est-à-dire obtient une différence $S_a \oplus e$, où $S_a = X_a \oplus M_a$ avec X_a la valeur correcte et M_a la valeur du masque, alors la faute n'affecte pas la valeur du masque sur la partie linéaire du schéma de chiffrement, mais l'affecte sur les opérations non-linéaires.

Le résultat de l'opération AND entre le mot corrompu $S_a \oplus e$ et un mot $S_b = X_b \oplus M_b$ sera effectivement égal à $S_a S_b \oplus e S_b$, et la technique consiste à ajouter par un XOR les termes nécessaires pour modifier $S_a S_b$ par $X_a X_b \oplus M_{ab}$ où M_{ab} est la nouvelle valeur du masque. Ainsi, le mot corrompu devient

$$X_a X_b \oplus M_{ab} \oplus e S_b = X_a X_b \oplus M_{ab} \oplus e X_b \oplus e M_b$$

au lieu de $X_a X_b \oplus e X_b$ dans le cas d'une injection sur l'implémentation non protégée. Le masque M_{ab} est alors correctement retiré à la fin de l'exécution mais l'attaque doit prendre en considération le fait que la faute dépend également du masque utilisé après une opération non-linéaire.

La DFA que nous avons proposée n'est donc pas directement applicable sur une implémentation masquée. Cependant, il est possible d'exploiter le fait que la valeur de $e M_b$ a 75% de chance d'être nulle afin de mener une DFA, et on parle d'attaque par fautes statistiques [Dobraunig et al. 2016].

Il est à noter que cette technique de masquage n'apporte également aucune protection contre les analyses simples de fautes qui exploitent le fait qu'une faute a ou n'a pas eu d'effet sur l'exécution, et contre un saut d'instruction sur la dernière couche d'addition de clé puisque le masque est retiré des données directement après cette opération qui est linéaire par rapport au XOR.

Le masquage du message clair ou de la clé secrète a néanmoins souvent été proposé pour se prémunir de certaines attaques par perturbation, que ce soit sur les schémas de chiffrement à clé publique [Kocher 1996], [Messerges et al. 1999] ou sur les schémas de chiffrement à clé secrète [Akkar et al. 2001].

L'idée étant de ne pas avoir deux exécutions avec les mêmes paramètres d'entrée, et pour cela le masque doit être changé à chaque exécution.

Afin de se prémunir uniquement de la DFA, il est possible d'appliquer le même masque lors de l'exécution de la fonction de chiffrement et de la fonction de déchiffrement, sans retirer le masque à la fin de l'exécution [GUILLEY et al. 2010].

Cependant, la valeur du masque ne doit pas être prévisible par l'attaquant dans ce cas là sinon, pour deux exécutions avec des masques M_1 et M_2, il peut choisir deux messages clairs P_1 et P_2 tels que $P_1 \oplus M_1 = P_2 \oplus M_2$ afin d'obtenir deux exécutions sur un même message clair.

Les chiffrements par blocs adaptables utilisent également une technique similaire, les Tweaks peuvent en effet être vus comme des masques, qui doivent être les mêmes lors de l'exécution de la fonction de chiffrement et de la fonction de déchiffrement, et qui doivent être changés à chaque nouvelle exécution.

La redondance est également une contre-mesure efficace pour se prémunir des attaques par perturbation, et permet de contrer la plupart des attaques actuelles. L'idée consiste à exécuter les mêmes opérations sur une ou plusieurs copies des données, spatialement ou temporellement, et à comparer les résultats obtenus.

Alors, afin d'effectuer une attaque par perturbation, l'attaquant doit obtenir le même effet sur toutes les copies des données.

Il est possible de seulement détecter la faute et de ne pas retourner la sortie du schéma de chiffrement lorsque toutes les copies ne donnent pas le même résultat, par exemple en bloquant le circuit intégré, mais dans ce cas un attaquant peut mener une analyse simple de fautes où il n'a besoin que de savoir si une faute a été injectée ou non lors de l'exécution.

Il est sinon possible de corriger la faute en appliquant un vote majoritaire sur trois copies ou plus, ce qui permet de se prémunir des analyses simples de fautes. Un vote majoritaire consiste à retourner la valeur qui apparaît le plus.

Ainsi, la redondance permet de se prémunir des attaques par perturbation sans considérer le modèle de faute, puisqu'elle permet d'empêcher l'attaquant d'obtenir des valeurs corrompues lorsqu'un nombre suffisant de copies est utilisé. Cependant, chaque copie spatiale (resp. temporelle) demande un surcoût du même nombre d'opérations à protéger en mémoire (resp. en temps) [MESTIRI et al. 2013].

3.4.3 Évaluation des contre-mesures face à la DFA

Tout d'abord, il est à noté que certains modes de chiffrement comme le mode OFB ou le mode CTR présentés en section 1.3.2.4 permettent de se prémunir de la DFA que nous avons proposée, puisque ces modes de chiffrement appliquent la fonction de chiffrement directement sur un IV ou sur un compteur, dont la valeur change à chaque exécution. De ce fait, un attaquant ne peut pas exécuter deux fois le chiffrement par blocs sur un même message clair.

Comme nous l'avons vu, le masquage permet également de contrecarrer le chemin d'attaque que nous avons introduit. La technique présentée en section 2.4 apporte ainsi une première sécurité d'un côté face à certaines attaques par observation, et d'un autre face à certaines attaques par perturbation.

Afin de ne se prémunir que de la DFA que nous avons proposée, il est alors moins coûteux de n'appliquer qu'un seul masque en entrée du chiffrement SPN sans le retirer à la fin [GUILLEY et al. 2010], mais dans ce cas l'attaquant ne doit pas être en mesure de prédire la valeur du masque pour qu'il ne puisse pas le retirer comme nous l'avons expliqué dans la section 3.4.2.

Nous avons proposé une technique afin de ne pas avoir de contrainte sur le masque, nous sommes donc en mesure de rendre le générateur de masques public, et donc prévisible par un attaquant.

Pour cela, le masque doit être ajouté aux données SM $= \mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P})$ au milieu de l'exécution. C'est-à-dire, pour un chiffrement SPN de r tours qui prend en entrée un message clair P de n bits, la contre-mesure consiste à exécuter

$$\mathcal{E}_{\mathrm{K}}^{(1)}(\mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P}) \oplus \mathrm{M})$$

où M est un masque aléatoire de n bits et $\mathcal{E}_{\mathrm{K}}^{(0)}$ (resp. $\mathcal{E}_{\mathrm{K}}^{(1)}$) correspond à l'exécution des $\lceil r/2 \rceil$ premiers tours (resp. $\lfloor r/2 \rfloor$ derniers tours) de la fonction de chiffrement.

La fonction de déchiffrement $\mathcal{D}_{\mathrm{K}} = \mathcal{D}_{\mathrm{K}}^{(1)} \circ \mathcal{D}_{\mathrm{K}}^{(0)}$ doit être synchronisée avec la fonction de chiffrement (comme pour le Tweak d'un chiffrement par blocs adaptable ou pour l'IV d'un mode opératoire).

Pour cela, il est possible d'utiliser le même processus que pour un mode opératoire qui synchronise l'IV pour le chiffrement et le déchiffrement, et qui est donc déjà souvent disponible sur les systèmes existant.

La figure 3.14 illustre la contre-mesure avec les notations introduites.

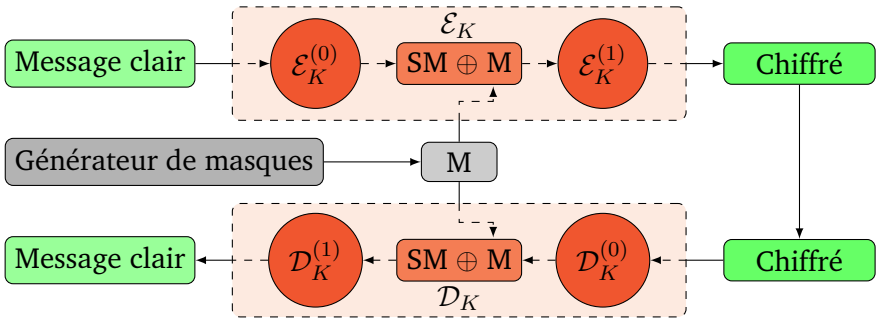


Figure 3.14.: Masquage au milieu de l'exécution.

Alors, le générateur de masques peut être public puisque, pour mener la DFA sur la fonction de chiffrement, l'attaquant doit obtenir un chiffré correct et un chiffré corrompu

$$\mathrm{C} = \mathcal{E}_{\mathrm{K}}^{(1)}(\mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P}_1) \oplus \mathrm{M}_1) \text{ et } \mathrm{C}^* = \mathcal{E}_{\mathrm{K}}^{(1)}(\mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P}_2) \oplus \mathrm{M}_2)$$

tels que les entrées de $\mathcal{E}_{\mathrm{K}}^{(1)}$ sont les mêmes sur les deux exécutions, c'est-à-dire

$$\mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P}_1) \oplus \mathrm{M}_1 = \mathcal{E}_{\mathrm{K}}^{(0)}(\mathrm{P}_2) \oplus \mathrm{M}_2. \tag{3.3}$$

De façon similaire, pour mener l'attaque sur la fonction de déchiffrement, les entrées de $\mathcal{E}_{\mathrm{K}}^{(0)}$ doivent être les mêmes sur les deux exécutions.

Il y a alors deux stratégies pour trouver deux paires d'entrées qui satisfont l'équation (3.3). La première consiste à utiliser un algorithme générique (sans exploiter aucune propriété spécifique du schéma de chiffrement).

En effet, à partir du paradoxe des anniversaires, il faut $2^{n/2}$ exécutions où n est la taille du bloc de données, c'est-à-dire que l'attaquant doit effectuer 2^{32} injections de fautes pour un bloc de 64 bits, ce qui n'est pas faisable en pratique.

La seconde stratégie consiste à exploiter certains chemins différentiels de $\mathcal{E}_K^{(0)}$, mais c'est à nouveau infaisable si $\mathcal{E}_K^{(0)}$ n'a pas de chemin différentiel de probabilité supérieure à $2^{n/2}$. De ce fait, le générateur de masques peut être un simple registre à décalage à rétroaction linéaire (LFSR en anglais pour "Linear Feedback Shift Register"), qui ne doit pas être modifiable par l'attaquant.

Il est à noter que l'attaquant ne doit pas avoir accès aux deux fonctions de chiffrement et de déchiffrement dans le cas où elles sont paramétrées par la même clé secrète et le même masque. En effet, l'attaquant peut sinon mener la DFA sur le déchiffrement puisqu'il connaît la valeur correcte du message clair.

Ainsi, le masquage apporte une première sécurité face aux attaques physiques, puisqu'il permet de se prémunir de la plupart des attaques par observation et de certaines attaques par perturbation, mais ne permet pas une protection contre toutes les DFA [BOSCHER et al. 2008], ni contre certaines analyses simples par fautes.

La redondance est alors un bon moyen de sécurité, ou apporte une sécurité supplémentaire, face aux attaques par perturbation. De ce fait, de plus en plus d'études tentent de réduire le coût de la redondance.

3.5 Intra-Instruction Redundancy

Récemment, une contre-mesure basée sur la redondance et nommée l'IIR (pour "Intra-Instruction Redundancy" en anglais) a été proposée pour se prémunir des attaques par fautes [PATRICK et al. 2016]. L'IIR consiste à utiliser le parallélisme par tranches de bits afin d'appliquer un schéma de chiffrement sur 32 blocs en entrée.

L'objectif est d'exploiter pleinement une architecture 32 bits en prenant en entrée 15 blocs de données entrelacés avec 15 blocs de redondance et 2 blocs de référence. Les blocs de référence sont des constantes en entrée (message clair et clé secrète) pour lesquelles le chiffré correspondant est connu.

La figure 3.15 illustre la composition des mots en entrée d'une implémentation protégée par l'IIR appliquée aux messages en clair de 128 bits $P_i = P_i^1 \cdots P_i^{128}$ avec $0 \leq i \leq 14$ et utilisant deux messages en clair de 128 bits de référence $RP_0 = RP_0^1 \cdots RP_0^{128}$ et $RP_1 = RP_1^1 \cdots RP_1^{128}$.

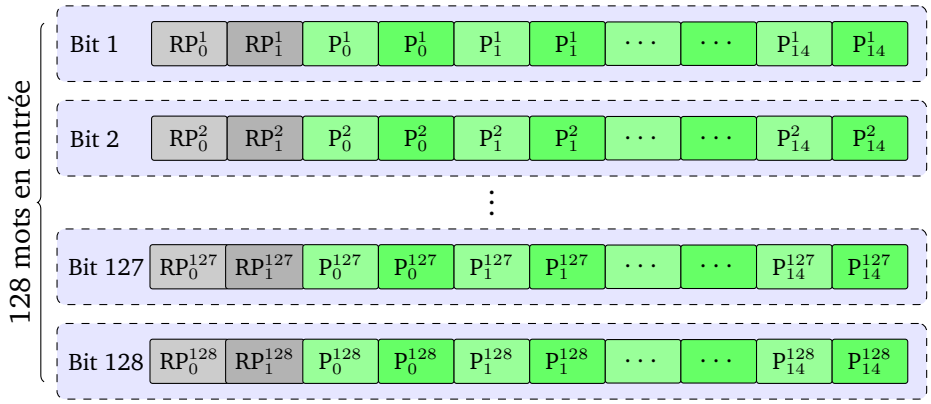


Figure 3.15.: Parallélisme par tranches de bits avec l'Intra-Instruction Redundancy.

L'IIR permet principalement de se prémunir contre les fautes sur un seul bit grâce à la redondance et des sauts d'instructions grâce aux blocs de référence. En effet, si le bloc de référence est choisi de telle sorte que chaque instruction affecte sa valeur, alors un saut d'instruction le modifiera et sera détecté à la fin de l'exécution.

Malheureusement, les fautes sur plusieurs bits peuvent être exploitables, par exemple une faute sur les deux bits d'un bloc de données ne sera pas détectée. De plus, l'IIR impose d'utiliser dans la plupart des cas une implémentation moins efficace du schéma de chiffrement à cause du surcoût nécessaire pour appliquer le parallélisme par tranches de bits [PORNIN 2001].

Il impose également de chiffrer 15 blocs de données à chaque exécution et d'utiliser n mots afin de stocker et manipuler une entrée de n bits.

Cependant, utiliser un bloc de référence est très efficace contre les sauts d'instructions. De ce fait, nous avons regardé la possibilité de garder cette propriété tout en utilisant une implémentation conventionnelle du schéma de chiffrement (c'est-à-dire sans l'utilisation du parallélisme par tranches de bits).

Conclusion du chapitre

Les attaques par perturbation sont de véritables menaces qu'il est important de prendre en considération lorsque l'on regarde la sécurité des objets connectés.

Nous avons pu en effet observer de nombreux chemins d'attaques dans ce chapitre, et des techniques diverses et variées pour les mettre en œuvre.

Nous avons notamment proposé un nouveau chemin d'attaque sur les chiffrements SPN, particulièrement efficace sur les structures favorisant le parallélisme par tranches de bits, et nous avons montré qu'un attaquant peut l'exploiter sur la plupart des implémentations en pratique, puisque nous l'avons validé en laboratoire par injection électromagnétique sur deux implémentations logicielles, 8 bits et 32 bits, et sur une implémentation matérielle de chiffrements SPN [LAC et al. 2017].

Bien qu'il existe aussi de nombreuses contre-mesures pour contrecarrer ces attaques, le surcoût nécessaire pour les déployer n'est souvent pas en accord avec les besoins en terme de performances dans le contexte de l'IOT.

Le masquage apporte une première sécurité contre certaines attaques par perturbation, mais ne permet de toutes les contrer.

La redondance est alors une technique efficace pour s'en prémunir, et nous avons finalement présenté l'IIR [PATRICK et al. 2016], qui a été introduit pour réduire le coût de la redondance sur les architectures 32 bits.

Internal Redundancy Countermeasure

Chapitre 4

> *Avec l'Internet et le Web, la demande en cryptologie a explosé. Et paradoxalement, la cryptologie est passée d'une science du secret à une science de confiance.*
>
> **— Jacques Stern**

Introduction

L'IRC est une contre-mesure logicielle basée sur de la redondance que nous avons proposée pour se prémunir des attaques par fautes sur des architectures 32-bits. Après avoir décrit le principe général de l'IRC et les prérequis nécessaires à son déploiement, nous détaillons comment appliquer l'IRC aux chiffrements par blocs et aux chiffrements à flot. Ensuite, nous présentons une technique pour utiliser efficacement le masquage avec l'IRC et nous analysons des tests pratiques menés en laboratoire sur des implémentations protégées par l'IRC avant de conclure par une généralisation de l'IRC à d'autres architectures. Ces travaux ont fait l'objet d'un article présenté à la conférence "ISCAS" en 2018 [Lac et al. 2018].

Sommaire

4.1 Principe général **90**
4.1.1 Prérequis 92
4.1.2 Implémentation 93
4.2 Déploiement de l'IRC sur les schémas de chiffrement à clé secrète **94**
4.2.1 Application aux chiffrements par blocs 95
4.2.2 Application aux chiffrements à flot 96
4.3 Implémentation efficace du masquage en utilisant l'IRC **99**
4.3.1 Masquage de premier ordre 99
4.3.2 Masquage d'ordre supérieur 100
4.4 Tests pratiques conduits en laboratoire **101**
4.4.1 Implémentation de PRIDE protégée par l'IRC 102
4.4.2 Implémentation de Trivium protégée par l'IRC 104
4.5 Généralisation **108**

4.1 Principe général

La redondance permet de se prémunir de la plupart des attaques par perturbation, comme nous l'avons mentionné à la section 3.4.

Il est commun d'utiliser une implémentation 32 bits d'un schéma de chiffrement, c'est-à-dire qui manipule des mots de 32 bits, afin d'exploiter pleinement les capacités d'une architecture 32 bits. Cependant, l'utilisation de la redondance spatiale demande un surcoût important dans ce cas, et elle n'est pas toujours applicable.

En effet, cela demande d'effectuer plusieurs fois la même instruction en parallèle, ce qui nécessite le plus souvent un processeur multi-cœurs, c'est-à-dire possédant plusieurs cœurs physiques fonctionnant simultanément, ce qui n'est pas le cas pour la plupart des architectures 32 bits destinées aux applications de l'IoT.

L'IIR, que nous avons présenté à la section 3.5, propose alors de réduire le coût de la redondance sur une architecture 32 bits en utilisant le parallélisme par tranches de bits au lieu d'une implémentation 32 bits classique, mais cela ne permet pas de se prémunir de toutes les fautes, impose un surcoût important pour stocker les données et demande de chiffrer 15 blocs à chaque exécution.

Afin de résoudre ces problèmes, et de diminuer le surcoût de la redondance, nous proposons d'utiliser une implémentation 8 bits efficace d'un schéma de chiffrement opérant simultanément sur 4 octets au sein d'un mot de 32 bits.

Nous utilisons également des blocs de référence, dont les valeurs sont publiques, afin d'augmenter l'efficacité de la contre-mesure, notamment pour se prémunir des sauts d'instructions. Les mots manipulés sont alors composés d'un octet de données entrelacé avec l'octet correspondant du bloc de référence et deux copies, qui dépendent du schéma de chiffrement ciblé et du niveau de sécurité requis.

Il existe deux possibilités de se prémunir d'une même faute sur k copies spatiales :

- Détection de fautes : il s'agit d'utiliser $k+1$ copies spatiales et de ne pas retourner la sortie du système lorsque toutes les copies ne conduisent pas au même résultat. Il est à noter que dans ce cas, un attaquant peut faire une analyse simple de fautes pour laquelle il a seulement besoin de savoir si une faute a été injectée ou non lors de l'exécution du schéma de chiffrement.
- Correction de fautes : il s'agit d'utiliser $2k+1$ copies spatiales et de retourner la valeur qui apparaît le plus en appliquant un vote de majorité. Ce mode ajoute une sécurité additionnelle face aux analyses simples de fautes.

L'IRC offre la possibilité d'utiliser l'une de ces deux stratégies. La figure 4.1 illustre la composition d'un mot de 32 bits utilisé par l'IRC en mode détection de fautes.

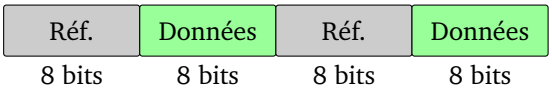


Figure 4.1.: Composition d'un mot de 32 bits en mode détection de fautes.

La figure 4.2 illustre la composition d'un mot de 32 bits utilisé par l'IRC en mode correction de fautes, pour lequel le bloc de référence est découpé en deux quartets placés entre les copies des données dans chaque mot de 32 bits.

Figure 4.2.: Composition d'un mot de 32 bits en mode correction de fautes.

L'IRC remplace alors chaque opérateur 8 bits par un seul flot d'instructions 32 bits correspondant à la même opération appliquée indépendamment sur chaque octet d'une façon SIMD (pour "Single Instruction on Multiple Data" en anglais), c'est-à-dire de sorte à travailler sur les octets en parallèle.

Il est à noter que certaines architectures disposent d'instructions SIMD adaptées au mode détection de fautes, ce n'est par contre pas le cas du mode correction de fautes, pour lequel les opérations sont plus complexes à implémenter sur 32 bits. Par exemple, certaines architectures ARM proposent une instruction d'addition sur un octet de façon SIMD, UADD8, et de soustraction, USUB8.

Le nombre d'instructions nécessaires pour chiffrer un même bloc de données avec une implémentation 8 bits est souvent plus important que celui d'une implémentation 32 bits, ce qui implique qu'une implémentation protégée par l'IRC a un surcoût en temps par rapport à une implémentation 32 bits.

Par contre, l'IRC permet de réduire considérablement le surcoût en taille de code puisqu'il ne demande qu'un seul flot d'instructions au lieu de quatre en parallèle (ce qui n'est pas toujours possible en fonction de l'architecture).

Finalement, à la fin de l'exécution, l'IRC effectue des comparaisons entre les différentes copies, et avec le bloc de référence stocké sur le circuit intégré. Si aucune faute n'a été injectée, toutes les copies sont alors égales et les chiffrés de référence obtenus sont aussi égaux à celui stocké dans le circuit.

De ce fait, afin de corrompre un message, l'attaquant doit être capable d'obtenir la même faute sur les deux copies des données sans affecter les blocs de référence.

Obtenir un tel effet est extrêmement difficile en pratique du fait que le bloc de référence est placé entre les deux copies des données, puisque les types de fautes qu'un attaquant est capable d'obtenir avec les techniques actuelles sont un saut d'instruction ou une mise à zéro, une mise à un ou une inversion d'un ou plusieurs bits consécutifs, comme nous l'avons mentionné à la section 3.2.

4.1.1 — Prérequis

Afin d'appliquer efficacement l'IRC, il est nécessaire de respecter certains prérequis sur le bloc de référence et sur l'implémentation utilisée.

Tout d'abord, aucun octet du bloc de référence ne doit être égal à 0xFF (resp. 0x00) sur toute la zone à protéger, sinon une mise à un (resp. une mise à zéro) du mot de 32 bits correspondant permet à l'attaquant d'obtenir un message corrompu valide puisque le bloc de référence n'est pas modifié.

Ensuite, chaque instruction de l'implémentation utilisée doit modifier la valeur du bloc de référence, sinon un saut de l'instruction correspondante n'affecte pas le bloc de référence et aboutit également à un message corrompu valide.

Il est possible d'obtenir ces deux propriétés en testant des messages clairs et des clés aléatoires. En effet, il suffit pour cela d'utiliser le code suivant après chaque instruction appliquée à un octet B lors du chiffrement et du déchiffrement :

```
uint8_t temp;
temp =B;
...
/* Instruction modifiant B */
...
if (B==0x00||B==0xff||B==temp){
    return 0;
}
```

Il est également possible d'utiliser deux blocs de référence différents pour satisfaire ces deux propriétés plus facilement.

Enfin, dans le cas où une implémentation utilise des instructions conditionnelles qui dépendent des données, comme MICKEY [Babbage et al. 2008], il est nécessaire de les transformer afin de n'utiliser que des instructions non-conditionnelles, ce qui engendre généralement un surcoût assez élevé.

Néanmoins, les instructions conditionnelles ne sont pas recommandées, puisqu'il est possible qu'un attaquant les exploite afin de mener une analyse simple de fautes.

Il est aussi préférable que le schéma de chiffrement n'utilise pas de tableau, puisque l'application SIMD d'un tableau demande généralement un surcoût élevé, sauf s'il existe une instruction dédiée dans le jeu d'instructions de l'architecture (ISA pour "Instruction Set Architecture" en anglais). De ce fait, lorsque le schéma de chiffrement utilise des S-boxes, il est recommandé d'utiliser les formes algébriques normales des coordonnées de chacune, c'est-à-dire d'exprimer les bits en sortie comme une expression booléenne des bits en entrée.

4.1.2 Implémentation

Les opérateurs bit à bit peuvent être considérés comme des opérateurs intrinsèquement SIMD, et peuvent donc être directement utilisés par l'IRC. En absence d'instructions SIMD dans l'ISA, il est possible d'implémenter les opérateurs non intrinsèquement SIMD en utilisant quelques instructions supplémentaires, systématiquement opérant de la même façon sur les 4 octets d'un mot de 32 bits afin d'assurer l'unicité du flot d'instruction.

Certains langages de programmation (comme le C++) offrent la possibilité d'utiliser des opérateurs comme des fonctions en les définissant pour de nouveaux types de données, on parle de surcharge d'opérateur, et cela permet alors de déployer l'IRC sur une implémentation de référence sans avoir à modifier son code.

Ainsi, en langage C++, il est possible de déployer l'IRC directement à partir d'un code de référence paramétrique d'un schéma de chiffrement :

```
template<typename byte>
void Cipher(byte *state, byte *key, ···) { ··· }
```

sans avoir à écrire un compilateur dédié. En effet, il suffit simplement d'utiliser l'instruction suivante pour obtenir une instance non protégée du schéma de chiffrement :

```
Cipher<uint8_t>(state, key, ···);
```

et l'instruction suivante pour obtenir une instance protégée par l'IRC :

```
Cipher<IRC>(state, key, ···);
```

où IRC est une classe qui implémente la contre-mesure d'une façon complètement générique, c'est-à-dire indépendamment du schéma de chiffrement, et qui fournit les opérateurs nécessaires pour travaille d'une façon SIMD sur les octets :

```
class IRC{
    private:union{
        uint8_t bytes[4];
        uint32_t word;
    }state;
```

```
    public:
    IRC(){state.word=0x0;}
    IRC(uint8_t data, uint8_t knownPT){
        state.bytes[0]=knownPT;
        state.bytes[1]=data;
        state.bytes[2]=knownPT;
        state.bytes[3]=data;}
    IRC &operator>>=(const uint8_t shift){
        static const uint32_t mask[9] = {
            0xffffffff,0x7f7f7f7f,0x3f3f3f3f,
            0x1f1f1f1f,0x0f0f0f0f,0x07070707,
            0x03030303,0x01010101,0x00000000};
        assert(shift>=0 && shift<=8);
        state.word>>=shift;
        state.word&=mask[shift];
        return *this;}
    IRC operator>>(const int shift)const{
        IRC res(*this);
        res>>=shift;
        return res;}
    IRC &operator+=(const IRC &other){
        uint32_t result=state.word&0x7f7f7f7f;
        result+=other.state.word&0x7f7f7f7f;
        uint32_t carry=result&0x80808080;
        result&=0x7f7f7f7f;
        carry+=state.word&0x80808080;
        carry+=other.state.word&0x80808080;
        carry&=0x80808080;
        state.word=result|carry;
        return *this;}
    IRC &operator+=(const uint8_t byte){
    IRC tmp(byte,byte);
    (*this)+=tmp;
    return *this;}
    ...
}
```

Enfin, certains compilateurs (comme GCC) permettent d'écrire un code portable, et de bénéficier automatiquement des instructions SIMD disponibles sur l'ISA ciblé lorsque les options appropriées du compilateur sont utilisées.

4.2 Déploiement de l'IRC sur les schémas de chiffrement à clé secrète

Nous allons à présent détailler le déroulement d'une implémentation protégée par l'IRC en mode détection de fautes dans le cas d'un chiffrement par blocs et d'un chiffrement à flot pour lesquels son déploiement est différent.

4.2.1 Application aux chiffrements par blocs

Soit $\mathcal{E}$ une implémentation 8 bits d'un chiffrement par blocs qui prend en entrée un message clair $P = P_1 \cdots P_b$ de b octets, utilise une clé secrète $K = K_1 \cdots K_{b'}$ de b' octets et produit un chiffré $C = C_1 \cdots C_b$ de b octets. L'IRC utilise un message clair de référence $RP = RP_1 \cdots RP_b$ de b octets, une clé de référence $RK = RK_1 \cdots RK_{b'}$ de b' octets et un chiffré de référence $RC = RC_1 \cdots RC_b$ de b octets.

Tout d'abord, pour chaque $1 \leq i \leq b$, l'IRC stocke dans un mot de 32 bits l'octet P_i concaténé avec RP_i, P_i et RP_i comme illustré à la figure 4.3.

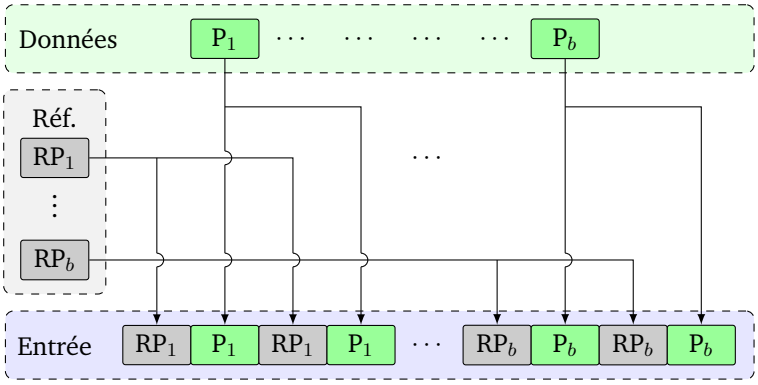


Figure 4.3.: L'IRC sur les chiffrements par blocs - composition des mots de 32 bits.

L'IRC stocke également, pour chaque $1 \leq i \leq b'$, l'octet K_i concaténé avec RK_i, K_i et RK_i. Il exécute ensuite la fonction de chiffrement avec un seul flot d'instructions 32 bits opérant indépendamment sur chaque octet, dénoté IRC($\mathcal{E}$), afin d'obtenir, pour chaque $1 \leq i \leq b$, l'octet C_i concaténé avec RC_i, C_i et RC_i. La figure 4.4 illustre l'exécution de IRC($\mathcal{E}$).

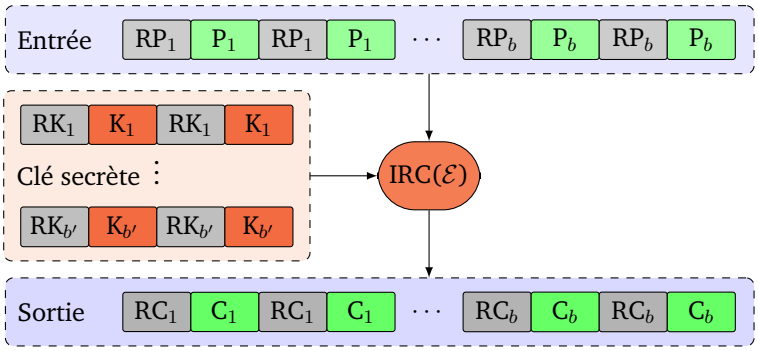


Figure 4.4.: L'IRC sur les chiffrements par blocs - exécution de IRC($\mathcal{E}$).

Finalement, l'IRC effectue des comparaisons impliquant chaque copie et le chiffré de référence stocké dans la mémoire du circuit. Le chiffré n'est alors retourné que si tous les tests sont valides comme illustré à la figure 4.5. Il s'agit d'une réaction possible, mais d'autres peuvent être envisagées.

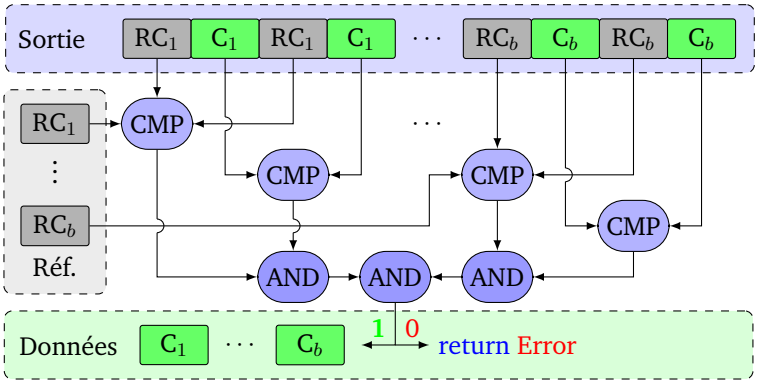


Figure 4.5.: L'IRC sur les chiffrements par blocs - comparaisons.

Il est à noter que cette dernière étape peut être faite avec seulement 2 comparaisons par mot de 32 bits : la première en comparant les 16 derniers bits de chaque mot avec les 16 premiers, et la seconde en comparant le chiffré de référence obtenu avec celui stocké dans la mémoire du circuit intégré.

4.2.2 — Application aux chiffrements à flot

Une implémentation 8 bits d'un chiffrement à flot synchrone est en général composée de deux étapes, que nous avons introduites à la section 1.3.2.4 :

- une étape d'initialisation où une fonction $\mathcal{I}$ génère un état interne initial IS_0 à partir de la clé secrète et d'un IV,
- une étape de génération de la suite chiffrante où une fonction de transition $\mathcal{T}$ et une fonction d'extraction $\mathcal{E}$ sont appliquées autant de fois que nécessaire sur l'état interne afin de générer une longueur suffisante de suite chiffrante. À chaque itération, la fonction $\mathcal{T}$ modifie la valeur de l'état interne et la fonction $\mathcal{E}$ retourne un octet de la suite chiffrante.

L'IRC applique alors dans un premier temps la même méthode pour l'étape d'initialisation que précédemment décrite sur les chiffrements par blocs.

Plus précisément, la fonction $\mathcal{I}$ est appliquée avec un seul flot d'instructions 32 bits opérant indépendamment sur chaque octet, noté IRC($\mathcal{I}$), afin de générer un état interne initial IRC(RS_0,IS_0) composé de l'état interne IS_0, d'un état interne de référence RS_0 et de leurs copies.

IRC(RS_0,IS_0) est obtenu à partir de IRC(RK,K), qui est composé de la clé secrète K, de la clé de référence RK et de leurs copies, et à partir de IRC(RI,IV), qui est composé de l'IV, de l'IV de référence RI et de leurs copies. Il est à noter que la clé et l'IV de référence peuvent être générés (plutôt que stockés).

La figure 4.6 illustre l'étape d'initialisation protégée par l'IRC en mode détection, où $IS_{n,i}$ (resp. K_i, IV_i, $RS_{n,i}$, RK_i et RI_i) est le i-ème octet de IS_n (resp. K, IV, RS_n, RK, RI). De plus, chaque opération de la figure 4.6 est effectuée pour tout $0 \leq i < b$ où b est le nombre d'octets de l'état interne.

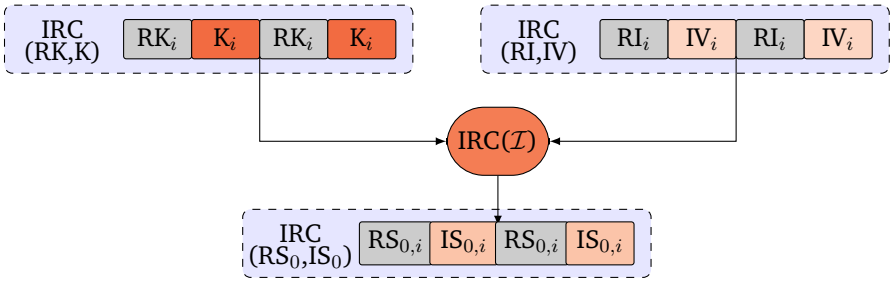


Figure 4.6.: L'IRC sur les chiffrements à flot - étape d'initialisation.

Ensuite, tous les octets de l'état interne IS_0 obtenu sont stockés dans des données temporaires, et les fonctions $\mathcal{E}$ et $\mathcal{T}$ sont appliquées à IRC(RS_0,S_0), également avec un seul flot d'instructions 32 bits opérant indépendamment sur chaque octet, respectivement noté IRC($\mathcal{E}$) et IRC($\mathcal{T}$).

La fonction IRC($\mathcal{E}$) génère le premier octet KS_0 de la suite chiffrante, concaténé avec l'octet RO de référence et leur copies, noté IRC(RO,KS_0), et la fonction IRC($\mathcal{T}$) permet d'obtenir un nouvel état interne IRC(RS_1,IS_1).

L'IRC effectue alors des comparaisons impliquant les copies et l'octet RO de référence stocké dans la mémoire du circuit afin de s'assurer que la valeur de KS_0 est correcte. Il stocke alors sa valeur à la place de l'octet de référence. La figure 4.7 illustre la génération du premier octet de la suite chiffrante protégée par l'IRC avec les mêmes notations que précédemment.

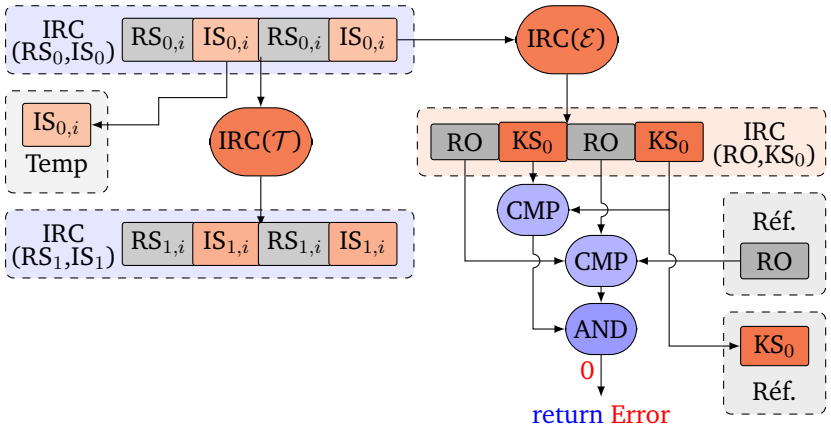


Figure 4.7.: L'IRC sur les chiffrements à flot - génération du premier octet de la suite chiffrante.

Cependant, même si la valeur du premier octet de la suite chiffrante est correcte, il est possible que IRC(RS_1,IS_1) contienne une faute puisque seulement une partie de l'état interne est généralement utilisée pour produire la suite chiffrante. Par conséquent, l'IRC effectue également des comparaisons sur IRC(RS_1,IS_1) afin de s'assurer que sa valeur est correcte.

Il remplace ensuite dans chaque mot de 32 bits obtenu les octets de l'état interne de référence RS_1 par IS_0 afin de protéger chaque nouvel octet de la suite chiffrante par son octet précédent comme illustré à la figure 4.8.

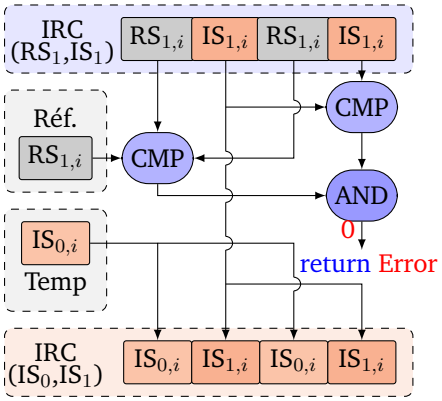


Figure 4.8.: L'IRC sur les chiffrements à flot - comparaisons.

Chaque mot de 32 bits généré par IRC($\mathcal{E}$) est alors composé de l'octet actuel de la suite chiffrante, de son octet précédent, et de leur copies.

L'IRC peut donc effectuer des comparaisons impliquant les différentes copies et l'octet précédent de la suite chiffrante, qui est stocké dans la mémoire du circuit. La figure 4.9 illustre la génération des autres octets de la suite chiffrante protégés par l'IRC.

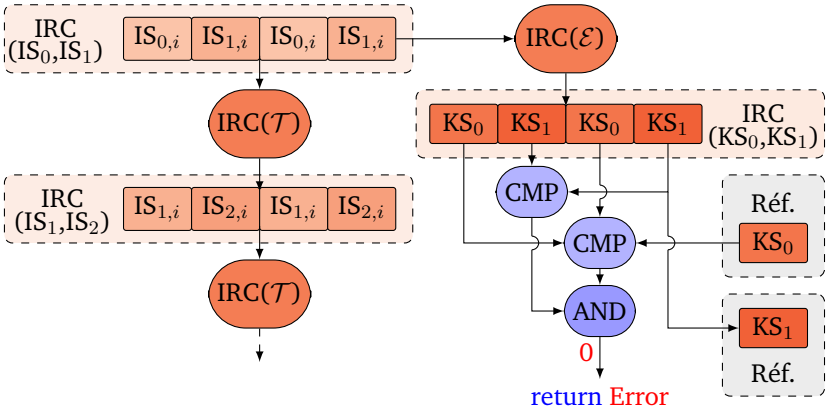


Figure 4.9.: L'IRC sur les chiffrements à flot - génération des octets suivants de la suite chiffrante.

Afin de déterminer quand retourner le premier octet KS_0 de la suite chiffrante, deux options sont possibles en fonction du niveau de sécurité requis :

- Retourner KS_0 après les comparaisons sur IRC(RO,KS_0) : on obtient une sécurité similaire au cas des chiffrements par blocs décrit précédemment.
- Retourner KS_0 après les comparaisons sur IRC(KS_0,KS_1) : on obtient une redondance temporelle supplémentaire puisque l'IRC génère KS_0 deux fois consécutivement et compare les valeurs générées. Cela requiert cependant une itération additionnelle des fonctions IRC($\mathcal{E}$) et IRC($\mathcal{T}$).

Il est également possible d'utiliser deux redondances temporelles en remplaçant dans chaque mot de IRC(IS_1,IS_2) le premier octet IS_1 par IS_0. Dans ce cas, l'IRC retourne KS_0 après les comparaisons sur IRC(KS_1,KS_2), ce qui demande deux itérations additionnelles des fonctions IRC($\mathcal{E}$) et IRC($\mathcal{T}$), et nécessite des données temporaires supplémentaires pour stocker KS_1 après les comparaisons sur IRC(KS_0,KS_1).

4.3 Comment implémenter efficacement le masquage en utilisant l'IRC ?

L'IRC offre la possibilité d'utiliser deux blocs de référence, mais seulement celui stocké entre les blocs de données est essentiel pour se prémunir des techniques d'injection de fautes actuelles, le second étant principalement utile pour satisfaire les prérequis que nous avons décrits en section 4.1.1 lorsqu'un seul ne suffit pas.

4.3.1 Masquage de premier ordre

Dans le cas où un seul bloc de référence permet de satisfaire les prérequis, nous proposons de remplacer le bloc de référence qui n'est pas stocké entre les blocs de données par un masque afin de mettre en œuvre un masquage de premier ordre similaire à la technique proposée en section 2.6.

Pour cela, il suffit tout d'abord de stocker dans chaque mot de 32 bits un octet P du message clair concaténé avec l'octet RP correspondant du bloc de référence, une copie de l'octet du message clair et un octet nul. Ensuite, un octet M d'un masque aléatoire est directement appliqué par un XOR sur chaque octet du mot de 32 bits, comme illustré à la figure 4.10.

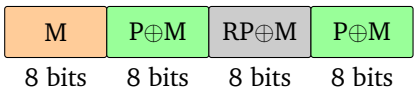


Figure 4.10.: Mot de 32 bits en entrée d'une implémentation masquée protégée par l'IRC.

La partie linéaire du schéma de chiffrement n'est alors pas modifiée puisqu'elle permet de calculer la valeur correcte du masque sans aucun surcoût.

Afin de calculer correctement la valeur du masque sur la partie non-linéaire, nous proposons d'utiliser l'une des deux techniques présentées en section 2.6 mais appliquées sur les octets au lieu des mots de 16 bits.

L'objectif est là encore de modifier le résultat de l'opération appliquée aux valeurs masquées afin d'obtenir un XOR entre la même opération appliquée aux valeurs non-masquées et un nouveau masque en utilisant une technique de masquage dérivée du schéma ISW. La première technique utilise un nouvel octet aléatoire tandis que la seconde utilise le résultat du XOR entre les deux masques des valeurs en entrée.

Il est à noter une fois de plus que la seconde technique n'est pas toujours applicable car il se peut qu'elle ne protège pas entièrement le schéma de chiffrement. En effet, nous rappelons qu'après chaque opération, chaque bit du masque est une combinaison linéaire de tous ses bits. Il faut donc s'assurer qu'après chaque opération, aucun bit n'est égal à la combinaison nulle, autrement cela signifie que les bits correspondant de l'état interne ne sont pas masqués. Il est également possible de remplacer $\mathrm{M}_a \oplus \mathrm{M}_b$ par seulement M_a ou M_b.

4.3.2 — Masquage d'ordre supérieur

Comme nous l'avons mentionné à la section 2.4.2, le masquage de premier ordre ne permet pas de se prémunir de toutes les attaques par observation, notamment des analyses différentielles d'ordre supérieur pour lesquelles il est nécessaire d'utiliser un masquage d'ordre supérieur.

Le masquage d'ordre d consiste alors à appliquer d masques différents sur le message clair, et à calculer la valeur correcte de chaque masque au cours de l'exécution, indépendamment des autres masques, en modifiant de façon similaire les opérations AND/OR, c'est-à-dire en générant et en appliquant d nouveaux masques de 8 bits aléatoires au lieu d'un seul dans le cas de la première technique, ou en appliquant le XOR entre tous les masques des valeurs en entrée pour la seconde technique.

Par contre, à la différence de la technique présentée à la section 2.6, le masquage n'utilise cette fois qu'un registre additionnel seulement par octet en entrée afin de stocker quatre masques supplémentaires, c'est-à-dire que calculer la valeur correcte de quatre masques spatialement (resp. temporellement) ne coûte que le double d'opérations en mémoire (resp. en temps).

Nous conseillons alors d'utiliser deux registres afin d'obtenir un vote majoritaire sur 4 blocs de données (c'est-à-dire une correction d'une faute affectant deux blocs avec des valeurs différentes), et un masquage d'ordre 2 avec des masques aléatoires, ce qui est souvent suffisant pour se prémunir de la plupart des attaques physiques.

En effet, pour chaque octet P des données en entrée, chacun des deux mots de 32 bits peut être composé d'un octet d'un des deux masques (M_0 pour le premier masque et M_1 pour le second), d'un octet RP du bloc de référence et de deux copies de l'octet P des données. Ensuite, M_0 et M_1 sont appliqués par un XOR sur chaque octet des deux mots comme l'illustre la figure 4.11.

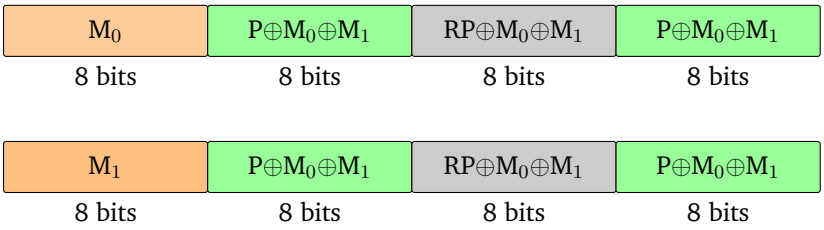


Figure 4.11.: Mots de 32 bits pour stocker un octet P des données en entrée d'une implémentation protégée par l'IRC avec un masquage d'ordre supérieur.

Il est possible d'utiliser deux blocs de référence différents afin de satisfaire les prérequis de l'IRC donnés à la section 4.1.1.

4.4 Tests pratiques conduits en laboratoire

Afin de tester la résistance de l'IRC face aux attaques par perturbation, nous l'avons déployée en mode détection de fautes sur deux architectures 32 bits différentes : un micro-contrôleur ARM Cortex-M3 sur lequel il n'existe aucune instruction SIMD et un micro-contrôleur ARM Cortex-M4 afin de bénéficier des instructions SIMD fournies dans son ISA.

Nous avons utilisé ces deux micro-contrôleurs car ils sont assez représentatifs des dispositifs utilisés pour l'IoT. Dans les deux cas, nous avons implémenté et exécuté des implémentations non sécurisées, puis protégées par l'IRC, du chiffrement par blocs PRIDE et du chiffrement à flot Trivium puisqu'il s'agit de deux schémas de chiffrement très performants en terme d'implémentation logicielle.

Nous avons ensuite comparé les performances que nous avons obtenues et nous avons analysé la résistance des implémentations protégées par l'IRC aux attaques physiques par des expérimentations pratiques. Il est à noter que dans les deux cas aucune autre contremesure n'a été implémentée.

4.4.1 – Implémentation de PRIDE protégée par l'IRC

Les spécifications de PRIDE sont données en section 2.3.2.1.

4.4.1.1 – Performances

Dans le tableau 4.1, nous comparons les performances et la taille du code de l'implémentation 8 bits de référence de PRIDE donnée en annexe A.4.3 et la même implémentation protégée par l'IRC avec le message clair de référence 0x19cb6e3cc15d254f, la clé de référence 0xb8f653fa05f4f9c39889ce4bb9015865 et le chiffré de référence correspondant 0x8b8cc44779935cf2.

Ce bloc de référence permet de satisfaire les prérequis, détaillés à la section 4.1.1, sur 97% des instructions de l'implémentation, avec une protection complète sur les deux premiers et derniers tours qui sont généralement les zones les plus ciblées lors d'une attaque par perturbation.

Il est possible d'obtenir un recouvrement complet en utilisant sur le second bloc de référence le message clair de référence 0x32c46c37168a7248, la clé de référence 0x485a895ac53577e7ffbd140564f5ca45 et le chiffré de référence correspondant 0x5b2569f55b45e69c.

Nous avons également inclus les performances de l'implémentation 32 bits de PRIDE donnée en annexe A.4.1 qui permet d'exploiter pleinement chacune des deux architectures.

Tableau 4.1.: Comparaisons de performances entre différentes implémentations de PRIDE. La taille du code est exprimée en octets et le coût est le nombre moyen de cycles d'horloge nécessaires pour chiffrer un octet.

	ARM Cortex-M3		ARM Cortex-M4	
	Taille	Coût	Taille	Coût
Implémentation 32 bits	464	356,5	416	350,5
Implémentation 8 bits	558	546,3	558	543,4
Implémentation IRC	886	788,0	636	638,5

PRIDE effectue 4 additions d'octets par tour. De ce fait, le micro-contrôleur ARM Cortex-M4 permet d'obtenir de meilleures performances grâce à l'instruction SIMD UADD8 disponible dans son ISA comme nous pouvons le voir dans le tableau 4.1.

Cependant, PRIDE effectue également de nombreuses rotations sur les octets, qui ne peuvent pas être remplacées par des instructions SIMD sur ces deux architectures, ce qui nécessite des instructions supplémentaires.

L'implémentation protégée par l'IRC effectue 4 exécutions de PRIDE en parallèle avec deux copies du bloc de données et deux copies du bloc de référence sur un processeur n'ayant qu'un seul cœur.

Pour obtenir le même niveau de sécurité sans notre approche, cela nécessiterait le plus souvent un processeur avec 4 cœurs afin d'effectuer 4 exécutions en parallèle, et demanderait en première approximation 4 fois plus de mémoire.

Ainsi, d'une part l'IRC permet d'appliquer ce type de contre-mesure sur un processeur n'ayant qu'un seul cœur, et d'autre part il permet de réduire considérablement la taille requise, au détriment du temps d'exécution.

4.4.1.2 – Résistance aux attaques par perturbation

Tout d'abord, nous avons utilisé une analyse des rayonnements électromagnétiques telle que décrite à la figure 2.8 afin de situer temporellement les deux derniers tours lors de l'exécution.

Ensuite, afin de tester la résistance aux attaques par perturbation de l'implémentation de PRIDE protégée par l'IRC en mode détection de fautes, nous avons utilisé le banc d'injection électromagnétique décrit à la figure 3.1 pour perturber son exécution sur le micro-contrôleur ARM Cortex-M3.

La durée de chaque impulsion électromagnétique était de 200ns, nous avons fait varier la tension par étape de 1V entre 180V et 219V et nous avons injecté 250 impulsions à chaque étape en ciblant le milieu du circuit intégré à différentes positions temporelles dans l'exécution des deux derniers tours de PRIDE.

Il y a plusieurs possibilités pour détecter des fautes grâce à l'IRC en mode détection de fautes : grâce à la redondance spatiale, grâce au premier bloc de référence ou grâce au second. De ce fait, nous avons examiné pour chaque faute la valeur du chiffré corrompu et comment la faute avait été détectée.

Au total, nous avons obtenu 4 823 fautes à partir des 10 000 injections. Parmi ces fautes, 3 107 étaient dues à un dysfonctionnement de l'UART et 1 716 auraient pu être exploitables. L'IRC nous a permis de détecter toutes les fautes : 3 444 ont été détectées grâce à la redondance spatiale, 4 805 grâce au premier bloc de référence et la totalité des 4 823 grâce au second.

Ainsi, l'IRC nous a permis de complètement contrecarrer un tel moyen d'injection de fautes, et seule l'utilisation du bloc de référence stocké entre les deux blocs de données a été nécessaire pour détecter toutes les fautes.

Pour rappel, l'injection électromagnétique avec cette approche nous avait permis de retrouver entièrement la clé secrète à partir de 2 000 injections dans le cas d'une implémentation non sécurisée de PRIDE.

4.4.2 – Implémentation de TRIVIUM protégée par l'IRC

TRIVIUM [DE CANNIÈRE 2006] est un chiffrement à flot recommandé par le projet eSTREAM [ECRYPT 2008] et standardisé par la norme ISO/IEC 29192-3 :2012 consacrée à la cryptographie légère.

4.4.2.1 – Le chiffrement à flot TRIVIUM

TRIVIUM utilise une clé secrète de 80 bits, un IV de 80 bits et manipule un état interne $s_1 \cdots s_{288}$ de 288 bits composé de 3 registres de tailles 93, 84 et 111 bits.

Sa fonction d'initialisation consiste à appliquer la fonction de transition 1152 fois (sans appliquer la fonction d'extraction, c'est-à-dire sans produire de sortie).

L'étape d'initialisation et la génération de la suite chiffrante $z_1 \cdots z_N$ de N bits sont effectuées en appliquant le code suivant :

```
1  (s1, s2, ··· , s93) ← (K0, ··· , K79, 0, ··· , 0)
2  (s94, s95, ··· , s177) ← (IV0, ··· , IV79, 0, ··· , 0)
3  (s178, s179, ··· , s288) ← (0, ··· , 0, 1, 1, 1)
4  for(i=1;i<1153+N;i++){
5      t1 ← s66 ⊕ s93;
6      t2 ← s162 ⊕ s177;
7      t3 ← s243 ⊕ s288;
8      if(i>1152){
9          zi−1152 ← t1 ⊕ t2 ⊕ t3;
10     }
11     t1 ← t1 ⊕ s91&s92 ⊕ s171;
12     t2 ← t2 ⊕ s175&s176 ⊕ s264;
13     t3 ← t3 ⊕ s286&s287 ⊕ s69;
14     (s1, s2, ··· , s93) ← (t3, s1, ··· , s92)
15     (s94, s95, ··· , s177) ← (t1, s94, ··· , s176)
16     (s178, s179, ··· , s288) ← (t2, s178, ··· , s287)
17 }
```

La figure 4.12 illustre le code précédent sous forme de schéma.

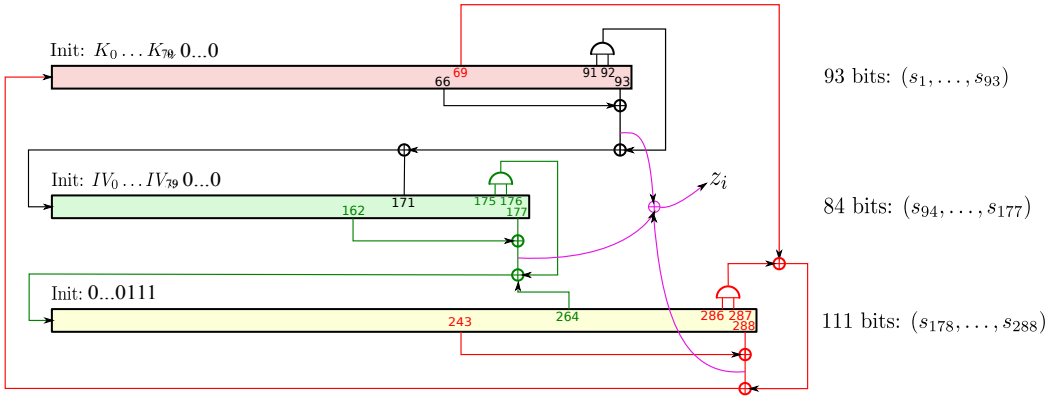


Figure 4.12.: Illustration de TRIVIUM [CANTEAUT et al. 2016a].

À ce jour, il n'existe pas d'attaque meilleure qu'une recherche exhaustive sur TRIVIUM. Par contre, il existe des attaques lorsque l'initialisation effectue moins de tours [DINUR et al. 2009], [AUMASSON et al. 2009], [FOUQUE et al. 2014], [TODO et al. 2017] et [LIU 2017].

De plus, certaines attaques par observation [FISCHER et al. 2007], [KAZMI et al. 2017] et par perturbation [HOJSÍK et al. 2008], [HU et al. 2012] ont également été proposées afin de retrouver la clé secrète utilisée par TRIVIUM.

4.4.2.2 – Performances

Dans le tableau 4.2, nous comparons les performances et empreintes de mémoire de l'implémentation 8 bits de TRIVIUM donnée en annexe A.7.2 et la même implémentation protégée par l'IRC avec l'IV de référence 0x73ad56fe566b227847f8, la clé de référence 0x4b752b672d363d93e7a3 et l'état interne de référence 0xd8b252aa20ecb9afb36cf7f4a42d1b1839fd86e63b68491fc392597c9477f22cd19562de.

Ce bloc de référence permet de satisfaire les prérequis sur 90% des instructions de l'implémentation. Il est possible d'obtenir un recouvrement complet en utilisant sur le second bloc de référence l'IV de référence 0xe475605c64c6d25b5f18, la clé de référence 0x0bcc0de165fa80897046 et l'état interne de référence 0xc606f18d33cec788fa981538f612d0cc24e440e2c901e50bd380c1920cb013fc70d05cf8.

Nous avons également inclus les performances de l'implémentation 32 bits de TRIVIUM donnée en annexe A.7.1 qui permet d'exploiter pleinement les registres de chacune des deux architectures.

Tableau 4.2.: Comparaisons de performances entre différentes implémentations de TRIVIUM. La taille est exprimée en octets et le coût est le nombre moyen de cycles d'horloge nécessaires pour chiffrer un octet.

	ARM Cortex-M3		ARM Cortex-M4	
	Taille	Coût	Taille	Coût
Implémentation 32 bits	292	42,0	292	41,5
Implémentation 8 bits	296	213,0	296	205,0
Implémentation IRC	448	293,0	448	286,0

TRIVIUM utilise principalement des opérateurs bit à bit, ainsi que de nombreux décalages dans le cas de l'implémentation protégée par l'IRC que nous avons utilisée, et tous ces décalages demandent un grand nombre d'instructions supplémentaires. De ce fait, les instructions SIMD disponibles dans l'ISA du micro-contrôleur ARM Cortex-M4 ne sont pas utilisées, ce qui implique des performances similaires entre les deux micro-contrôleurs.

D'une part l'IRC permet encore une fois d'appliquer de la redondance spatiale sur un processeur n'ayant qu'un seul cœur, et d'autre part il permet de réduire considérablement la taille requise, au détriment du temps d'exécution, qui cette fois nécessite un surcoût beaucoup plus important que dans le cas de PRIDE.

Cela vient du fait que l'implémentation 32 bits de TRIVIUM est beaucoup plus efficace que son implémentation 8 bits (elle effectue 4 fois moins d'opérations pour produire la même longueur de suite chiffrante).

4.4.2.3 – Résistance aux attaques par perturbation

Tout d'abord, nous avons utilisé l'analyse des rayonnements électromagnétiques de TRIVIUM donnée à la figure 4.13 afin d'identifier toutes les étapes de son exécution.

Il est à noter qu'après les 144 tours de l'étape d'initialisation de son implémentation 8 bits, nous avons généré 10 octets de suite chiffrante.

Ensuite, afin de tester la résistance de l'implémentation de TRIVIUM protégée par l'IRC en mode détection de fautes, nous avons une fois de plus utilisé le banc d'injection d'impulsions électromagnétiques décrit à la figure 3.1 afin de perturber son exécution sur le micro-contrôleur ARM Cortex-M3.

La durée de chaque impulsion électromagnétique était de 200ns, nous avons fait varier la tension par étape de 1V entre 180V et 219V et nous avons injecté 250 impulsions à chaque étape en ciblant le milieu du circuit intégré à différentes positions temporelles dans l'exécution de TRIVIUM.

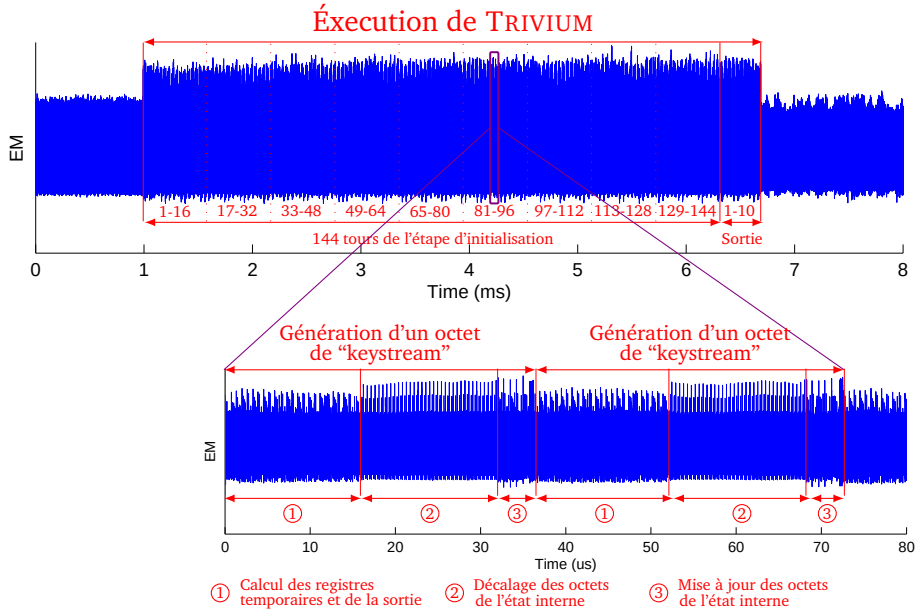


Figure 4.13.: Analyse des rayonnements électromagnétiques de TRIVIUM.

Il y a plusieurs possibilités pour détecter des fautes grâce à l'IRC en mode détection de fautes : grâce à la redondance spatiale, grâce à la première redondance temporelle ou grâce à la seconde. De ce fait, nous avons examiné pour chaque faute la valeur du chiffré corrompu et comment la faute avait été détectée.

Au total, nous avons obtenu 3 703 fautes à partir des 10 000 injections. Parmi ces fautes, 3 437 étaient dues à un dysfonctionnement de l'UART et 266 auraient pu être exploitables. L'IRC nous a permis de détecter toutes les fautes : 3 690 ont été détectées grâce à la redondance spatiale, 2 491 grâce à la première redondance temporelle et 2 915 grâce à la seconde.

Ainsi, l'IRC nous a permis une fois de plus de nous prémunir de l'injection électromagnétique, mais cette fois l'utilisation de la redondance spatiale et de la redondance temporelle (obtenue grâce à l'ajout d'un bloc de référence) a été nécessaire pour détecter toutes les fautes injectées.

Nous pensons que l'IRC permet de se prémunir de la plupart des moyens d'injection de fautes actuels, notamment ceux pouvant être utilisés dans le contexte de l'IoT. En effet, obtenir la même faute sur deux octets non consécutifs dans un registre est aujourd'hui infaisable de façon maîtrisée en pratique, même avec un laser.

L'utilisation de plus de blocs de données ne nous semble donc pas nécessaire à ce jour, bien que cette possibilité existe pour augmenter le niveau de sécurité.

4.5 Généralisation

Le principe de l'IRC peut être généralisé aux architectures ayant des registres de tailles différentes. Par exemple, l'IRC peut être déployé en mode correction de fautes afin de sécuriser deux blocs de données différents sur une architecture 64 bits en organisant les mots de 64 bits de la manière illustrée à la figure 4.14.

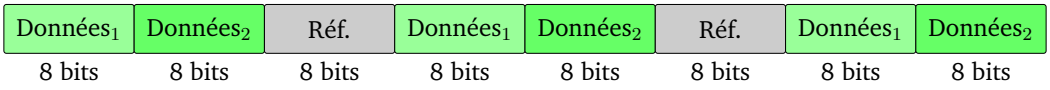


Figure 4.14.: Composition d'un mot de 64 bits en mode correction de fautes.

En effet, l'IRC exécute dans ce cas le schéma de chiffrement par un unique flot d'instructions 64 bits opérant indépendamment sur chaque octet, puis à la fin de l'exécution, effectue un vote majoritaire impliquant chaque copie et le message de référence stocké sur le circuit intégré.

L'IRC peut également être déployé en mode détection de fautes afin de sécuriser trois blocs de données différents sur une architecture 64 bits en organisant les mots de 64 bits de la manière illustrée à la figure 4.15.

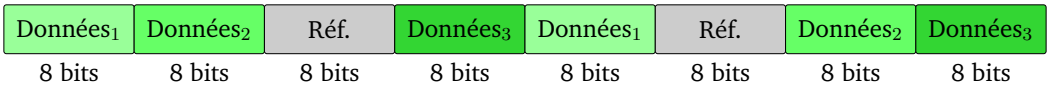


Figure 4.15.: Composition d'un mot de 64 bits en mode détection de fautes.

Il est également possible d'augmenter le niveau de sécurité en ne sécurisant qu'un seul bloc sur une architecture 64 bits, c'est-à-dire utiliser plus de copies des données et plus de blocs de référence, mais nous pensons que ce n'est pas nécessaire au regard des techniques d'injections de fautes actuelles.

Par contre, il est possible d'utiliser trois masques appliqués sur trois copies des données afin d'obtenir une implémentation sécurisée face à la plupart des attaques physiques actuelles, grâce au masquage d'ordre trois, au bloc de référence et au vote majoritaire sur la redondance spatiale.

En effet, le mot de 64 bits associé à l'octet P des données en entrée, sur lequel sont appliqués les trois masques M_0, M_1 et M_2, peut être composé de $M_1 \oplus M_2$, $M_0 \oplus M_2$, $M_0 \oplus M_1$, d'un octet RP du bloc de référence et de trois copies de l'octet P des données. Ensuite, M_0, M_1 et M_2 sont ajoutés par un XOR à chaque octet du mot de 64 bits comme l'illustre la figure 4.16 où $X = P \oplus M_0 \oplus M_1 \oplus M_2$ et $RX = RP \oplus M_0 \oplus M_1 \oplus M_2$.

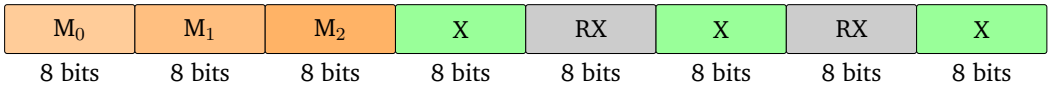


Figure 4.16.: Composition d'un mot de 64 bits en mode correction de fautes avec un masquage d'ordre trois.

De façon plus générale, l'IRC peut sécuriser n'importe quelle implémentation de m bits en étant déployée sur une architecture ℓm bits en appliquant le même concept à condition que $\ell \geq 3$ afin d'utiliser au moins deux copies des données et un bloc de référence toujours stocké entre les données.

Conclusion du chapitre

Dans ce chapitre, nous avons introduit l'IRC, une contre-mesure logicielle générique qui permet de déployer efficacement la redondance spatiale sur une architecture 32 bits à partir d'une implémentation 8 bits d'un schéma de chiffrement, dans le but de se prémunir des techniques d'injection de fautes.

Après avoir détaillé le principe général de l'IRC et son déploiement de façon générique sur les chiffrements symétriques, nous avons proposé une méthode permettant de combiner l'IRC avec une technique de masquage.

Nous avons validé la résistance d'une implémentation protégée par l'IRC face à l'injection électromagnétique, d'abord sur un chiffrement par blocs, puis sur un chiffrement à flot, par des mises en œuvre en laboratoire pour lesquelles nous avons détecté toutes les fautes.

Finalement, nous avons conclu par une généralisation de l'IRC à d'autres architectures, notamment aux architectures 64 bits sur lesquelles l'IRC peut être déployée avec différents niveaux de sécurité.

Nous avons pu constater lors de nos tests de performances que le surcoût de l'IRC dépend fortement de l'implémentation ciblée. En effet, l'utilisation d'instructions SIMD permet de réduire considérablement ce surcoût.

L'addition sur un octet est la seule instruction SIMD que nous avons pu exploiter, et il serait intéressant d'avoir d'autres instructions SIMD comme le décalage sur un octet afin de réduire davantage le surcoût de l'IRC.

De plus, l'IRC ne peut pas être déployée sur une implémentation utilisant des tableaux, sauf si une instruction dédiée est disponible dans l'ISA, et la plupart des schémas de chiffrement n'ont pas d'implémentation efficace sans tableau.

PRIDE est l'un des chiffrements par blocs les plus efficaces en termes d'implémentation logicielle comme nous l'avons déjà mentionné, et il est également l'un des schémas de chiffrement les plus adaptés à l'IRC, puisque son implémentation de référence n'utilise pas de tableau et que les seules opérations non intrinsèquement SIMD qu'elle utilise sont l'addition sur un octet et la rotation.

Cependant, PRIDE manipule des blocs de 64 bits, et la S-box sur 4 bits qu'il utilise n'offre pas une marge de sécurité suffisante [LALLEMAND et al. 2017].

Afin de proposer un schéma de chiffrement adapté à l'IRC, offrant une marge de sécurité suffisante vis-à-vis des cryptanalyses classiques, et ayant également une structure facilitant le masquage, nous avons orienté nos travaux sur la conception d'une structure entrelacée dont l'implémentation de référence n'utilise pas de tableau et effectue un maximum d'opérations intrinsèquement SIMD. Nous avons nommé ce nouveau schéma de chiffrement GARFIELD (pour "Guaranteed yet Affordable Resistance against Fault Injection for Embedded Lightweight Devices").

GARFIELD : un schéma de chiffrement pour l'IoT

Chapitre 5

> *Faites confiance à cette petite voix dans votre tête qui dit "ne serait-ce pas intéressant si…"; Et ensuite, faites-le.*
>
> — **Duane Michals**

Introduction

GARFIELD est un schéma de chiffrement que nous avons conçu pour répondre aux besoins de sécurité et de performance des objets connectés. Après avoir expliqué nos motivations et décrit les spécifications de GARFIELD, nous comparons ses performances avec d'autres schémas de chiffrement en fonction de plusieurs niveaux de sécurité. Nous étudions ensuite sa résistance vis-à-vis des cryptanalyses classiques connues dans la littérature, avant de conclure par des mises en œuvre d'attaques physiques que nous avons menées en laboratoire sur des implémentations protégées et non protégées de GARFIELD. Ces travaux ont fait l'objet d'un article que nous allons prochainement soumettre à un journal.

Sommaire

5.1 Pourquoi GARFIELD ? **112**
5.2 Spécifications **113**
5.2.1 Structure générale 113
5.2.2 Couche de substitution 113
5.2.3 Étage linéaire 117
5.2.4 Constantes de tour 122
5.2.5 Implémentations selon les besoins en sécurité 123
5.3 Performances **124**
5.4 Résistance aux attaques mathématiques **127**
5.4.1 Cryptanalyse linéaire et différentielle 128
5.4.2 Attaques d'ordre supérieur, par interpolation et variantes 129
5.4.3 Attaques par sous-espace invariant 130
5.4.4 Attaques par invariant non-linéaire 132
5.5 Mise en œuvre d'attaques physiques **133**
5.5.1 DFA menée en laboratoire sur GARFIELD 134
5.5.2 CEMA menée en laboratoire sur GARFIELD 137

5.1 Pourquoi GARFIELD ?

Les attaques physiques sont de réelles menaces à prendre en considération lorsque l'on étudie la sécurité des objets connectés. En effet, comme nous avons pu le constater dans les chapitres 2 et 3, il existe des chemins d'attaques pour retrouver la clé utilisée par un schéma de chiffrement quels que soient l'implémentation utilisée et le circuit intégré ciblé.

Le masquage, la redondance et les blocs de référence apparaissent alors parmi les meilleures solutions pour se prémunir de ces attaques. L'IRC, que nous avons introduit dans le chapitre 4, propose une technique pour combiner ces contre-mesures de façon à réduire le surcoût qu'elles engendrent.

L'IRC est particulièrement efficace lorsque l'implémentation ciblée utilise un maximum d'instructions qui sont applicables efficacement de façon SIMD, c'est-à-dire qui demandent peu d'instructions supplémentaires pour être effectuées en parallèle, sur les octets d'un mot de 32 ou 64 bits.

Malheureusement, comme nous l'avons mentionné à la fin du chapitre 4, la plupart des schémas de chiffrement légers existant aujourd'hui ne possèdent pas d'implémentation adéquate pour l'IRC. PRIDE est l'un des schémas de chiffrement les mieux adaptés à l'IRC, mais il manipule des blocs de 64 bits, et la S-box sur 4 bits qu'il utilise n'offre pas une marge de sécurité suffisante [Lallemand et al. 2017].

Sa structure entrelacée rend efficace le parallélisme par tranches de bits sur sa couche de substitution, une technique d'implémentation qui permet notamment de faciliter le masquage.

C'est pour toutes ces raisons que nous avons orienté nos travaux sur la conception d'une structure entrelacée offrant une marge de sécurité suffisante vis-à-vis des cryptanalyses classiques, et dont l'implémentation utilise un maximum d'instructions qui sont facilement applicables de façon SIMD.

Nous avons ainsi obtenu une structure légère résistante aux cryptanalyses classiques et adéquate pour combiner l'IRC et le masquage.

Nous avons nommé ce nouveau schéma de chiffrement GARFIELD (pour "Guaranteed yet Affordable Resistance against Fault Injection for Embedded Lightweight Devices"), et nous en avons proposé plusieurs implémentations avec différents niveaux de sécurité pour les applications de l'IoT.

5.2 Spécifications

GARFIELD a une structure très proche de PRIDE, dont les spécifications sont données à la section 2.3.2.1, à la différence qu'il n'effectue pas un blanchiment des données avec une sous-clé indépendante des autres, et que le dernier tour de sa fonction de chiffrement n'est pas modifié, c'est-à-dire que l'étage linéaire est bien appliqué.

5.2.1 Structure générale

GARFIELD est un chiffrement par blocs itératif composé de 11 tours. Il prend en entrée un message clair de 128 bits et utilise une structure entrelacée.

Son état interne peut donc être représenté par une matrice binaire avec 8 lignes et 16 colonnes, et chaque tour est formé des étapes suivantes :

- Additionner par un XOR la clé secrète et la constante de tour à l'entrée du tour,
- Appliquer la S-box $\mathcal{S}$ sur chaque colonne de l'état interne,
- Pour $0 \leq \omega \leq 7$, appliquer la matrice $\mathcal{L}_\omega$ sur la ligne ω de l'état interne.

Afin de chiffrer un message clair M, la fonction de chiffrement applique les 11 tours que nous venons de décrire, en utilisant une première constante de tour nulle, puis effectue un XOR avec la clé secrète afin d'obtenir le chiffré C.

La figure 5.1 illustre la structure générale de GARFIELD.

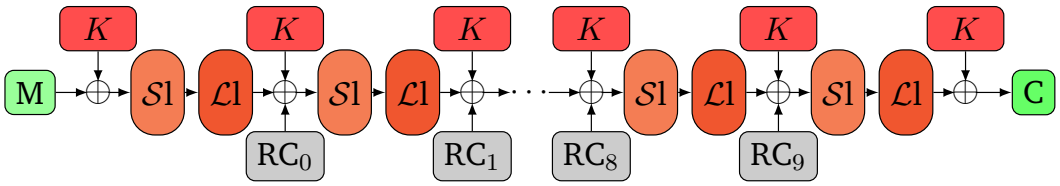


Figure 5.1.: Structure générale de GARFIELD.

Dans cette section, nous décrivons la couche de substitution $\mathcal{S}$l, l'étage linéaire $\mathcal{L}$l et les constantes de tour $\mathrm{RC}_0, \cdots, \mathrm{RC}_9$ utilisés dans GARFIELD.

5.2.2 Couche de substitution

La couche de substitution dans GARFIELD consiste à appliquer la même S-box sur les 16 colonnes de l'état interne, ce qui permet de rendre efficace la technique du parallélisme par tranches de bits comme nous l'avons décrit à la section 2.3.1.2, et donc de ne pas utiliser de tableau afin de pouvoir déployer plus efficacement l'IRC.

Pour augmenter l'efficacité de cette technique, nous avons cherché à diminuer au maximum le nombre d'opérations AND/OR/XOR nécessaires pour exprimer les bits en sortie de la S-box en fonction des bits en entrée, tout en conservant de bonnes propriétés cryptographiques.

5.2.2.1 – Propriétés cryptographiques d'une S-box

Soit $\mathcal{S} : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$ une S-box sur n bits.

Définition 1. *La* linéarité *de $\mathcal{S}$ est définie par :*

$$\mathcal{L}(\mathcal{S}) = \max_{a,b \neq 0 \in \mathbb{F}_2^n} |2 \times \#\{x \in \mathbb{F}_2^n : a \cdot x = b \cdot \mathcal{S}(x)\} - 2^n| \ .$$

*L'*uniformité différentielle *de $\mathcal{S}$ est définie par :*

$$\Delta(\mathcal{S}) = \max_{a \neq 0, b \in \mathbb{F}_2^n} \#\{x \in \mathbb{F}_2^n : \mathcal{S}(x \oplus a) \oplus \mathcal{S}(x) = b\} \ .$$

Ces quantités mesurent la résistance offerte par une S-box contre la cryptanalyse linéaire [Matsui 1994] et la cryptanalyse différentielle [Biham et al. 1991] : plus leurs valeurs sont faibles, plus le schéma de chiffrement est résistant.

L'uniformité différentielle correspond à la plus grande valeur dans la table de distribution des différences, c'est-à-dire correspond au nombre maximum de candidats valides qu'il est possible d'obtenir à partir d'une différentielle sur la S-box comme nous l'avons mentionné à la section 3.2.2.

Une faible uniformité différentielle peut donc sembler défavorable dans le contexte de la DFA car cela assure que l'ensemble de candidats qu'un attaquant pourra obtenir à partir d'une différentielle est toujours de petite taille. Cependant, comme nous avons pu l'observer à la section 3.3, deux ensembles de candidats sont souvent suffisants pour retrouver la valeur correcte de la clé secrète en sortie d'une S-box, et cela quelles que soient leurs tailles.

De plus, le fait que la valeur de l'uniformité différentielle soit faible a tendance à augmenter la résistance à la cryptanalyse différentielle puisque cela garantit que la probabilité d'une différentielle sur un tour est faible. De ce fait, il est largement préférable d'augmenter la résistance d'un schéma de chiffrement face à la cryptanalyse différentielle, même si cela diminue le nombre maximum de candidats qu'il est possible d'obtenir à partir d'une différentielle.

5.2.2.2 – Construction d'une S-box 8 bits à partir de S-boxes 4 bits

Il est possible d'obtenir une S-box légère sur 8 bits avec de bonnes propriétés cryptographiques en utilisant des S-boxes sur 4 bits dans un réseau de Feistel sur 3 tours [CANTEAUT et al. 2016b].

La figure 5.2 illustre une S-box sur 8 bits involutive (c'est-à-dire égale à son inverse) définie à partir d'un réseau de Feistel sur 3 tours utilisant deux S-boxes sur 4 bits $\mathcal{S}_1$ et $\mathcal{S}_2$. L'octet en entrée (resp. en sortie) de la S-box obtenue est noté $x = x_0 \cdots x_7$ (resp. $y = y_0 \cdots y_7$).

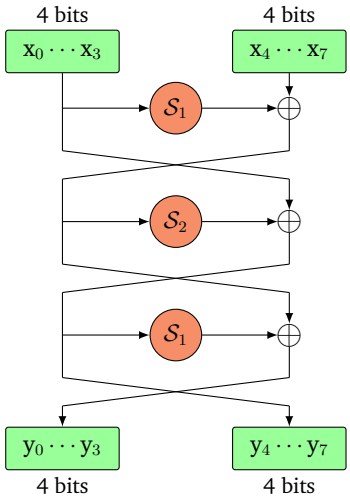


Figure 5.2.: Réseau de Feistel sur 3 tours.

Les meilleures propriétés cryptographiques pour ce type de construction peuvent être obtenues en utilisant une fonction APN (pour "Almost Perfect Nonlinear") pour $\mathcal{S}_1$, c'est-à-dire une fonction telle que $\Delta(\mathcal{S}_1) = 2$, et une permutation sur 4 bits pour $\mathcal{S}_2$ telle que $\Delta(\mathcal{S}_2) = 4$ [CANTEAUT et al. 2015, Th. 9]. Cela vient du fait qu'il n'existe pas de permutation APN sur $\mathbb{F}_2^4$.

Une telle S-box sur 8 bits légère avec $\mathcal{L}(\mathcal{S}) = 64$ et $\Delta(\mathcal{S}) = 8$ est décrite dans [CANTEAUT et al. 2016b]. Elle peut être exécutée avec 12 opérations AND/OR et 26 opérations XOR, et implémentée avec seulement 8 instructions AND/OR et 13 instructions XOR.

5.2.2.3 – Spécifications de la couche de substitution de GARFIELD

La S-box sur 8 bits qui est appliquée à chaque colonne de l'état interne dans GARFIELD est proche de celle donnée dans [CANTEAUT et al. 2016b], avec des propriétés cryptographiques similaires.

Nous l'avons également obtenue à partir d'un réseau de Feistel sur 3 tours utilisant une fonction APN $\mathcal{S}_1$ et une permutation $\mathcal{S}_2$ d'uniformité différentielle 4.

Nous avons utilisé pour $\mathcal{S}_1$ et $\mathcal{S}_2$ deux S-boxes sur 4 bits ayant des opérations en commun afin de réduire le nombre d'instructions nécessaires pour la stocker. Le tableau 5.1 donne les valeurs de $\mathcal{S}_1$ et $\mathcal{S}_2$.

Tableau 5.1.: Valeurs des S-boxes utilisées par la couche de substitution dans GARFIELD.

	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
$\mathcal{S}_1$	0x0	0x9	0xb	0xa	0xa	0xc	0x0	0xe	0x8	0x0	0x8	0x7	0xb	0xc	0xa	0xa
$\mathcal{S}_2$	0x0	0x8	0xb	0x3	0xa	0x4	0x9	0x7	0x6	0xf	0xe	0x1	0xd	0x2	0xc	0x5

La figure 5.3 illustre les portes logiques requises pour implémenter $\mathcal{S}_1$ et $\mathcal{S}_2$. L'entrée (resp. la sortie) de chaque S-box est notée $x = x_0 \cdots x_3$ (resp. $y = y_0 \cdots y_3$).

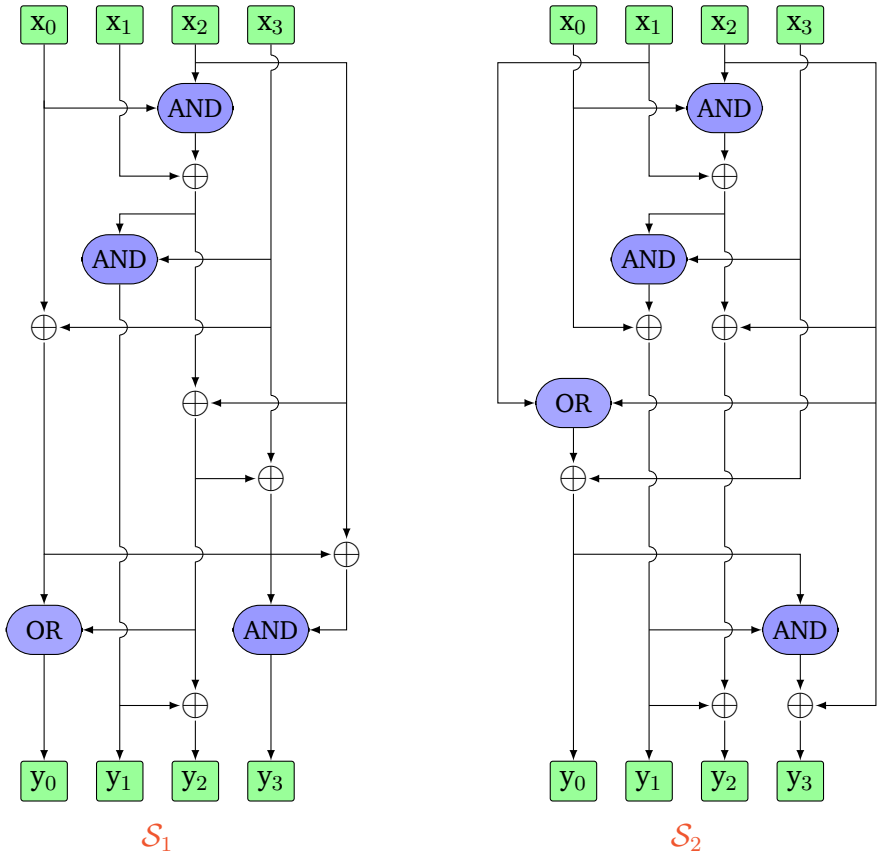


Figure 5.3.: S-boxes utilisées par la couche de substitution dans GARFIELD.

Finalement, la table de vérité de la S-box utilisée par GARFIELD est donnée par le tableau 5.2, et elle a les propriétés cryptographiques suivantes : $\mathcal{L}(\mathcal{S}) = 64$ et $\Delta(\mathcal{S}) = 8$. De plus, elle peut être exécutée avec 12 opérations AND/OR et 30 opérations XOR, et implémentée avec seulement 6 instructions AND/OR et 13 instructions XOR (et son inverse a les mêmes propriétés puisqu'il s'agit d'une involution).

Tableau 5.2.: S-box utilisée par GARFIELD.

$\mathcal{S}$	0x-0	0x-1	0x-2	0x-3	0x-4	0x-5	0x-6	0x-7	0x-8	0x-9	0x-a	0x-b	0x-c	0x-d	0x-e	0x-f
0x0-	0x00	0x3e	0x13	0x6d	0x2a	0x48	0x06	0x9b	0x2e	0x09	0x3c	0xcc	0x7d	0xc0	0xa0	0x31
0x1-	0x98	0x67	0x1c	0x02	0x75	0x37	0xbe	0x42	0x77	0x81	0x55	0xd2	0x12	0x7f	0x1f	0x1e
0x2-	0x5b	0xb0	0x5d	0x45	0x32	0xe9	0xed	0x4a	0xa6	0x92	0x04	0xff	0x33	0x71	0x08	0x89
0x3-	0x93	0x0f	0x24	0x2c	0xeb	0xf0	0xf5	0x15	0xb9	0xba	0xcb	0xd1	0x0a	0x78	0x01	0x90
0x4-	0xca	0x73	0x17	0x4f	0x68	0x23	0xfc	0xa9	0x05	0xe3	0x27	0xb8	0xf9	0xfb	0x52	0x43
0x5-	0xf4	0x7c	0x4e	0xf6	0x61	0x1a	0xe2	0xb6	0x5a	0x9d	0x58	0x20	0xe0	0x22	0x8b	0x7a
0x6-	0x69	0x54	0xd0	0x6e	0xc9	0xdb	0xcf	0x11	0x44	0x60	0xe6	0xb3	0x7e	0x03	0x63	0x72
0x7-	0x97	0x2d	0x6f	0x41	0xd6	0x14	0xe1	0x18	0x3d	0xde	0x5f	0x7b	0x51	0x0c	0x6c	0x1d
0x8-	0x86	0x19	0xa1	0xa8	0x8f	0xef	0x80	0xce	0x88	0x2f	0x8a	0x5e	0x9f	0xa7	0xe5	0x84
0x9-	0x3f	0x91	0x29	0x30	0xd7	0xc1	0x99	0x70	0x10	0x96	0xa2	0x07	0xc7	0x59	0xfd	0x8c
0xa-	0x0e	0x82	0x9a	0xa3	0xa4	0xdc	0x28	0x8d	0x83	0x47	0xb1	0xd9	0xb4	0xc4	0xae	0xaf
0xb-	0x21	0xaa	0xb2	0x6b	0xac	0xc2	0x57	0xf3	0x4b	0x38	0x39	0xc6	0xbc	0xfa	0x16	0xf7
0xc-	0x0d	0x95	0xb5	0xea	0xad	0xc5	0xbb	0x9c	0xf2	0x64	0x40	0x3a	0x0b	0xcd	0x87	0x66
0xd-	0x62	0x3b	0x1b	0xd4	0xd3	0xdd	0x74	0x94	0xec	0xab	0xfe	0x65	0xa5	0xd5	0x79	0xf8
0xe-	0x5c	0x76	0x56	0x49	0xee	0x8e	0x6a	0xe7	0xf1	0x25	0xc3	0x34	0xd8	0x26	0xe4	0x85
0xf-	0x35	0xe8	0xc8	0xb7	0x50	0x36	0x53	0xbf	0xdf	0x4c	0xbd	0x4d	0x46	0x9e	0xda	0x2b

À titre de comparaison, la S-box utilisée par l'AES satisfait $\mathcal{L}(\mathcal{S}) = 32$ et $\Delta(\mathcal{S}) = 4$, mais demande 32 opérations AND/OR et 83 opérations XOR pour être exécutée [CANTEAUT et al. 2016b].

5.2.3 Étage linéaire

L'étage linéaire dans GARFIELD correspond à la concaténation de 8 L-boxes opérant sur les lignes de l'état interne. Chaque L-box est définie par une matrice binaire de 16 lignes et 16 colonnes, et est décrite par une séquence d'opérations sur les deux octets de la ligne traitée de l'état interne.

Là encore, nous avons cherché à utiliser au maximum des opérations qui peuvent être facilement implémentées de façon SIMD, et à diminuer au plus le nombre d'opérations requises pour appliquer les 8 L-boxes, tout en conservant de bonnes propriétés cryptographiques pour chacune.

5.2.3.1 Propriétés cryptographiques d'une L-box

Un premier critère de diffusion pour un tel étage linéaire est le suivant : si la superposition de toutes les matrices n'a aucun bit égal à 0, alors une diffusion maximale est garantie dans le sens que la sortie de chaque S-box au tour r influence l'entrée de chaque S-box au tour $r + 1$ [ALBRECHT et al. 2014a].

Cependant, bien que les auteurs dans [ALBRECHT et al. 2014a] aient souligné l'intérêt cryptographique de cette propriété, il n'y a actuellement aucun schéma de chiffrement ayant atteint cet objectif à notre connaissance.

Par exemple, la superposition de toutes les matrices utilisées par l'étage linéaire de PRIDE a 144 bits égaux à 0 [ALBRECHT et al. 2014b].

Le choix de l'étage linéaire affecte également la résistance du schéma de chiffrement à la cryptanalyse linéaire et différentielle, qui peut être quantifiée par le nombre de branchements ("branch number" en anglais) de l'étage linéaire [DAEMEN 1995] par rapport à l'alphabet de la S-box, qui est l'ensemble des mots binaires en entrée de cette dernière. Alors, plus le nombre de branchements est élevé, plus le schéma de chiffrement est résistant.

Définition 2. *Le* nombre de branchements différentiels *d'un étage linéaire L sur $(\mathbb{F}_2^\omega)^c$ par rapport à $\mathbb{F}_2^\omega$ est défini par :*

$$\mathcal{B}_d(L) = \min_{v \in (\mathbb{F}_2^\omega)^c, v \neq 0} wt(v) + wt(L(v))$$

et son nombre de branchements linéaires *est défini par :*

$$\mathcal{B}_l(L) = \min_{v \in (\mathbb{F}_2^\omega)^c, v \neq 0} wt(v) + wt(L^T(v))$$

où L^T est la transposée de L et $wt(v)$ est le poids de Hamming de v, c'est-à-dire le nombre de coordonnées non nulles de v, vu comme un mot de c éléments de $\mathbb{F}_2^\omega$.

Le nombre de branchements différentiels (resp. linéaires) correspond au nombre minimum de S-boxes actives dans une caractéristique différentielle (resp. un chemin linéaire) sur 2 tours. Dans le cas d'une structure entrelacée, le nombre de branchements de l'étage linéaire par rapport à $\mathbb{F}_2^\omega$ est égal au nombre de branchements minimum des L-boxes par rapport à $\mathbb{F}_2$ [ALBRECHT et al. 2014b, théorème 1].

La valeur la plus élevée qu'il est possible d'obtenir pour le nombre de branchements différentiels et pour le nombre de branchements linéaires dans le cas d'une matrice binaire de 16 lignes et 16 colonnes est de 8 [ALBRECHT et al. 2014b, annexe E].

5.2.3.2 – Méthode de construction pour les L-boxes

Chaque L-box utilisée par l'étage linéaire dans GARFIELD est décrite par une séquence d'opérations qui dépend de la ligne traitée de l'état interne.

Les opérations sont choisies parmi le XOR et l'opération rol8_d qui applique une rotation binaire de d bits à gauche dans un octet.

La figure 5.4 illustre les opérations rol8_3 et rol8_5. L'entrée (resp. la sortie) de chaque opération est notée $x{=}x_0 \cdots x_7$ (resp. $y{=}y_0 \cdots y_7$).

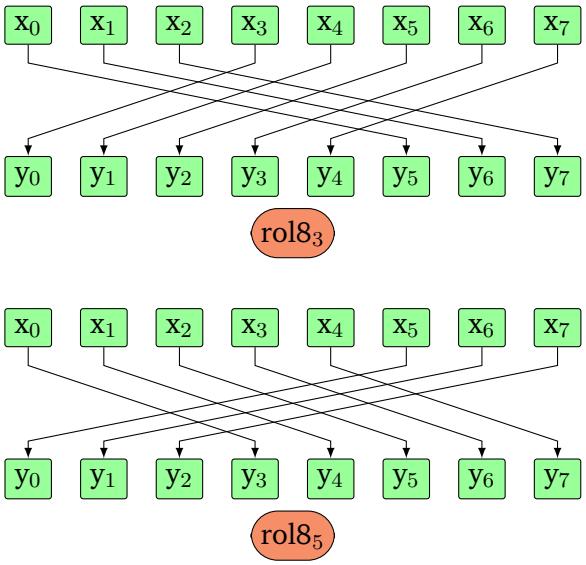


Figure 5.4.: Opérations rol8_3 et rol8_5 utilisées par l'étage linéaire de GARFIELD.

Alors, le tableau 5.3 donne les instructions ARM nécessaires pour appliquer les opérations rol8_3 et rol8_5 d'une façon SIMD, c'est-à-dire simultanément sur les 4 octets d'un mot de 32 bits, respectivement notées 4rol8_3 et 4rol8_5.

Tableau 5.3.: Instructions ARM nécessaires pour appliquer les opérations rol8_3 et rol8_5 d'une façon SIMD sur un mot de 32 bits.

$y \leftarrow 4\text{rol8}_3(x)$
AND Temp, 0xf8f8f8f8, x, LSL #3
AND y, 0x07070707, x, LSR #5
ORR y, y, Temp

$y \leftarrow 4\text{rol8}_5(x)$
AND Temp, 0xe0e0e0e0, x, LSL #5
AND y, 0x1f1f1f1f, x, LSR #3
ORR y, y, Temp

Il est à noter qu'une instruction de rotation SIMD permettrait de réduire considérablement le coût d'une implémentation de GARFIELD protégée par l'IRC.

Nous avons alors cherché à minimiser le nombre d'instructions requises pour stocker le code de l'étage linéaire, et également le surcoût nécessaire pour implémenter son inverse puisque l'IRC permet de se prémunir des injections de fautes même lorsque l'attaquant a accès à la fois à la fonction de chiffrement et à la fonction de déchiffrement (ce qui n'est pas le cas du masquage comme nous l'avons expliqué en section 3.4), ce qui peut être utile pour certaines applications.

5.2.3.3 – Spécifications de l'étage linéaire de GARFIELD

La figure 5.5 illustre la fonction appliquée par l'étage linéaire dans GARFIELD à la ligne ω de l'état interne, avec $0 \leq \omega \leq 7$. L'entrée (resp. la sortie) de la matrice utilisée est notée $x = x_0 \cdots x_{15}$ (resp. $y = y_0 \cdots y_{15}$).

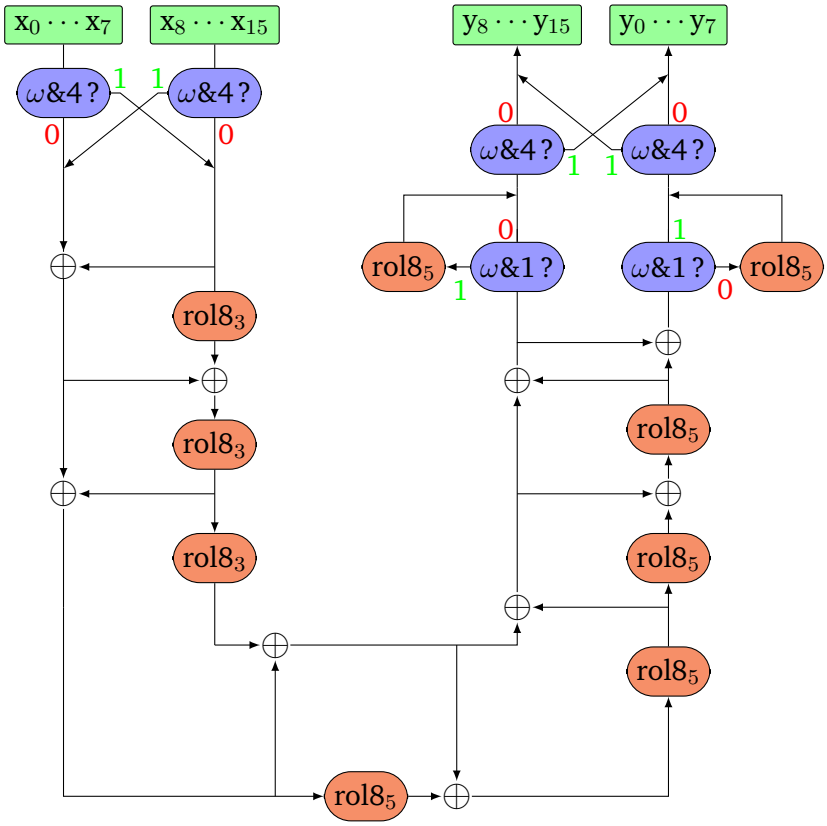


Figure 5.5.: Fonction linéaire appliquée par l'étage linéaire dans GARFIELD sur la ligne ω avec $0 \leq \omega \leq 7$.

Les matrices utilisées par l'étage linéaire dans GARFIELD sont alors les suivantes :

$$\mathrm{L}_0 = \mathrm{L}_2$$

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

$$\mathrm{L}_1 = \mathrm{L}_3$$

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

$$L_4 = L_6$$
$$\begin{pmatrix}
0&0&1&0&0&0&1&0&1&0&0&0&1&1&1&1\\
0&0&0&1&0&0&0&1&1&1&0&0&0&1&1&1\\
1&0&0&0&1&0&0&0&1&1&1&0&0&0&1&1\\
0&1&0&0&0&1&0&0&1&1&1&1&0&0&0&1\\
0&0&1&0&0&0&1&0&1&1&1&1&1&0&0&0\\
0&0&0&1&0&0&0&1&0&1&1&1&1&1&0&0\\
1&0&0&0&1&0&0&0&0&0&1&1&1&1&1&0\\
0&1&0&0&0&1&0&0&0&0&0&1&1&1&1&1\\
1&0&1&0&0&1&1&1&0&1&1&1&1&0&1&1\\
1&1&0&1&0&0&1&1&1&0&1&1&1&1&0&1\\
1&1&1&0&1&0&0&1&1&1&0&1&1&1&1&0\\
1&1&1&1&0&1&0&0&0&1&1&0&1&1&1&1\\
0&1&1&1&1&0&1&0&1&0&1&1&0&1&1&1\\
0&0&1&1&1&1&0&1&1&1&0&1&1&0&1&1\\
1&0&0&1&1&1&1&0&1&1&1&0&1&1&0&1\\
0&1&0&0&1&1&1&1&1&1&1&1&1&0&1&0
\end{pmatrix}$$

$$L_5 = L_7$$
$$\begin{pmatrix}
0&1&1&1&1&0&1&1&1&0&1&0&0&1&1&1\\
1&0&1&1&1&1&0&1&1&1&0&1&0&0&1&1\\
1&1&0&1&1&1&1&0&1&1&1&0&1&0&0&1\\
0&1&1&0&1&1&1&1&1&1&1&1&1&0&1&0\\
1&0&1&1&0&1&1&1&0&1&1&1&0&1&0&0\\
1&1&0&1&1&0&1&1&0&0&1&1&1&1&0&1\\
1&1&1&0&1&1&0&1&1&0&0&1&1&1&1&0\\
1&1&1&1&0&1&1&0&0&1&0&0&1&1&1&1\\
1&0&0&0&1&1&1&1&0&0&1&0&0&0&1&0\\
1&1&0&0&0&1&1&1&0&0&0&1&0&0&0&1\\
1&1&1&0&0&0&1&1&1&0&0&0&1&0&0&0\\
1&1&1&1&0&0&0&1&0&1&0&0&0&1&0&0\\
1&1&1&1&1&0&0&0&0&0&1&0&0&0&1&0\\
0&1&1&1&1&1&0&0&0&0&0&1&0&0&0&1\\
0&0&1&1&1&1&1&0&1&0&0&0&1&0&0&0\\
0&0&0&1&1&1&1&1&0&1&0&0&0&1&0&0
\end{pmatrix}$$

Et les matrices utilisées par l'inverse de l'étage linéaire dans GARFIELD sont :

$$L_0^{-1} = L_2^{-1}$$
$$\begin{pmatrix}
1&1&1&1&0&1&1&0&0&0&0&1&1&1&1&1\\
0&1&1&1&1&0&1&1&1&0&0&0&1&1&1&1\\
1&0&1&1&1&1&0&1&1&1&0&0&0&1&1&1\\
1&1&0&1&1&1&1&0&1&1&1&0&0&0&1&1\\
0&1&1&0&1&1&1&1&1&1&1&1&1&0&0&1\\
1&0&1&1&0&1&1&1&1&1&1&1&1&1&0&0\\
1&1&0&1&1&0&1&1&0&1&1&1&1&1&0&0\\
1&1&1&0&1&1&0&1&0&0&1&1&1&1&1&0\\
0&0&1&1&1&1&0&1&0&0&0&1&0&0&0&1\\
1&0&0&1&1&1&1&0&1&0&0&0&1&0&0&0\\
0&1&0&0&1&1&1&1&0&1&0&0&0&1&0&0\\
1&0&1&0&0&1&1&1&0&0&1&0&0&0&1&0\\
1&1&0&1&0&0&1&1&0&0&0&1&0&0&0&1\\
1&1&1&0&1&0&0&1&1&0&0&0&1&0&0&0\\
1&1&1&1&0&1&0&0&0&1&1&0&0&1&0&0\\
0&1&1&1&1&0&1&0&0&0&0&1&1&0&1&0
\end{pmatrix}$$

$$L_1^{-1} = L_3^{-1}$$
$$\begin{pmatrix}
1&1&0&1&1&1&1&0&1&1&1&0&0&0&1&1\\
0&1&1&0&1&1&1&1&1&1&1&1&1&0&0&1\\
1&0&1&1&0&1&1&1&1&1&1&1&1&1&0&0\\
1&1&0&1&1&0&1&1&0&1&1&1&1&1&0&0\\
1&1&1&0&1&1&0&1&0&0&1&1&1&1&1&0\\
1&1&1&1&0&1&1&0&0&0&0&1&1&1&1&1\\
0&1&1&1&1&0&1&1&1&0&0&0&1&1&1&1\\
1&0&1&1&1&1&0&1&1&1&0&0&0&1&1&1\\
1&1&1&0&1&0&0&1&1&0&0&0&1&0&0&0\\
1&1&1&1&0&1&0&0&0&1&0&0&0&1&0&0\\
0&1&1&1&1&0&1&0&0&0&1&0&0&0&1&0\\
0&0&1&1&1&1&0&1&0&0&0&1&0&0&0&1\\
1&0&0&1&1&1&1&0&1&0&0&0&1&0&0&0\\
0&1&0&0&1&1&1&1&0&1&0&0&0&1&0&0\\
1&0&1&0&0&1&1&1&0&0&1&0&0&0&1&0\\
1&1&0&1&0&0&1&1&0&0&0&1&0&0&0&1
\end{pmatrix}$$

$$L_4^{-1} = L_6^{-1}$$
$$\begin{pmatrix}
0&0&0&1&0&0&0&1&0&0&1&1&1&1&0&1\\
1&0&0&0&1&0&0&0&1&0&0&1&1&1&1&0\\
0&1&0&0&0&1&0&0&0&1&0&0&1&1&1&1\\
0&0&1&0&0&0&1&0&1&0&1&0&0&1&1&1\\
0&0&0&1&0&0&0&1&1&1&0&1&0&0&1&1\\
1&0&0&0&1&0&0&0&1&1&1&0&1&0&0&1\\
0&1&0&0&0&1&0&0&1&1&1&1&0&1&0&0\\
0&0&1&0&0&0&1&0&0&1&1&1&1&0&1&0\\
0&0&0&1&1&1&1&1&1&1&1&1&1&1&0&1&1&0\\
1&0&0&0&1&1&1&1&0&1&1&1&1&0&1&1\\
1&1&0&0&0&1&1&1&1&0&1&1&1&1&0&1\\
1&1&1&0&0&0&1&1&1&1&0&1&1&1&1&0\\
1&1&1&1&0&0&0&1&0&1&1&0&1&1&1&1\\
1&1&1&1&1&0&0&0&1&0&1&1&0&1&1&1\\
0&1&1&1&1&1&0&0&1&1&0&1&1&0&1&1\\
0&0&1&1&1&1&1&0&1&1&1&0&1&1&0&1
\end{pmatrix}$$

$$L_5^{-1} = L_7^{-1}$$
$$\begin{pmatrix}
1&0&0&0&1&0&0&0&1&1&1&0&1&0&0&1\\
0&1&0&0&0&1&0&0&1&1&1&1&0&1&0&0\\
0&0&1&0&0&0&1&0&0&1&1&1&1&0&1&0\\
0&0&0&1&0&0&0&1&0&0&1&1&1&1&0&1\\
1&0&0&0&1&0&0&0&1&0&0&1&1&1&1&0\\
0&1&0&0&0&1&0&0&0&1&0&0&1&1&1&1\\
0&0&1&0&0&0&1&0&1&0&1&0&0&1&1&1\\
0&0&0&1&0&0&0&1&1&1&0&1&0&0&1&1\\
1&1&1&0&0&0&1&1&1&1&0&1&1&1&1&0\\
1&1&1&1&0&0&0&1&0&1&1&0&1&1&1&1\\
1&1&1&1&1&0&0&0&1&0&1&1&0&1&1&1\\
0&1&1&1&1&1&0&0&1&1&0&1&1&0&1&1\\
0&0&1&1&1&1&1&0&1&1&1&0&1&1&0&1\\
0&0&0&1&1&1&1&1&0&1&0&0&0&1&0&0\\
1&0&0&0&1&1&1&0&1&1&1&1&0&1&1&1\\
1&1&0&0&0&1&1&1&0&1&1&1&1&0&1&1
\end{pmatrix}$$

La matrice L_ω (resp. L_ω^{-1}) est appliquée sur la ligne $0 \leq \omega \leq 7$ de l'état interne lors de l'application de l'étage linéaire (resp. de son inverse).

La superposition des quatre matrices n'a pas de bit égal à 0. De plus, chaque matrice binaire L_ω appliquée sur la ligne $0 \leq \omega \leq 7$ de l'état interne dans GARFIELD a les propriétés cryptographiques suivantes : un support égal à 176, c'est-à-dire 176 de ses bits sont égaux à 1, $\mathcal{B}_d(L_\omega) = 8$ et $\mathcal{B}_l(L_\omega) = 8$, ce qui est optimal. L'inverse de l'étage linéaire a les mêmes propriétés cryptographiques.

5.2.4 – Constantes de tour

Les constantes de tour sont principalement choisies pour se prémunir des attaques par glissement ("slide attacks" en anglais) [BIRYUKOV et al. 1999] et des attaques par invariant [LEANDER et al. 2011], [LEANDER et al. 2015], [TODO et al. 2016].

En effet, les attaques par invariant sont particulièrement efficaces sur les structures SPN avec une génération légère des clés de tour (ou même aucun cadencement de clés), par exemple lorsque les clés de tour ne diffèrent que par une constante de tour comme dans GARFIELD et de nombreux autres schémas de chiffrement légers. Il est alors de la plus grande importance de choisir les constantes de tour avec prudence afin de se prémunir de ces attaques.

5.2.4.1 – Spécifications des constantes de tour utilisées par GARFIELD

La première (resp. la dernière) constante de tour dans un chiffrement SPN est inutile puisqu'un attaquant peut la retirer à partir du message clair (resp. du chiffré).

Les autres constantes de tour $\mathrm{RC}_0, \cdots, \mathrm{RC}_9$ utilisées par GARFIELD ont alors les valeurs suivantes :

$$
\begin{aligned}
\mathrm{RC}_0 &= \texttt{0x307dca1764b1fe4b98e5327fcc1966b3} \\
\mathrm{RC}_1 &= \texttt{0x4d9ae73481ce1b68b5024f9ce93683d0} \\
\mathrm{RC}_2 &= \texttt{0x6ab704519eeb3885d21f6cb90653a0ed} \\
\mathrm{RC}_3 &= \texttt{0x87d4216ebb0855a2ef3c89d62370bd0a} \\
\mathrm{RC}_4 &= \texttt{0xa4f13e8bd82572bf0c59a6f3408dda27} \\
\mathrm{RC}_5 &= \texttt{0xc10e5ba8f5428fdc2976c3105daaf744} \\
\mathrm{RC}_6 &= \texttt{0xde2b78c5125facf94693e02d7ac71461} \\
\mathrm{RC}_7 &= \texttt{0xfb4895e22f7cc91663b0fd4a97e4317e} \\
\mathrm{RC}_8 &= \texttt{0x1865b2ff4c99e63380cd1a67b4014e9b} \\
\mathrm{RC}_9 &= \texttt{0x3582cf1c69b603509dea3784d11e6bb8}
\end{aligned}
$$

Les 16 octets de ces constantes de tour peuvent être successivement générés, de gauche à droite, en mettant à jour la valeur d'un seul octet RC.

Il suffit pour cela d'initialiser dans un premier temps RC avec la valeur 0xe3. Ensuite, chaque nouvel octet de la constante de tour est obtenu en incrémentant RC de 0x4d modulo 256, et une addition supplémentaire avec 0x4d est effectuée une fois que les 16 octets d'une constante ont été générés.

La même procédure peut être appliquée pour le déchiffrement : pour générer successivement les octets de $\mathrm{RC}_9, \ldots, \mathrm{RC}_0$, de gauche à droite, RC est initialisé avec la valeur 0xe8, et la valeur ajoutée après le calcul de chaque constante de tour de 32 bits est 0x13 (au lieu de 0x4d).

5.2.5 Implémentations selon les besoins en sécurité

GARFIELD a été principalement conçu pour proposer une implémentation 8 bits très efficace appliquée de façon SIMD, qui est donnée en annexe A.3.3, afin de déployer l'IRC avec le plus faible surcoût possible. La structure de GARFIELD permet également de déployer efficacement le masquage.

Il est ainsi possible d'implémenter GARFIELD en utilisant efficacement différentes contre-mesures pour se prémunir des attaques par observation et des attaques par perturbation. De ce fait, nous proposons plusieurs implémentations 32 bits de GARFIELD selon les besoins en sécurité :

- I_0-GARFIELD : l'implémentation 8 bits appliquée de façon SIMD non protégée donnée en annexe A.3.3,
- I_1-GARFIELD : l'implémentation 16 bits appliquée de façon SIMD donnée en annexe A.3.2 utilisant la seconde technique de masquage de premier ordre décrite en section 2.6,
- I_2-GARFIELD : l'implémentation 8 bits appliquée de façon SIMD protégée par l'IRC en mode détection de fautes comme décrit en section 4.1,
- I_3-GARFIELD : l'implémentation I_2-GARFIELD utilisant la seconde technique de masquage de premier ordre décrite en section 4.3.

Les attaques par observation sont souvent plus faciles à mettre en œuvre que les attaques par perturbation, et c'est pourquoi le niveau de sécurité I_1 peut parfois être suffisant en fonction du cas d'utilisation.

Néanmoins, dans un cas où les clés utilisées sont éphémères, c'est-à-dire changées très régulièrement, une attaque par perturbation peut être plus pertinente qu'une attaque par observation, et c'est alors le niveau de sécurité I_2 qui peut être suffisant.

De plus, il semble intéressant de connaître les performances d'une implémentation au niveau de sécurité I_2 puisque l'on peut imaginer combiner l'IRC avec d'autres contre-mesures que le masquage afin de contrecarrer les attaques par observation.

Le niveau de sécurité I_3 est alors souvent le plus adapté puisqu'il permet de se prémunir à la fois des attaques par observation et des attaques par perturbation.

D'autres niveaux de sécurité sont également possibles, par exemple en ne protégeant seulement qu'une partie de l'implémentation, en utilisant un masquage d'ordre supérieur, en déployant l'IRC en mode correction de fautes ou en ciblant des architectures de 64 bits ou plus, comme nous l'avons décrit en section 4.5.

Enfin, dans le cas d'une implémentation de GARFIELD protégée par l'IRC, le message clair de référence est 0xf238632e0097b3eda895a5ffe5004bff, la clé de référence est 0x3118b88b556aee109f3c5d20d9b4a750 et le chiffré de référence est 0x03bced0fc78 39c57de1eb3d64601905b.

Ce bloc de référence permet de protéger complètement le chiffrement et le déchiffrement par rapport aux prérequis pour l'IRC que nous avons donnés en section 4.1.1.

5.3 Performances

Le processeur 32 bits que nous avons utilisé pour tous les tests est un micro-contrôleur ARM Cortex-M4 puisqu'il est assez représentatif des dispositifs utilisés pour l'IoT, et nous avons utilisé le niveau d'optimisation -O3 pour la compilation. De plus, nous n'avons pas inclus le coût de la génération des masques dans les tests. Enfin, aucun tableau n'est utilisé dans les implémentations de GARFIELD.

Tout d'abord, le tableau 5.4 donne les performances que nous avons obtenues pour la couche de substitution utilisée dans GARFIELD pour les implémentations de niveau de sécurité I_0 à I_3 que nous venons de décrire en section 5.2.5. La taille est l'espace requis pour stocker le code.

Tableau 5.4.: Performances de la couche de substitution utilisée dans GARFIELD. La taille est exprimée en octets et le coût est le nombre moyen de cycles d'horloge nécessaires pour chiffrer un octet.

Implémentation	Taille	Coût
I_0-GARFIELD	172	5,6
I_1-GARFIELD	248	25,2
I_2-GARFIELD	172	22,4
I_3-GARFIELD	322	68,6

Il est à noter que le masquage a un sérieux impact sur les performances, plus que l'IRC. De plus, la couche de substitution inverse a les mêmes performances puisque la S-box utilisée est une involution.

Le tableau 5.5 donne ensuite les performances que nous avons obtenues pour l'étage linéaire utilisé dans GARFIELD et son inverse pour les implémentations de niveau de sécurité I_0 à I_3. La taille est là aussi l'espace requis pour stocker le code.

Tableau 5.5.: Performances de l'étage linéaire utilisé dans GARFIELD et de son inverse. La taille est exprimée en octets et le coût est le nombre moyen de cycles d'horloge nécessaires pour chiffrer un octet.

Implémentation	Chiffrement		Déchiffrement		Chiff.+Déchiff.	
	Taille	Coût	Taille	Coût	Taille	Coût
I_0-GARFIELD	180	7,6	202	7,8	260	18,9
I_1-GARFIELD	190	31,3	190	31,1	236	71,6
I_2-GARFIELD	180	30,4	202	31,1	260	75,7
I_3-GARFIELD	180	30,4	202	31,1	260	75,7

Nous avons alors comparé les performances de certains schémas de chiffrement à partir d'implémentations avec différents niveaux de sécurité.

Pour chaque schéma de chiffrement $\mathcal{E}$, nous avons adapté son code de référence afin d'obtenir plusieurs implémentations 32 bits optimisées : $\mathcal{E}$_8b, $\mathcal{E}$_16b et $\mathcal{E}$_32b qui appliquent $\mathcal{E}$ respectivement sur quatre blocs, deux blocs et un bloc de données avec un seul flot d'instructions 32 bits.

Nous avons alors été en mesure de déployer les niveaux de sécurité I_1, I_2 et I_3, tels que décrits dans la section 5.2.5, afin d'obtenir des implémentations I_1-$\mathcal{E}$, I_2-$\mathcal{E}$ et I_3-$\mathcal{E}$, et d'effectuer des comparaisons avec les implémentations de GARFIELD.

L'implémentation 32 bits de référence de GARFIELD est donnée en annexe A.3.1, et est comparée à son implémentation de niveau de sécurité I_0.

De plus, nous avons utilisé une implémentation de l'AES protégée par l'IIR décrite dans [Patrick et al. 2016], une implémentation de l'AES protégée par le masquage décrite dans [Schwabe et al. 2016], et une implémentation de l'AES protégée par les deux techniques, respectivement notées IIR-AES, M-AES et M-IIR-AES.

L'implémentation M-IIR-AES utilise le parallélisme par tranches de bits et prend en entrée 11 blocs de masques concaténés avec 10 blocs de données, 1 bloc de référence et 10 blocs de redondance.

Chaque opération & et | doit alors être modifiée afin de calculer correctement la valeur des masques avec la technique présentée en section 2.4.2.

La figure 5.6 illustre la composition d'un mot de 32 bits en entrée d'une implémentation protégée par l'IIR et le masquage où M_i avec $0 \leq i \leq 10$ sont les 11 blocs de masques, X_i avec $0 \leq i \leq 9$ sont les 10 blocs de données masqués et RX est le bloc de référence masqué.

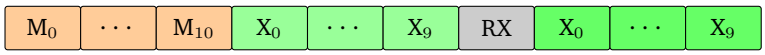


Figure 5.6.: Composition d'un mot de 32 bits en entrée d'une implémentation protégée par l'IIR et le masquage.

Le cadencement des clés utilisé par l'AES, qui nécessite 454 octets pour être stocké et 2237 cycles pour être effectué, n'a pas été inclus dans les tests puisqu'il peut être exécuté une seule fois.

Le tableau 5.6 donne finalement les comparaisons des performances que nous avons obtenues pour le chiffrement et le déchiffrement de l'AES, Fantomas, PRIDE, Robin, SPECK et GARFIELD avec les implémentations que nous avons proposées pour les différents niveaux de sécurité.

La taille comprend cette fois l'espace requis pour stocker le message clair, la clé secrète, le code du schéma de chiffrement et les tableaux utilisés. Dans le cas des implémentations protégées par l'IRC, elle inclut également l'espace requis pour stocker le code des vérifications et le bloc de référence (message clair, clé secrète et chiffré) qui nécessite 1,5 fois plus de mémoire pour GARFIELD que pour PRIDE.

Le code en langage C de chacune des implémentations que nous avons utilisées est donné en annexe A, et nous donnons pour chacune dans le tableau 5.6 entre parenthèses la section correspondante dans l'annexe.

Il est à noter que l'implémentation I_3-PRIDE ne permet pas de contrecarrer complètement l'injection de fautes, car elle n'utilise qu'un seul bloc de référence, et qu'il est nécessaire d'utiliser deux blocs de référence pour satisfaire les prérequis de l'IRC sur toute l'exécution de ce schéma de chiffrement comme nous l'avons décrit en section 4.4.1.

Le déploiement de l'IRC sur une structure ARX entraîne un surcoût élevé puisque la plupart des structures ARX utilisent l'addition sur 16, 32 ou 64 bits, qui est très coûteuse à effectuer de façon SIMD sur les octets d'un mot de 32 bits, comme nous pouvons le constater pour SPECK qui effectue des additions sur 64 bits.

Tableau 5.6.: Comparaisons des performances de certaines implémentations pour différents niveaux de sécurité. La taille est exprimée en octets et le coût est le nombre moyen de cycles d'horloge nécessaires pour chiffrer un octet.

Implémentation	Chiffrement		Déchiffrement		Chiff.+Déchiff.	
	Taille	Coût	Taille	Coût	Taille	Coût
AES_32b (A.1.1)	656	243,8	808	423,7	1 248	740,1
Fantomas_32b (A.2.1)	1 380	169,5	1 404	168,9	2 688	349,6
PRIDE_32b (A.4.1)	440	350,3	484	374,3	716	862,1
Robin_32b (A.5.1)	1 308	270,2	1 304	270,7	1 400	547,9
SPECK_32b (A.6.1)	172	323,9	188	339,4	342	710,6
GARFIELD_32b (A.3.1)	580	344,8	564	345,6	776	749,7
I_0-GARFIELD (A.3.3)	624	182,9	642	183,5	760	402,7
M-AES (A.1.2)	7 908	884,9	8 144	1 073,5	9 592	2 014,1
I_1-Fantomas (A.2.2)	1 584	475,3	1 616	478,8	2 980	966,3
I_1-PRIDE (A.4.2)	640	708,3	652	748,6	936	1 538,1
I_1-Robin (A.5.2)	1 496	669,5	1 500	669,6	1 588	1 346,6
I_1-GARFIELD (A.3.2)	664	675,6	664	672,7	792	1 559,3
IIR-AES (A.1.2)	7 850	845,0	8 056	1 050,3	9 454	1 927,6
I_2-PRIDE (A.4.3)	644	731,5	652	740,0	1 006	1 617,3
I_2-SPECK (A.6.2)	404	5 426,3	524	7 412,1	926	12 788,7
I_2-GARFIELD (A.3.3)	700	756,7	718	759,1	836	1 660,6
M-IIR-AES (A.1.2)	7 972	2 197,9	8 212	2 498,4	9 664	4 780,0
I_3-PRIDE (A.4.3)	776	1 403,6	780	1 421,9	1 148	2 974,3
I_3-GARFIELD (A.3.3)	952	1 258,1	964	1 264,3	1 084	2 667,1

Finalement, comme nous l'avons déjà mentionné, on observe que l'implémentation 8 bits de GARFIELD appliquée de façon SIMD est plus efficace que son implémentation 32 bits. Cependant, elle impose de prendre 4 blocs en entrée à chaque exécution.

5.4 Résistance aux attaques mathématiques

Nous avons fait en sorte que GARFIELD soit sécurisé, c'est-à-dire qu'il n'y ait pas d'attaques connues à ce jour avec une complexité significativement meilleure que les attaques génériques contre les chiffrements par blocs avec les mêmes paramètres, sans aucune connaissance sur la clé secrète. Nous ne considérons donc pas la sécurité pour les modèles d'attaques à clés liées, connues ou choisies par l'attaquant, dans notre contexte de l'IoT.

Nous analysons alors la résistance offerte par GARFIELD contre les attaques classiques, en nous concentrant particulièrement sur les attaques par invariant [Todo et al. 2016], [Leander et al. 2015] puisque ces techniques récentes ont été exploitées pour effectuer la cryptanalyse de la plupart des structures entrelacées proposées.

Dans cette section, nous notons n (resp. k) la taille du bloc en entrée (resp. de la clé secrète) manipulé par le chiffrement par blocs $\mathcal{E}$ considéré. De plus, nous notons

$\mathcal{R}$ sa fonction de tour sans la couche d'addition de clé, et $\mathcal{E}_K^{(r)}$ désigne r tours de chiffrement avec la clé secrète K.

5.4.1 Cryptanalyse linéaire et différentielle

Si l'on suppose que les clés de tour sont indépendantes et uniformément aléatoires, alors la complexité d'une attaque linéaire est proportionnelle à l'inverse du MELP (pour "Maximum Expected Linear Potential" en anglais) sur r tours, et la complexité d'une attaque différentielle est proportionnelle à l'inverse du MEDP (pour "Maximum Expected Differential approximation Probability" en anglais) sur r tours.

Définition 3. *Le* MELP *sur r tours d'un chiffrement par blocs $\mathcal{E}$ est noté* $\mathrm{MELP}_r(\mathcal{E})$ *et est défini par :*

$$\max_{a,b\in\mathbb{F}_2^n\setminus\{0\}} \sum_{K\in\mathbb{F}_2^k} \frac{\left(2\times\#\{x\in\mathbb{F}_2^n : a\cdot x = b\cdot\mathcal{E}_K^{(r)}(x)\}-2^n\right)^2}{2^{2n+k}}$$

et le MEDP *sur r tours de $\mathcal{E}$ est noté* $\mathrm{MEDP}_r(\mathcal{E})$ *et est défini par :*

$$\max_{a,b\in\mathbb{F}_2^n\setminus\{0\}} \sum_{K\in\mathbb{F}_2^k} \frac{\#\{x\in\mathbb{F}_2^n : \mathcal{E}_K^{(r)}(x\oplus a)\oplus\mathcal{E}_K^{(r)}(x)=b\}}{2^k}\,.$$

La résistance contre la cryptanalyse linéaire (resp. différentielle) est garantie si MELP_r (resp. MEDP_r) est suffisamment petit sur $r \leq R-2$ tours, où R est le nombre de tour du chiffrement par blocs. Malheureusement, il est généralement trop difficile de calculer la valeur exacte de MELP_r (resp. MEDP_r), bien que ce soit parfois possible pour $r=2$ comme les auteurs l'ont montré dans [Daemen et al. 2006], [Keliher et al. 2007] pour l'AES.

Cependant, il est possible d'obtenir une borne supérieure de MELP_2 (resp. MEDP_2) dans le cas d'un chiffrement SPN utilisant une S-box $\mathcal{S}$ de ω bits en calculant :

$$\mathrm{MELP}_2 \leq \delta_{\mathrm{MELP}} = \left(\frac{\mathcal{L}(\mathcal{S})}{2^\omega}\right)^{2(\mathcal{B}_l-1)}$$

$$\text{et}\quad \mathrm{MEDP}_2 \leq \delta_{\mathrm{MEDP}} = \left(\frac{\Delta(\mathcal{S})}{2^\omega}\right)^{\mathcal{B}_d-1}$$

où $\mathcal{B}_l$ (resp. $\mathcal{B}_d$) est le nombre de branchements linéaires (resp différentiels) de l'étage linéaire [Hong et al. 2001], [Daemen et al. 2002, Section B.2].

Cela permet d'obtenir une borne inférieure sur la complexité des meilleures attaques linéaires et différentielles sur 2 tours. Le tableau 5.8 donne les valeurs de δ_{MELP} et δ_{MEDP} pour certains schémas de chiffrement.

Tableau 5.7.: Valeurs des bornes supérieures de MELP et MEDP sur 2 tours pour certains schémas de chiffrement.

	AES	Fantomas	PRIDE	Robin	GARFIELD
δ_{MELP}	2^{-24}	2^{-28}	2^{-6}	2^{-28}	2^{-28}
δ_{MEDP}	2^{-24}	2^{-28}	2^{-6}	2^{-28}	2^{-35}

Pour plus de tours, nous pouvons fournir une borne supérieure sur la probabilité d'un chemin différentiel, et une borne supérieure sur la corrélation au carré d'un chemin linéaire. Pour $2r$ tours, la probabilité qu'une attaque exploitant un seul chemin différentiel existe est au plus 2^{-40r} et la corrélation au carré d'un seul chemin linéaire est au plus 2^{-32r}.

Ces bornes ne garantissent pas que les attaques linéaires et différentielles sont infaisables puisqu'elles ne prennent pas en compte le fait que plusieurs chemins puissent être utilisés. Cependant, la plupart des attaques en pratique exploite l'existence d'une approximation linéaire ou différentielle avec un chemin dominant.

5.4.2 — Attaques différentielles d'ordre supérieur, attaques par interpolation et variantes

De nombreuses attaques exploitent certaines propriétés spécifiques des formes algébriques normales des composantes du schéma de chiffrement.

Les attaques différentielles d'ordre supérieur sont notamment basées sur le fait que la fonction de chiffrement (ou une version réduite à moins de tours de la fonction de chiffrement) a un faible degré algébrique pour toutes les clés secrètes.

Dans GARFIELD, la fonction de tour a un degré 6, qui correspond au degré de la S-box utilisée. Plus précisément, si on considère les 255 composantes de la S-box, c'est-à-dire les combinaisons linéaires non nulles de ses coordonnées, on observe que la S-box a respectivement 3, 60, 192 composantes de degré 4, 5, 6.

Une borne supérieure du degré sur plusieurs tours est donnée dans [BOURA et al. 2011], et dépend du degré maximal δ_k de la fonction booléenne formée par le produit des k coordonnées de la S-box.

Par définition, on a $\delta_1 = 6$ et $\delta_k \in \{6, 7\}$ pour tout $2 \leq k < 8$. De plus, on sait de [BOURA et al. 2013, corollaire 3.3] que la plus faible valeur de k pour laquelle $\delta_k = 7$ est égale à $8 - \deg(\mathcal{S}^{-1}) = 2$ puisque $\mathcal{S}$ est une involution.

Nous pouvons alors appliquer le théorème 2 dans [BOURA et al. 2011] avec $\gamma = 6$ et déduire une borne supérieure sur le degré de r tours de GARFIELD lorsque r varie, donnée dans le tableau 5.8.

Tableau 5.8.: Borne supérieure sur le degré de GARFIELD dérivée de [BOURA et al. 2011].

Nombre de tours	1	2	3	4	5
Degré algébrique	6	36	112	125	127

Nous pouvons alors observer que la borne sur le degré est maximale après 5 tours. Bien que ce résultat corresponde à une borne supérieure sur le degré d'une fonction itérative, il a été montré que cela fournit une bonne estimation du degré dans de nombreux cas pratiques.

Nous pensons que le nombre total de tours dans GARFIELD fournit alors une bonne marge de sécurité par rapport aux attaques exploitant un faible degré ou une forme algébrique normale peu dense du schéma de chiffrement, incluant les attaques différentielles d'ordre supérieur [KNUDSEN 1995], [LAI 1994], certains distingueurs de type "cube distinguishers" [AUMASSON et al. 2009] ou les attaques récentes basées sur la propriété de division [TODO 2015].

5.4.3 — Attaques par sous-espace invariant

Une attaque par sous-espace invariant [LEANDER et al. 2011], [LEANDER et al. 2015] exploite l'existence d'un sous-espace $V \subseteq \mathbb{F}_2^n$ et de deux constantes $\alpha, \beta \in \mathbb{F}_2^n$ tels que :

$$\mathcal{R}(V \oplus \alpha) = V \oplus \beta,$$

où $\mathcal{R}$ est la fonction de tour sans la couche d'addition de clé.

Alors, si toutes les clés de tour K_i appartiennent à $V \oplus \alpha \oplus \beta$, l'attaquant obtient la propriété suivante : $\forall P \in V, \mathcal{E}_K(P \oplus \beta) \in V \oplus \alpha$, qui évidemment lui permet d'obtenir un distingueur pour des clés faibles. Les clés faibles sont celles pour lesquelles toutes les clés de tour appartiennent à $V \oplus \alpha \oplus \beta$.

Ensuite, une telle attaque peut être obtenue s'il existe un sous-espace $V \subseteq \mathbb{F}_2^n$ et deux constantes $\alpha, \beta \in \mathbb{F}_2^n$ tels que :

$$\mathrm{Llayer}(\mathrm{Slayer}(V \oplus \alpha)) = V \oplus \beta.$$

et tels que V contient toutes les différences entres les constantes de tour utilisées par le schéma de chiffrement.

En effet, si toutes les clés de tour $K_i = K \oplus \mathrm{RC}_i$ appartiennent à $V \oplus \alpha \oplus \beta$, alors toutes les différences

$$K_i \oplus K_j = \mathrm{RC}_i \oplus \mathrm{RC}_j$$

sont dans V pour $i \neq j$.

Puisque la première constante de tour RC_0 dans GARFIELD est égale à zéro, nous déduisons que l'espace linéaire V doit contenir toutes les combinaisons linéaires des constantes de tour RC_i.

Alors, l'algorithme donné dans [LEANDER et al. 2015] permet de trouver un tel sous-espace invariant pour n'importe quelle fonction de tour de n bits. Il prend en entrée un vecteur aléatoire $\alpha \in \mathbb{F}_2^n$ et un sous-espace $V \subseteq \mathbb{F}_2^n$ initialisé par toutes les combinaisons linéaires des constantes de tour.

L'algorithme est le suivant :

```
1  fonction Closure(V, α){
2   β ← Llayer(Slayer(α))
3   StableCount ← 0
4   tant que StableCount<N
5        Tirer un vecteur aléatoire x ∈ span(V)⊕α
6        si Llayer(Slayer(x))⊕β ∈ span(V) alors
7             StableCount ← StableCount +1
8        sinon
9             Ajouter Llayer(Slayer(x))⊕β à V
10            StableCount ← 0
11  retourner span(V)⊕α
12 }
```

où N contrôle le risque d'erreur. La probabilité p qu'un sous-espace invariant de dimension d ne soit pas détecté par cette approche après avoir exécuté l'algorithme pour m itérations, avec différents vecteurs aléatoires α, est alors donnée dans [LEANDER et al. 2015] par :

$$p = (1 - 2^{d-n})^m$$

Dans le cas de GARFIELD, nous avons exécuté la fonction Closure avec $N = 50$ pour 2^{34} itérations, avec différents vecteurs aléatoires α comme dans [LEANDER et al. 2015], et nous n'avons obtenu aucun sous-espace invariant.

Ainsi, la probabilité qu'un sous-espace invariant de dimension supérieure à 97 n'ait pas été détecté est inférieure à 0.001.

5.4.4 — Attaques par invariant non-linéaire

Une attaque par invariant non-linéaire consiste à trouver une fonction booléenne $f : \mathbb{F}_2^n \to \mathbb{F}_2$ telle que, pour chaque tour, il existe une constante $\varepsilon_i \in \mathbb{F}_2$ qui satisfait :

$$f(x) = f(\mathcal{R}(x) \oplus K_i) \oplus \varepsilon_i, \ \forall x \in \mathbb{F}_2^n \ .$$

Nous nous concentrons sur les fonctions booléennes f qui ont la particularité d'être invariantes à la fois par la couche de substitution et par l'étage linéaire à chaque tour. En effet, toutes les attaques par invariant non-linéaire exploitent de telles fonctions booléennes en pratique [TODO et al. 2016].

L'absence de telles fonctions n'implique pas qu'il n'en existe aucun invariant par la fonction de tour, mais cela signifie qu'un attaquant doit nécessairement considérer l'ensemble de la fonction de tour dans l'attaque, et aucun des algorithmes connus aujourd'hui n'est applicable pour cette taille de bloc.

Ainsi, nous avons cherché des fonctions booléennes f telles que, pour toutes les clés de tour $K_i = K \oplus \mathrm{RC}_i$, nous avons

$$f(x) = f(\mathrm{Llayer}(x) \oplus K_i) \oplus \varepsilon_i \ , \forall x \in \mathbb{F}_2^n \ .$$

Alors, il a été prouvé dans [BEIERLE et al. 2017] que l'ensemble des structures linéaires de f contient l'espace vectoriel W formé par toutes les combinaisons linéaires des valeurs de $\mathrm{Llayer}^i(\mathrm{RC}_j)$ pour $0 \leq j \leq 9$ et $i \geq 0$. La dimension de W est alors inférieure au produit entre le nombre de constantes de tour linéairement indépendantes et le degré du polynôme minimal de l'étage linéaire.

Dans le cas de GARFIELD, l'étage linéaire a un polynôme minimal de degré 16. Nous avons alors vérifié que l'espace W couvre entièrement l'espace $\mathbb{F}_2^{128}$.

Nous avons finalement déduit de [BEIERLE et al. 2017, Prop. 2] qu'il n'existe pas de fonction booléenne invariante non-triviale à la fois sur la couche de substitution et sur l'étage linéaire pour chaque tour, et cette propriété est indépendante de la couche de substitution.

La résistance aux attaques par invariant non-linéaire résulte du choix d'un étage linéaire avec un polynôme minimal ayant un degré raisonnable, ce qui est très différent du cas de SCREAM ou de iSCREAM [TODO et al. 2016].

5.5 Mise en œuvre d'attaques physiques

Du fait que GARFIELD a une structure entrelacée, la CEMA que nous avons décrite en section 2.3 et la DFA que nous avons présentée en section 3.3 sont applicables aux implémentations non protégées de GARFIELD.

Afin de tester la résistance de GARFIELD à ces attaques, nous avons proposé dans un premier temps une mise en œuvre pratique de la DFA introduite en section 3.3 sur les deux implémentations 32 bits non protégées de GARFIELD, GARFIELD_32b et I_0-GARFIELD. Nous avons ensuite appliqué la même attaque sur l'implémentation de GARFIELD protégée par l'IRC sans masquage, I2-GARFIELD, avant d'appliquer finalement la CEMA présentée en section 2.3 sur cette même implémentation, et sur l'implémentation de GARFIELD protégée par l'IRC avec masquage, I3-GARFIELD.

Les différentes implémentations de GARFIELD que nous avons manipulées sont données en annexe A.3, et le processeur 32 bits que nous avons utilisé pour tous les tests était un micro-contrôleur ARM Cortex-M3.

Aucune autre contre-mesure que l'IRC ou le masquage n'a été implémentée sur ces implémentations ou sur le circuit intégré.

Dans un premier temps, nous avons effectué une analyse des rayonnements électromagnétiques de GARFIELD. La figure 5.7 donne les courbes que nous avons obtenues qui nous ont permis d'identifier toutes les étapes effectuées par GARFIELD.

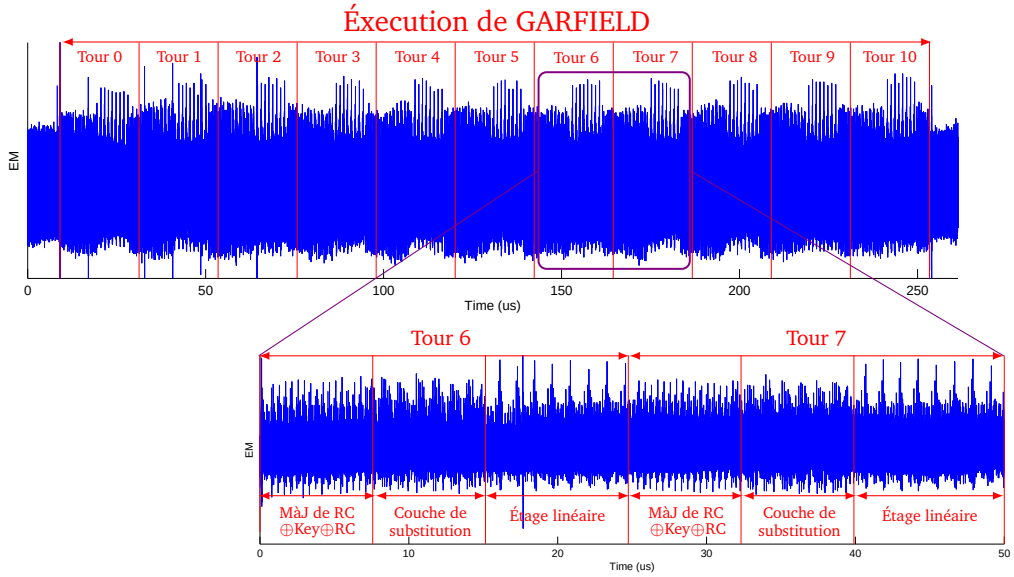


Figure 5.7.: Analyse des rayonnements électromagnétiques de GARFIELD.

5.5.1 – DFA menée en laboratoire sur GARFIELD

La DFA présentée en section 3.3 permet de retrouver entièrement la clé secrète utilisée par une implémentation non protégée de GARFIELD en seulement deux fautes théoriques sur différentes lignes de l'état interne.

Nous rappelons que l'objectif de l'attaque est de corrompre une ligne de l'état interne lors de l'exécution de l'avant-dernier étage linéaire afin d'obtenir des différentielles sur chaque S-box de la dernière couche de substitution.

Alors, chaque différentielle permet à l'attaquant de réduire les valeurs possibles pour l'octet de clé correspondant en utilisant la table de distribution des différences de la S-box de GARFIELD.

5.5.1.1 – Mise en œuvre pratique sur I_0-GARFIELD

Tout d'abord, nous avons utilisé l'analyse des rayonnements électromagnétiques de I_0-GARFIELD décrite à la figure 5.7 afin d'identifier le dernier étage linéaire que nous avons ensuite ciblé pour mener la DFA présentée en section 3.3.

Pour notre expérimentation, nous avons utilisé pour toutes les exécutions la même clé secrète K = 0x18a27b3d61fe84580a31e4b7c6ed99d1 et le même message clair P = 0xe824d36e15807fcb16a74241af039029 sur les 4 blocs (nous rappelons que I_0-GARFIELD chiffre 4 blocs d'une façon SIMD), et le chiffré correct était C = 0x7d9fcd3efeaf26519d3f91c37da85f92 sur chaque bloc.

Afin d'injecter des fautes sur le circuit intégré, nous avons utilisé le banc d'injection électromagnétique décrit à la figure 3.1 puisque cela permet de ne pas avoir à décapsuler le circuit intégré, et de cibler assez précisément des instructions durant l'exécution de la fonction de chiffrement de I_0-GARFIELD.

Les impulsions de tension avaient une durée de 200ns, nous avons fait varier la tension par pas de 1V entre 180V et 219V, et nous avons injecté 50 impulsions en ciblant le milieu du circuit intégré. Cette approche nous a permis d'obtenir des fautes satisfaisant les conditions de la DFA, c'est-à-dire n'affectant qu'une ligne de l'état interne, puisque I_0-GARFIELD manipule des octets.

Plus précisément, à partir des 2 000 impulsions électromagnétiques, nous n'avons pas obtenu de chiffré dans 598 des cas, et nous avons obtenu 84 chiffrés corrompus dont 16 exploitables. Les chiffrés corrompus non exploitables provenaient d'un dysfonctionnement de l'UART dû aux fautes.

Il est à noter que les chiffrés corrompus n'avaient pas la même valeur sur les 4 blocs, et auraient donc tous été détectés par l'IRC.

Le tableau 5.9 fournit les chiffrés corrompus exploitables que nous avons obtenus ainsi que notre connaissance sur les différences autour de la dernière couche de substitution pour chacun, c'est-à-dire la valeur de la différence en sortie Δ_{out} (calculée à partir du chiffré correct et du chiffré corrompu), et la valeur δ_{in} de chaque octet de la différence en entrée, où θ, β et γ désignent respectivement les ensembles {0x01,0x10}, {0x08,0x20} et {0x08,0x80} des valeurs possibles pour δ_{in}.

Tableau 5.9.: Chiffrés corrompus obtenus à partir de fautes lors de l'exécution de I_0-GARFIELD en notation hexadécimale.

Chiffrés corrompus	Δ_{out}	δ_{in}
47f7e05d1fbb833d9d3f2a0b1b8cc646	b5a6004700e636000000000000000000	80
95c18cd91fbb6e039d3f91d228f8f501	b5a6c50096e636000000000000000000	80
1b24b6b1feafea269d84b3b939e77d35	00009e000000d9000000000000000000	θ
07bd74e3579c2651e32b567e03bc617f	000000000000000004a00006b0000e4	40
5cd61f08b6fd0b329de2d5d95fd20ab5	00e5003c009000ef0000000000000000	08
07bd74e355415c7352bd6974b22a57ad	00000000000000000006134004b0000eb	01
300653b618d8f5409d3fa6f63d51564a	00000000000000000872500e6000032	80
7d9ff02fc705ef04f19a60acf2d32349	000000000000000003e007900000000	10
3c783b21bf48f8b29d3f5d65f541d77b	b500000000e600000000000000000000	γ
a596e05d922bb0e09d3f6e8882c14eaf	0000004796e636000000000000000000	80
0610cd3eead12651047291c36c7bd70c	00a800002b000000000000000000000	β
ee35cd3e501429ae832eb13f7c4f8346	000000000000000039001f0000be00bf	10
7d9f397a60272651023aed186dd62186	0000000000000000063000e00000b00	20
4bddcd3e922b70bd5122d5625f0f5f92	00000016003cb8840000000000000000	02
61bdb90d598db273a3d269747da89338	0000000000000000000095b10000c159	04
7d9fcd3e852092dc15a180fed7911b66	003b00160000000000000000000000000	02

Le tableau 5.10 donne les différentielles sur $\mathcal{S}^{-1}$ qui nous ont permis de retrouver autant d'information qu'avec toutes les fautes, et donne les candidats en notation hexadécimale obtenus par intersection, noté $\cap$, pour chaque octet B_i de

$$\mathcal{L}\text{l}^{-1}(\text{K}) = 0\text{x}4408\text{e}8\text{bba}473\text{fb}986\text{ae}4\text{a}6342360\text{b}3\text{b}0$$

avec $0 \leq i \leq 15$, où $\mathcal{L}$l est l'étage linéaire de GARFIELD.

Par exemple, le premier chiffré corrompu dans le tableau 5.9 permet d'obtenir la différentielle (0xa6,0x80) sur la S-box inverse pour B_1, et de réduire ses valeurs possibles à l'ensemble {0x08,0xae,0x2e,0x88,0x58, 0xfe,0x7e,0xd8}.

Ensuite, le dernier chiffré corrompu permet d'obtenir la différentielle (0x3b,0x02) pour B_1, et de réduire ses valeurs possibles à l'ensemble {0x08,0x33}, et donc de retrouver sa valeur correcte par intersection des ensembles obtenus.

Tableau 5.10.: Connaissance à partir du tableau 5.9 sur chaque octet B_i pour $0 \leq i \leq 15$ en notation hexadécimale.

	Différentielles obtenues sur $\mathcal{S}^{-1}$	$\cap$
B_0	(b5,80)	{04,31,44,71,84,b1,c4,f1}
B_1	(a6,80) et (3b,02)	08
B_2	(c5,80) et (9e,θ)	e8
B_3	(47,80) et (3c,08)	bb
B_4	(96,80) et (2b,β)	a4
B_5	(e6,80) et (90,08)	73
B_6	(36,80) et (d9,θ)	fb
B_7	(ef,08) et (84,02)	98
B_8	(39,10)	{0a,1a,23,33,43,53,6a,7a}
B_9	(4a,40) et (61,01)	e4
B_{10}	(34,01) et (25,80)	a6
B_{11}	(79,10) et (0e,20)	34
B_{12}	(6b,40) et (4b,01)	23
B_{13}	(be,10)	{40,50,60,70,ce,de,ee,fe}
B_{14}	(0b,20) et (c1,04)	b3
B_{15}	(e4,40) et (eb,01)	b0

Finalement, nous avons obtenu 512 candidats pour $\mathcal{L}l^{-1}(K)$ et nous avons retrouvé la clé secrète en testant toutes les possibilités.

5.5.1.2 – Mise en œuvre pratique sur GARFIELD_32b

Une fois de plus, nous avons utilisé l'analyse des rayonnements électromagnétiques de GARFIELD_32b pour identifier toutes les étapes lors de son exécution.

Cependant, GARFIELD_32b manipule des mots de 32 bits composés des octets de 4 lignes différentes de l'état interne. De ce fait, appliquer la DFA présentée en section 3.3 est plus difficile puisque chaque différence sur un octet avant la dernière couche de substitution pourra prendre 16 valeurs différentes (du fait que la faute peut affecter 4 lignes de l'état interne).

Afin d'appliquer une DFA sur GARFIELD_32b, nous avons alors exploité un autre chemin d'attaque présenté à la section 3.2.2 qui consiste à effectuer un saut d'instruction sur la dernière couche d'addition de clé afin de retrouver la valeur de la clé à partir de la différence entre le chiffré correct et le chiffré corrompu.

Nous avons également utilisé le banc d'injection électromagnétique décrit à la figure 3.1 et nous avons directement obtenu complètement la clé secrète deux fois à partir de seulement 20 impulsions. La faute obtenue était un saut d'instruction, et aurait donc été détectée par l'IRC.

5.5.1.3 – Mise en œuvre pratique sur I_2-GARFIELD

Là encore, nous avons utilisé l'analyse des rayonnements électromagnétiques de I_2-GARFIELD pour identifier toutes les étapes lors de son exécution. Nous avons alors utilisé une fois de plus le banc d'injection électromagnétique et nous avons ciblé le dernier tour afin de couvrir les deux chemins d'attaque précédents.

Les impulsions de tension avaient une durée de 200ns, nous avons fait varier la tension par pas de 1V entre 180V et 219V, et nous avons injecté 250 impulsions en ciblant le milieu du circuit intégré.

À partir des 10 000 impulsions électromagnétiques, nous n'avons pas obtenu de chiffré dans 1 852 des cas, et nous avons obtenu 427 chiffrés corrompus dont 33 exploitables : 18 à partir d'une faute sur le dernier étage linéaire et 15 à partir d'un saut d'instruction sur la dernière couche d'addition de clé. Les chiffrés corrompus non exploitables provenaient d'un dysfonctionnement de l'UART dû aux fautes.

L'IRC nous a permis de complètement contrecarrer un tel moyen d'injection de fautes : toutes les 427 fautes ont été détectées, et l'utilisation de la redondance spatiale et du bloc de référence a été nécessaire pour toutes les détecter.

5.5.2 – CEMA menée en laboratoire sur GARFIELD

La CEMA présentée en section 2.3 est applicable aux implémentations non protégées de GARFIELD. Nous rappelons que l'objectif est de calculer les prédictions sur chaque octet avant la première couche de substitution (ou après la dernière) en fonction de chaque hypothèse d'octet de clé possible, et d'appliquer le coefficient de corrélation de Pearson entre les prédictions et les données d'acquisition des rayonnements électromagnétiques de la couche de substitution correspondante.

5.5.2.1 – Mise en œuvre pratique sur I_2-GARFIELD

Tout d'abord, nous avons utilisé l'analyse des rayonnements électromagnétiques de I_2-GARFIELD afin d'identifier la première couche de substitution que nous avons ensuite ciblée pour mener l'attaque. Nous avons exécuté I_2-GARFIELD pour 1 000 messages clairs aléatoires avec la clé secrète 0x18a27b3d61fe84580a31e4b7c6ed99d1 en utilisant le bloc de référence donné en section 5.2.5.

Nous avons alors ciblé la première couche de substitution pour effectuer des acquisitions avec le banc d'écoute du rayonnement électromagnétique décrit à la figure 2.2, et nous avons obtenu des traces avec 5 000 points par exécution.

Notons T la matrice des traces obtenues :

$$\mathrm{T} = \begin{bmatrix} \mathrm{T}_0 \\ \vdots \\ \mathrm{T}_{4999} \end{bmatrix} = \begin{bmatrix} \mathrm{t}_{0,0} & \cdots & \mathrm{t}_{0,999} \\ \vdots & \ddots & \vdots \\ \mathrm{t}_{4999,0} & \cdots & \mathrm{t}_{4999,999} \end{bmatrix}.$$

Nous avons ensuite calculé les matrices d'estimation E^i à partir du poids de Hamming wt du XOR entre l'octet i de chaque message clair, noté $(\mathrm{M}_j)_i$ pour $0 \leq j \leq 999$, et chaque hypothèse de clé $0 \leq \mathrm{H_K} \leq 255$:

$$\mathrm{E}^i = \begin{bmatrix} \mathrm{E}^i_0 \\ \vdots \\ \mathrm{E}^i_{255} \end{bmatrix} = \begin{bmatrix} \mathrm{e}^i_{0,0} & \cdots & \mathrm{e}^i_{0,999} \\ \vdots & \ddots & \vdots \\ \mathrm{e}^i_{255,0} & \cdots & \mathrm{e}^i_{255,999} \end{bmatrix}$$

avec $\mathrm{e}^i_{\mathrm{H_K},j} = wt((\mathrm{M}_j)_i \oplus \mathrm{H_K})$. Nous avons enfin calculé les matrices des coefficients de corrélation P^i, pour $0 \leq i \leq 7$, à partir du coefficient de corrélation de Pearson noté ρ entre E^i et T :

$$\mathrm{P}^i = \begin{bmatrix} \mathrm{P}^i_0 \\ \vdots \\ \mathrm{P}^i_{4999} \end{bmatrix} = \begin{bmatrix} \rho^i_{0,0} & \cdots & \rho^i_{0,255} \\ \vdots & \ddots & \vdots \\ \rho^i_{4999,0} & \cdots & \rho^i_{4999,255} \end{bmatrix}$$

avec $\rho^i_{t,\mathrm{H_K}} = \rho(\mathrm{T}_t, \mathrm{E}^i_{\mathrm{H_K}})$. La figure 5.8 donne les courbes obtenues en traçant P^0 sur l'intervalle 830 à 930 points en abscisse.

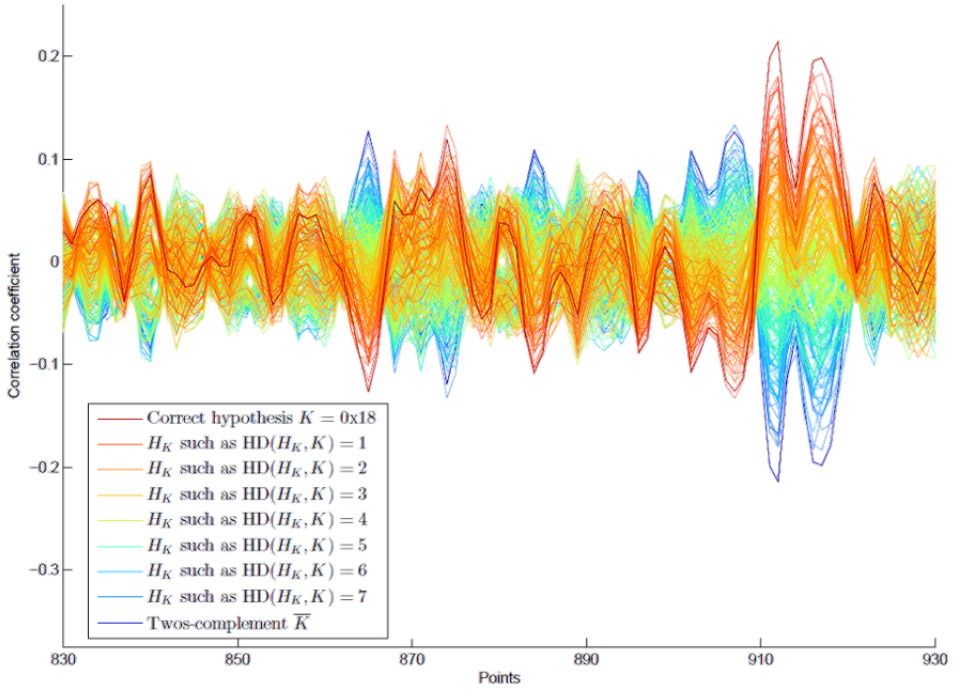


Figure 5.8.: Courbes de corrélation obtenues pour K_0 sur I_2-GARFIELD.

Nous avons pu ainsi clairement distinguer la valeur de corrélation la plus élevée pour K_0 qui était 0x18. En répétant les étapes précédentes sur les octets suivants de K_0, nous avons retrouvé sa valeur correcte à partir de tous les P^i.

5.5.2.2 – Mise en œuvre pratique sur I_3-GARFIELD

Nous avons exécuté I_3-GARFIELD pour 100 000 messages clairs aléatoires avec les mêmes paramètres que précédemment, mais en utilisant cette fois un masque aléatoire différent à chaque exécution. Nous avons alors appliqué l'attaque comme auparavant en utilisant le coefficient de corrélation de Pearson à partir de 25 000, 50 000, 75 000 et 100 000 messages clairs aléatoires. La figure 5.9 donne les courbes obtenues en traçant P^0 sur la couche de substitution complète.

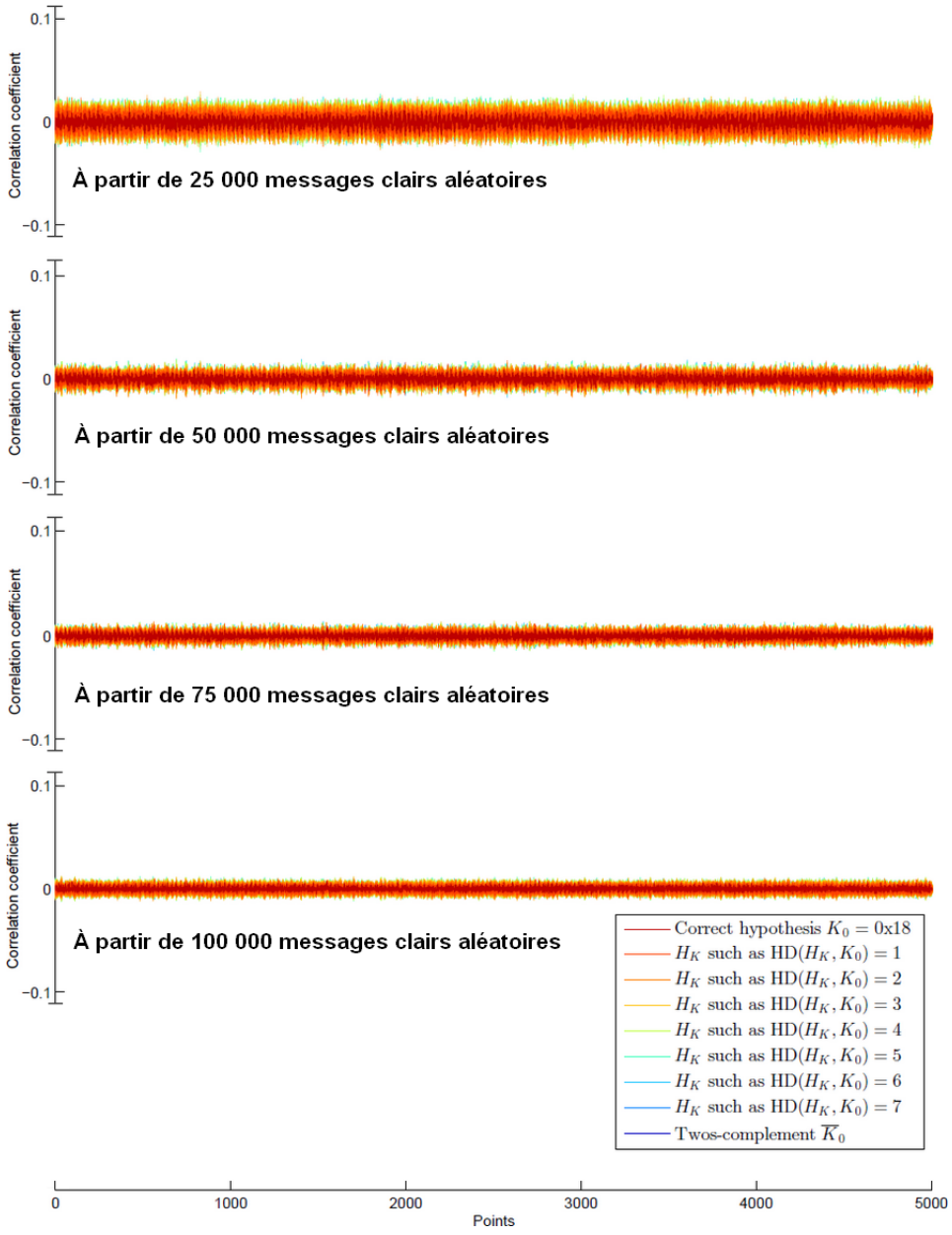


Figure 5.9.: Courbes de corrélation obtenues pour K_0 sur I_3-GARFIELD.

Bien que nous ayons utilisé 100 fois plus d'acquisitions, nous n'avons détecté aucune corrélation. Il est à noter que la corrélation a tendance à diminuer lorsque l'on augmente le nombre de courbes, ce qui est un des avantages de la CEMA.

En effet, comme nous l'avons mentionné en section 2.2.3.3, la CEMA permet d'éliminer certains mauvais pics de corrélation en utilisant plus de données d'acquisition. De ce fait, une légère corrélation, qui peut ne pas être observée avec peu de courbes, est détectée avec suffisamment de courbes.

Finalement, les mises en œuvre pratiques que nous avons effectuées ont montré que notre schéma masqué de l'IRC est nécessaire pour se prémunir d'une CEMA.

Conclusion du chapitre

GARFIELD a une structure entrelacée dont l'implémentation utilise un maximum d'instructions qui sont facilement applicables de façon SIMD, afin d'être en mesure de déployer efficacement l'IRC et le masquage. Nous avons tenté de minimiser le nombre d'opérations nécessaires pour implémenter chacune de ses composantes, tout en ayant de bonnes propriétés cryptographiques afin que GARFIELD soit sécurisé vis-à-vis des cryptanalyses classiques.

Nous avons alors proposé plusieurs implémentations avec différents niveaux de sécurité pour les applications de l'IoT, et nous pensons que GARFIELD permet ainsi d'obtenir un bon compromis entre sécurité et performances.

Nous avons montré que chaque implémentation obtenue a des performances très proches de celles d'une implémentation de PRIDE avec le même niveau de sécurité, qui est à ce jour l'un des chiffrements par blocs les plus performants en terme d'implémentation logicielle mais qui a le défaut d'opérer sur une taille de blocs plus petite, et a des performances meilleures que celles d'une implémentation de l'AES avec un niveau de sécurité similaire.

Nous avons ensuite montré que GARFIELD permet de se prémunir des attaques classiques, en particulier des attaques par invariant qui ont permis la cryptanalyse de la plupart des structures entrelacées.

Finalement, nous avons détaillé des mises en œuvre en laboratoire de la CEMA introduite à la section 2.3, et de la DFA proposée à la section 3.3.1, sur les différentes implémentations de GARFIELD. L'implémentation de GARFIELD sécurisée par l'IRC et le masquage nous a alors permis de contrecarrer ces attaques.

Chapitre 6 Conclusion et perspectives

L'objectif de notre étude était d'analyser les besoins en terme de sécurité de l'IoT, plus précisément la résistance des schémas de chiffrement légers aux attaques physiques, et la conception et l'étude de solutions afin de s'en prémunir.

6.1 Conclusion

Afin de répondre à notre problématique, nous avons dans un premier temps porté notre étude sur le schéma de chiffrement PRIDE, qui est à ce jour l'un des chiffrements par blocs les plus performants en terme d'implémentation logicielle.

Nous avons introduit une analyse de corrélation électromagnétique, et une analyse différentielle de fautes exploitant la structure de PRIDE, dont la couche de substitution peut être implémentée efficacement avec la technique du parallélisme par tranches de bits. Il s'agit de la première approche d'attaques physiques à notre connaissance contre ce type de structures, avec deux nouveaux chemins d'attaques que nous avons validés expérimentalement en laboratoire sur des cibles réelles.

Nous avons ensuite généralisé ces attaques aux autres chiffrements SPN ayant une structure similaire, et nous avons proposé une contre-mesure logicielle générique afin de se prémunir des injections de fautes que nous avons nommée l'IRC.

L'IRC permet de se prémunir de la plupart des injections de fautes, et peut être combinée efficacement avec le masquage de premier ordre et d'ordre supérieur afin de contrecarrer la plupart des attaques par observation.

Nous avons déployé l'IRC sur PRIDE, qui utilise l'addition sur un octet, et nous avons constaté que l'instruction SIMD UADD8 disponible sur un micro-contrôleur ARM Cortex-M4, qui applique l'addition en parallèle sur les octets d'un mot de 32 bits, permet de considérablement réduire le surcoût engendré par l'IRC.

Avoir d'autres instructions SIMD à disposition, notamment la rotation, permettrait de réduire d'avantage le coût d'une implémentation protégée par l'IRC.

PRIDE est également l'un des schémas de chiffrement les mieux adaptés à l'IRC, puisque, contrairement à la plupart des autres schémas de chiffrement, son implémentation de référence n'utilise pas de tableau. Cependant, PRIDE manipule des blocs de 64 bits, et n'offre pas une marge de sécurité suffisante aux attaques classiques [LALLEMAND et al. 2017].

Afin de proposer un schéma de chiffrement adapté à l'IRC, offrant une marge de sécurité suffisante vis-à-vis de la cryptanalyse classique, et ayant également une structure facilitant le masquage, nous avons alors orienté nos travaux sur la conception d'une structure entrelacée, que nous avons nommée GARFIELD, et dont l'implémentation de référence met en œuvre un certain nombre d'instructions qui sont facilement applicables de façon SIMD.

Nous avons vérifié la résistance de GARFIELD aux attaques mathématiques classiques, notamment aux récentes attaques par invariants [LEANDER et al. 2011], [LEANDER et al. 2015], [TODO et al. 2016] qui ont permis la cryptanalyse de la plupart des structures entrelacées.

Nous avons obtenu des performances similaires à celles de PRIDE pour différents niveaux de sécurité, avec ou sans l'IRC et le masquage. Finalement, nous avons validé expérimentalement la résistance d'une implémentation sécurisée de GARFIELD face aux attaques physiques que nous avons proposées sur ce type de structure.

Ces travaux de recherche ont fait l'objet de plusieurs publications scientifiques internationales dont la liste est rappelée à la section 1.7.

6.2 Perspectives

Afin de compléter et d'améliorer les résultats que nous avons obtenus, des études sont possibles, et même souvent nécessaires, pour optimiser le déploiement de l'IRC en fonction du cas d'utilisation, en particulier s'il est combiné avec le masquage.

En effet, les applications de l'IoT utilisent rarement un schéma de chiffrement seul, mais généralement avec un protocole dédié, et bien que nous nous soyons concentrés uniquement sur la sécurisation du schéma de chiffrement, il est important d'étudier au cas par cas comment déployer au mieux l'IRC au sein de l'application ciblée.

Par exemple, la façon de stocker les données dans chaque registre peut être différente d'une application à une autre, et il est nécessaire d'adapter le protocole dédié afin d'optimiser le placement des données et des blocs de référence dans les registres.

Ensuite, nous n'avons pas décrit de méthode particulière pour appliquer les tests de vérifications à la fin d'une implémentation protégée par l'IRC.

Bien que le nombre de tests soit assez conséquent, et qu'il est donc difficile pour un attaquant de tous les contourner, on peut imaginer une attaque par fautes exploitant plusieurs injections à différentes positions temporelles : la première sur une instruction précise n'affectant qu'un seul registre à la fin de l'exécution, et les autres pour contourner les vérifications correspondantes.

Par exemple, cibler la dernière couche d'addition de clé dans GARFIELD peut n'affecter qu'un seul registre à la fin de l'exécution, et l'attaquant peut obtenir un chiffré corrompu s'il arrive à contourner toutes les vérifications sur le registre corrompu. Par contre, cibler l'avant-dernier étage linéaire dans GARFIELD affecte nécessairement plusieurs registres à la fin de l'exécution, car le chiffrement applique ensuite sa couche de substitution et son étage linéaire, et il devient très difficile pour un attaquant de contourner toutes les vérifications correspondantes.

De ce fait, il peut être nécessaire d'utiliser un protocole particulier pour effectuer les vérifications, voire même une contre-mesure supplémentaire dédiée à les sécuriser.

Dans le cas où l'IRC est combiné avec le masquage, il est absolument nécessaire d'aborder la question du choix du générateur de masques aléatoires employé. Il s'agit d'une étape fondamentale du masquage sur laquelle portent de nombreuses études [Howgrave-Graham et al. 2001], [Tsoi et al. 2003], [Li et al. 2006], et il est possible d'obtenir un générateur de masques aléatoires à partir d'une implémentation logicielle [Viega 2003] ou matérielle [Konuma et al. 2005].

Afin d'éviter certaines attaques qui exploitent les propriétés statistiques d'un générateur de masques aléatoires, il est également nécessaire que ce dernier vérifie un certains nombre de tests, comme les tests du NIST [Bassham et al. 2010].

Nous n'avons étudié le déploiement de l'IRC que sur des implémentations logicielles, et il serait également intéressant d'étudier la possibilité de le déployer sur des implémentations matérielles.

De plus, il serait aussi d'un grand intérêt d'étudier les différentes possibilités pour combiner l'IRC avec d'autres contre-mesures que le masquage en fonction de l'architecture et de l'application ciblées.

Enfin, les structures ARX proposent des implémentations logicielles très efficaces, parmi lesquelles on peut notamment citer SPECK [Beaulieu et al. 2015], comme nous l'avons constaté à la section 5.3, ou SPARX [Dinu et al. 2016].

Cependant, la plupart utilisent l'addition sur 16, 32 ou 64 bits, qui est très coûteuse à effectuer de façon SIMD sur les octets d'un mot de 32 bits.

Bien que l'analyse de la résistance de ces structures aux attaques mathématiques classiques soit souvent difficile, il serait intéressant d'étudier la possibilité de proposer une structure ARX utilisant l'addition sur 8 bits afin de minimiser le coût du déploiement de l'IRC.

Implémentations utilisées

Annexe A

Cette annexe fournit le code en langage C des différentes implémentations 32 bits que nous avons utilisées pour tester les performances et la résistance aux attaques physiques de certains schémas de chiffrement sur des architectures 32 bits. Chaque implémentation permet d'appliquer soit la fonction de chiffrement, soit la fonction de déchiffrement, et il est alors possible d'en déduire facilement une implémentation uniquement d'une seule des deux fonctions. Nous décrivons également comment déployer des contremesures dédiées aux architectures 32 bits pour se prémunir des attaques physiques lorsque cela est possible.

Sommaire

A.1 Implémentations de l'AES **146**
A.1.1 Implémentation 32 bits 146
A.1.2 Parallélisme par tranches de bits 148
A.2 Implémentations de Fantomas **153**
A.2.1 Implémentation 32 bits 153
A.2.2 Implémentation 16 bits appliquée sur 2 blocs 156
A.3 Implémentations de GARFIELD **158**
A.3.1 Implémentation 32 bits 158
A.3.2 Implémentation 16 bits appliquée sur 2 blocs 160
A.3.3 Implémentation 8 bits appliquée sur 4 blocs 162
A.4 Implémentations de PRIDE **165**
A.4.1 Implémentation 32 bits 165
A.4.2 Implémentation 16 bits appliquée sur 2 blocs 168
A.4.3 Implémentation 8 bits appliquée sur 4 blocs 171
A.5 Implémentations de Robin **174**
A.5.1 Implémentation 32 bits 174
A.5.2 Implémentation 16 bits appliquée sur 2 blocs 176
A.6 Implémentations de SPECK **178**
A.6.1 Implémentation 32 bits 178
A.6.2 Implémentation 8 bits appliquée sur 4 blocs 179
A.7 Implémentations de TRIVIUM **182**
A.7.1 Implémentation 32 bits 182
A.7.2 Implémentation 8 bits appliquée sur 4 blocs 183

A.1 – Implémentations de l'AES

Les spécifications de l'AES sont données dans [DAEMEN et al. 2002]. Le message clair, le chiffré ainsi que les clés sont stockés de gauche à droite dans les registres.

A.1.1 – Implémentation 32 bits

L'implémentation 32 bits de l'AES que nous avons utilisée, décrite dans [BERTONI et al. 2003], prend en entrée un message clair ou un chiffré stocké dans les registres "state", les 11 clés de tours stockées dans les registres "Rkey" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon.

Son code est le suivant :

```
void AES_32b(uint32_t state[4], uint32_t Rkey[11][4], uint8_t dec){
    uint8_t i,round;
    if(dec){
        for(i=0;i<4;i++){
            state[i] ^= Rkey[10][i];
        }
        ShiftRows(state,dec);
        SubBytes(state,dec);
        for(i=0;i<4;i++){
            state[i] ^= Rkey[9][i];
        }
        for(round=0;round<9;round++){
            MixColumns(state,dec);
            ShiftRows(state,dec);
            SubBytes(state,dec);
            for(i=0;i<4;i++){
                state[i] ^= Rkey[8-round][i];
            }
        }
    }
    else{
        for(round=0;round<9;round++){
            for(i=0;i<4;i++){
                state[i] ^= Rkey[round][i];
            }
            SubBytes(state,dec);
            ShiftRows(state,dec);
            MixColumns(state,dec);
        }
        for(i=0;i<4;i++){
            state[i] ^= Rkey[9][i];
        }
        SubBytes(state,dec);
        ShiftRows(state,dec);
        for(i=0;i<4;i++){
            state[i] ^= Rkey[10][i];
        }
    }
}
```

La couche de substitution utilisée par AES_32b est donnée par le code suivant :

```
static const uint8_t sbox[2][256] = {{
  0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
  0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
  0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
  0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
  0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
  0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
  0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
  0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
  0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
  0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
  0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
  0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
  0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
  0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
  0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
  0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16
  },{
```

```
    0x52,0x09,0x6a,0xd5,0x30,0x36,0xa5,0x38,0xbf,0x40,0xa3,0x9e,0x81,0xf3,0xd7,0xfb,
    0x7c,0xe3,0x39,0x82,0x9b,0x2f,0xff,0x87,0x34,0x8e,0x43,0x44,0xc4,0xde,0xe9,0xcb,
    0x54,0x7b,0x94,0x32,0xa6,0xc2,0x23,0x3d,0xee,0x4c,0x95,0x0b,0x42,0xfa,0xc3,0x4e,
    0x08,0x2e,0xa1,0x66,0x28,0xd9,0x24,0xb2,0x76,0x5b,0xa2,0x49,0x6d,0x8b,0xd1,0x25,
    0x72,0xf8,0xf6,0x64,0x86,0x68,0x98,0x16,0xd4,0xa4,0x5c,0xcc,0x5d,0x65,0xb6,0x92,
    0x6c,0x70,0x48,0x50,0xfd,0xed,0xb9,0xda,0x5e,0x15,0x46,0x57,0xa7,0x8d,0x9d,0x84,
    0x90,0xd8,0xab,0x00,0x8c,0xbc,0xd3,0x0a,0xf7,0xe4,0x58,0x05,0xb8,0xb3,0x45,0x06,
    0xd0,0x2c,0x1e,0x8f,0xca,0x3f,0x0f,0x02,0xc1,0xaf,0xbd,0x03,0x01,0x13,0x8a,0x6b,
    0x3a,0x91,0x11,0x41,0x4f,0x67,0xdc,0xea,0x97,0xf2,0xcf,0xce,0xf0,0xb4,0xe6,0x73,
    0x96,0xac,0x74,0x22,0xe7,0xad,0x35,0x85,0xe2,0xf9,0x37,0xe8,0x1c,0x75,0xdf,0x6e,
    0x47,0xf1,0x1a,0x71,0x1d,0x29,0xc5,0x89,0x6f,0xb7,0x62,0x0e,0xaa,0x18,0xbe,0x1b,
    0xfc,0x56,0x3e,0x4b,0xc6,0xd2,0x79,0x20,0x9a,0xdb,0xc0,0xfe,0x78,0xcd,0x5a,0xf4,
    0x1f,0xdd,0xa8,0x33,0x88,0x07,0xc7,0x31,0xb1,0x12,0x10,0x59,0x27,0x80,0xec,0x5f,
    0x60,0x51,0x7f,0xa9,0x19,0xb5,0x4a,0x0d,0x2d,0xe5,0x7a,0x9f,0x93,0xc9,0x9c,0xef,
    0xa0,0xe0,0x3b,0x4d,0xae,0x2a,0xf5,0xb0,0xc8,0xeb,0xbb,0x3c,0x83,0x53,0x99,0x61,
    0x17,0x2b,0x04,0x7e,0xba,0x77,0xd6,0x26,0xe1,0x69,0x14,0x63,0x55,0x21,0x0c,0x7d
    }};
void SubBytes(uint32_t state[4], uint8_t dec){
    uint8_t i;
    for(i=0;i<4;i++){
        state[i] = (sbox[dec][(state[i]>>24)&0xff]<<24);
        state[i] |= (sbox[dec][(state[i]>>16)&0xff]<<16);
        state[i] |= (sbox[dec][(state[i]>>8)&0xff]<<8);
        state[i] |= (sbox[dec][state[i]&0xff]);
    }
}
```

L'étage linéaire utilisé par AES_32b est donné par le code suivant :

```
void ShiftRows(uint32_t state[4], uint8_t dec){
    state[1] = (state[1]<<8*(2*dec+1))|(state[1]>>8*(3-2*dec));
    state[2] = (state[2]<<16)|(state[2]>>16);
    state[3] = (state[3]<<8*(3-2*dec))|(state[3]>>8*(2*dec+1));
}
void MixColumns(uint32_t state[4], uint8_t dec){
    uint8_t i;
    uint32_t out[4],t;
    out[0] = state[1]^state[2]^state[3];
    out[1] = state[0]^state[2]^state[3];
    out[2] = state[0]^state[1]^state[3];
    out[3] = state[0]^state[1]^state[2];
    for(i=0;i<4;i++){
        t = (state[i]>>7)&0x01010101;
        t = t*0x1b;
        state[i] = ((state[i]<<1)&0xfefefefe)^t;
    }
    out[0] ^= state[0]^state[1];
    out[1] ^= state[1]^state[2];
    out[2] ^= state[2]^state[3];
    out[3] ^= state[0]^state[3];
    if(dec){
        for(i=0;i<4;i++){
            t = (state[i]>>7)&0x01010101;
            t = t*0x1b;
            state[i] = ((state[i]<<1)&0xfefefefe)^t;
        }
        out[0] ^= state[0]^state[2];
        out[1] ^= state[1]^state[3];
        out[2] ^= state[0]^state[2];
        out[3] ^= state[1]^state[3];
        for(i=0;i<4;i++){
            t = (state[i]>>7)&0x01010101;
```

```
            t = t*0x1b;
            state[i] = ((state[i]<<1)&0xfefefefe)^t;
        }
        for(i=0;i<4;i++){
            for(t=0;t<4;t++){
                out[i] ^= state[t];
            }
        }
    }
    for(i=0;i<4;i++){
        state[i] = out[i];
    }
}
```

A.1.2 — Parallélisme par tranches de bits

L'implémentation de l'AES avec le parallélisme par tranches de bits que nous avons utilisée, décrite dans [SCHWABE et al. 2016], prend en entrée 32 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 127$ correspond à la concaténation du bit n de chaque message), 32 ensembles de 11 clés de tours stockées dans les registres "Rkey" (le registre $0 \leq n \leq 127$ de la clé de tour $0 \leq k \leq 10$ correspond à la concaténation du bit n de chaque clé utilisée au tour k), et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void AES_BS(uint32_t state[128], const uint32_t Rkey[11][128], uint8_t dec){
    uint8_t i,round;
    if(dec){
        for(i=0;i<128;i++){
            state[i] ^= Rkey[10][i];
        }
        ShiftRows(state,dec);
        SubBytes(state,dec);
        for(i=0;i<128;i++){
            state[i] ^= Rkey[9][i];
        }
        for(round=0;round<9;round++){
            MixColumns(state,dec);
            ShiftRows(state,dec);
            SubBytes(state,dec);
            for(i=0;i<128;i++){
                state[i] ^= Rkey[8-round][i];
            }
        }
    }
    else{
        for(round=0;round<9;round++){
            for(i=0;i<128;i++){
                state[i] ^= Rkey[round][i];
            }
            SubBytes(state,dec);
            ShiftRows(state,dec);
            MixColumns(state,dec);
        }
        for(i=0;i<128;i++){
            state[i] ^= Rkey[9][i];
        }
        SubBytes(state,dec);
        ShiftRows(state,dec);
        for(i=0;i<128;i++){
            state[i] ^= Rkey[10][i];
        }
    }
}
```

La couche de substitution utilisée par AES_BS est donnée par le code suivant :

```
void SubBytes(uint32_t state[128], uint8_t dec){
    uint8_t i=0;
    uint32_t T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,T17,T18,T19,T20,
    T21,T22,T23,T24,T25,T26,T27,M1,M2,M3,M4,M5,M6,M7,M8,M9,M10,M11,M12,M13,
    M14,M15,M16,M17,M18,M19,M20,M21,M22,M23,M24,M25,M26,M27,M28,M29,M30,
    M31,M32,M33,M34,M35,M36,M37,M38,M39,M40,M41,M42,M43,M44,M45,M46,M47,
    M48,M49,M50,M51,M52,M53,M54,M55,M56,M57,M58,M59,M60,M61,M62,M63,P0,P1,
    P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,P14,P15,P16,P17,P18,P19,P20,P21,P22,P23,
    P24,P25,P26,P27,P28,P29,Y5,R5,R13,R17,R18,R19;
```

```
    for(i=0;i<128;i=i+8){
        if(dec){
            T23= state[i]^state[i+3];
            T22= ~(state[i+1]^state[i+3]);
            T2 = ~(state[i]^state[i+1]);
            T1 = state[i+3]^state[i+4];
            T24= ~(state[i+4]^state[i+7]);
            R5 = state[i+6]^state[i+7];
            T8 = ~(state[i+1]^T23);
            T19= T22^R5;
            T9 = ~(state[i+7]^T1);
            T10= T2^T24;
            T13= T2^R5;
            T3 = T1^R5;
            T25= ~(state[i+2]^T1);
            R13= state[i+1]^state[i+6];
            T17= ~(state[i+2]^T19);
            T20= T24^R13;
            T4 = state[i+4]^T8;
            R17= ~(state[i+2]^state[i+5]);
            R18= ~(state[i+5]^state[i+6]);
            R19= ~(state[i+2]^state[i+4]);
            Y5 = state[i]^R17;
            T6 = T22^R17;
            T16= R13^R19;
            T27= T1^R18;
            T15= T10^T27;
            T14= T10^R18;
            T26= T3^T16;
            M1= T13&T6;
            M2= T23&T8;
            M3= T14^M1;
            M4= T19&Y5;
            M5= M4^M1;
            M6= T3&T16;
            M7= T22&T9;
            M8= T26^M6;
            M9= T20&T17;
            M10= M9^M6;
            M11= T1&T15;
            M12= T4&T27;
            M13= M12^M11;
            M14= T2&T10;
            M15= M14^M11;
            M16= M3^M2;
            M17= M5^T24;
            M18= M8^M7;
            M19= M10^M15;
            M20= M16^M13;
            M21= M17^M15;
            M22= M18^M13;
            M23= M19^T25;
            M24= M22^M23;
            M25= M22&M20;
            M26= M21^M25;
            M27= M20^M21;
            M28= M23^M25;
            M29= M28&M27;
            M30= M26&M24;
            M31= M20&M23;
            M32= M27&M31;
            M33= M27^M25;
            M34= M21&M22;
            M35= M24&M34;
            M36= M24^M25;
            M37= M21^M29;
            M38= M32^M33;
            M39= M23^M30;
            M40= M35^M36;
            M41= M38^M40;
            M42= M37^M39;
            M43= M37^M38;
            M44= M39^M40;
            M45= M42^M41;
            M46= M44&T6;
            M47= M40&T8;
            M48= M39&Y5;
            M49= M43&T16;
            M50= M38&T9;
            M51= M37&T17;
            M52= M42&T15;
            M53= M45&T27;
            M54= M41&T10;
            M55= M44&T13;
            M56= M40&T23;
            M57= M39&T19;
            M58= M43&T3;
            M59= M38&T22;
            M60= M37&T20;
            M61= M42&T1;
            M62= M45&T4;
            M63= M41&T2;
            P0 = M52^M61;
            P1 = M58^M59;
            P2 = M54^M62;
            P3 = M47^M50;
            P4 = M48^M56;
            P5 = M46^M51;
            P6 = M49^M60;
            P7 = P0^P1;
            P8 = M50^M53;
            P9 = M55^M63;
            P10= M57^P4;
            P11= P0^P3;
            P12= M46^M48;
            P13= M49^M51;
            P14= M49^M62;
            P15= M54^M59;
            P16= M57^M61;
            P17= M58^P2;
            P18= M63^P5;
            P19= P2^P3;
            P20= P4^P6;
            P22= P2^P7;
            P23= P7^P8;
            P24= P5^P7;
            P25= P6^P10;
            P26= P9^P11;
            P27= P10^P18;
            P28= P11^P25;
            P29= P15^P20;
            state[i]   = P13^P22;
            state[i+1] = P26^P29;
            state[i+2] = P17^P28;
            state[i+3] = P12^P22;
            state[i+4] = P23^P27;
            state[i+5] = P19^P24;
            state[i+6] = P14^P23;
            state[i+7] = P9^P16;
        }
```

```
        else{
            T1 = state[i]^state[i+3];
            T2 = state[i]^state[i+5];
            T3 = state[i]^state[i+6];
            T4 = state[i+3]^state[i+5];
            T5 = state[i+4]^state[i+6];
            T6 = T1^T5;
            T7 = state[i+1]^state[i+2];
            T8 = state[i+7]^T6;
            T9 = state[i+7]^T7;
            T10= T6^T7;
            T11= state[i+1]^state[i+5];
            T12= state[i+2]^state[i+5];
            T13= T3^T4;
            T14= T6^T11;
            T15= T5^T11;
            T16= T5^T12;
            T17= T9^T16;
            T18= state[i+3]^state[i+7];
            T19= T7^T18;
            T20= T1^T19;
            T21= state[i+6]^state[i+7];
            T22= T7^T21;
            T23= T2^T22;
            T24= T2^T10;
            T25= T20^T17;
            T26= T3^T16;
            T27= T1^T12;
            M1= T13&T6;
            M2= T23&T8;
            M3= T14^M1;
            M4= T19&state[i+7];
            M5= M4^M1;
            M6= T3&T16;
            M7= T22&T9;
            M8= T26^M6;
            M9= T20&T17;
            M10= M9^M6;
            M11= T1&T15;
            M12= T4&T27;
            M13= M12^M11;
            M14= T2&T10;
            M15= M14^M11;
            M16= M3^M2;
            M17= M5^T24;
            M18= M8^M7;
            M19= M10^M15;
            M20= M16^M13;
            M21= M17^M15;
            M22= M18^M13;
            M23= M19^T25;
            M24= M22^M23;
            M25= M22&M20;
            M26= M21^M25;
            M27= M20^M21;
            M28= M23^M25;
            M29= M28&M27;
            M30= M26&M24;
            M31= M20&M23;
            M32= M27&M31;
            M33= M27^M25;
            M34= M21&M22;
            M35= M24&M34;
            M36= M24^M25;
            M37= M21^M29;
            M38= M32^M33;
            M39= M23^M30;
            M40= M35^M36;
            M41= M38^M40;
            M42= M37^M39;
            M43= M37^M38;
            M44= M39^M40;
            M45= M42^M41;
            M46= M44&T6;
            M47= M40&T8;
            M48= M39&state[i+7];
            M49= M43&T16;
            M50= M38&T9;
            M51= M37&T17;
            M52= M42&T15;
            M53= M45&T27;
            M54= M41&T10;
            M55= M44&T13;
            M56= M40&T23;
            M57= M39&T19;
            M58= M43&T3;
            M59= M38&T22;
            M60= M37&T20;
            M61= M42&T1;
            M62= M45&T4;
            M63= M41&T2;
            P0 = M61^M62;
            P1 = M50^M56;
            P2 = M46^M48;
            P3 = M47^M55;
            P4 = M54^M58;
            P5 = M49^M61;
            P6 = M62^P5;
            P7 = M46^P3;
            P8 = M51^M59;
            P9 = M52^M53;
            P10= M53^P4;
            P11= M60^P2;
            P12= M48^M51;
            P13= M50^P0;
            P14= M52^M61;
            P15= M55^P1;
            P16= M56^P0;
            P17= M57^P1;
            P18= M58^P8;
            P19= M63^P4;
            P20= P0^P1;
            P21= P1^P7;
            P22= P3^P12;
            P23= P18^P2;
            P24= P15^P9;
            P25= P6^P10;
            P26= P7^P9;
            P27= P8^P10;
            P28= P11^P14;
            P29= P11^P17;
            state[i] = P6^P24;
            state[i+1] = ~(P16^P26);
            state[i+2] = ~(P19^P28);
            state[i+3] = P6^P21;
            state[i+4] = P20^P22;
            state[i+5] = P25^P29;
            state[i+6] = ~(P13^P27);
            state[i+7] = ~(P6^P23);
        }
    }
}
```

où $\sim X$ ou $\overline{X}$ est l'inversion binaire de X. Les rotations des lignes effectuées lors de l'étage linéaire utilisé par AES_BS sont données par le code suivant :

```
void ShiftRows(uint32_t state[128], uint8_t dec){
    uint8_t i;
    uint32_t Temp[16];
    if(dec){
        for(i=0;i<8;i++){
            Temp[i] = state[i+56];
        }
        for(i=63;i>39;i--){
            state[i] = state[i-8];
        }
        for(i=0;i<8;i++){
            state[i+32] = Temp[i];
        }
    }
    else{
        for(i=0;i<8;i++){
            Temp[i] = state[i+32];
        }
        for(i=32;i<56;i++){
            state[i] = state[i+8];
        }
        for(i=0;i<8;i++){
            state[i+56] = Temp[i];
        }
    }
    for(i=0;i<16;i++){
        Temp[i] = state[i+64];
    }
    for(i=64;i<80;i++){
        state[i] = state[i+16];
    }
    for(i=0;i<16;i++){
        state[i+80] = Temp[i];
    }
    if(dec){
        for(i=0;i<8;i++){
            Temp[i] = state[i+96];
        }
        for(i=96;i<120;i++){
            state[i] = state[i+8];
        }
        for(i=0;i<8;i++){
            state[i+120] = Temp[i];
        }
    }
    else{
        for(i=0;i<8;i++){
            Temp[i] = state[i+120];
        }
        for(i=127;i>103;i--){
            state[i] = state[i-8];
        }
        for(i=0;i<8;i++){
            state[i+96] = Temp[i];
        }
    }
}
```

L'application des matrices lors de l'étage linéaire utilisé par AES_BS est donnée par le code suivant :

```
void MixColumns(uint32_t state[128], uint8_t dec){
    uint8_t i,j;
    uint32_t out[128],t;
    for(i=0;i<32;i++){
        out[i] = state[i+32]^state[i+64]^state[i+96];
        out[i+32] = state[i]^state[i+64]^state[i+96];
        out[i+64] = state[i]^state[i+32]^state[i+96];
        out[i+96] = state[i]^state[i+32]^state[i+64];
    }
    for(i=0;i<128;i+=8){
        t = state[i];
        for(j=0;j<7;j++){
            state[i+j] = state[i+j+1];
        }
        state[i+3] ^= t;
        state[i+4] ^= t;
        state[i+6] ^= t;
        state[i+7] = t;
    }
    for(i=0;i<32;i++){
        out[i] ^= state[i]^state[i+32];
        out[i+32] ^= state[i+32]^state[i+64];
```

```
23          out[i+64] ^= state[i+64]^state[i+96];
24          out[i+96] ^= state[i]^state[i+96];
25      }
26      if(dec){
27          for(i=0;i<128;i+=8){
28              t = state[i];
29              for(j=0;j<7;j++){
30                  state[i+j] = state[i+j+1];
31              }
32              state[i+3] ^= t;
33              state[i+4] ^= t;
34              state[i+6] ^= t;
35              state[i+7] = t;
36          }
37          for(i=0;i<32;i++){
38              out[i] ^= state[i]^state[i+64];
39              out[i+32] ^= state[i+32]^state[i+96];
40              out[i+64] ^= state[i]^state[i+64];
41              out[i+96] ^= state[i+32]^state[i+96];
42          }
43          for(i=0;i<128;i+=8){
44              t = state[i];
45              for(j=0;j<7;j++){
46                  state[i+j] = state[i+j+1];
47              }
48              state[i+3] ^= t;
49              state[i+4] ^= t;
50              state[i+6] ^= t;
51              state[i+7] = t;
52          }
53          for(i=0;i<32;i++){
54              out[i] ^= state[i]^state[i+32]^state[i+64]^state[i+96];
55              out[i+32] ^= state[i]^state[i+32]^state[i+64]^state[i+96];
56              out[i+64] ^= state[i]^state[i+32]^state[i+64]^state[i+96];
57              out[i+96] ^= state[i]^state[i+32]^state[i+64]^state[i+96];
58          }
59      }
60      for(i=0;i<128;i++){
61          state[i] = out[i];
62      }
63  }
```

Cette implémentation permet d'appliquer l'AES sur 32 blocs en entrée, et exploite donc pleinement une architecture 32 bits, mais il est également possible de remplacer tous les mots de 32 bits par des mots de taille quelconque d afin d'appliquer l'AES sur d blocs en entrée pour exploiter une architecture d bits.

Afin d'appliquer le masquage sur l'AES, il est possible d'utiliser le schéma décrit en section 2.6. Il suffit pour cela que chaque mot de 32 bits en entrée soit composé d'un bit de chacun des 16 masques et d'un bit de chacun des 16 blocs de données masquées comme illustré à la figure 2.11, de ne pas effectuer l'inversion binaire sur le masque, de ne pas effectuer le XOR entre le masque et la clé, et que chaque opération & soit modifiée afin de calculer correctement la valeur des masques comme détaillé en section 2.6.

De plus, afin d'appliquer l'IIR tel que décrit en section 3.5, il suffit que chaque mot de 32 bits soit composé de deux copies d'un bit de chacun des 15 blocs de données et d'un bit de chacun des 2 blocs de référence comme illustré à la figure 3.15.

Enfin, pour combiner le masquage et l'IIR, il suffit que chaque mot de 32 bits en entrée soit composé de deux copies d'un bit de chacun des 10 blocs de données, d'un bit de chacun des 11 masques et d'un bit du bloc de référence comme illustré à la figure 5.6, et que chaque opération & soit modifiée afin de calculer correctement la valeur des masques avec la technique présentée en section 2.4.2.

A.2 Implémentations de Fantomas

Les spécifications de Fantomas sont données dans [GROSSO et al. 2015]. Le message clair, le chiffré ainsi que les clés sont stockés de gauche à droite dans les registres.

A.2.1 Implémentation 32 bits

Nous avons proposé une implémentation 32 bits de Fantomas à partir de son code de référence afin d'exploiter pleinement une architecture 32 bits. Elle prend en entrée un message clair ou un chiffré stocké dans les registres "state", la clé secrète stockée dans les registres "key" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void Fantomas_32b(uint32_t state[4], uint32_t key[4], uint8_t dec){
    uint8_t round,i;
    uint32_t temp;
    for(i=0;i<4;i++){
        state[i] ^= key[i];
    }
    for(round=0;round<12;round++){
        if(dec^1){
            state[0] ^= (LBox1[0][round+1]<<16);
            Slayer(state,dec);
        }
        for(i=0;i<4;i++){
            temp = (LBox2[dec][(state[i]>>24)&0xff]^LBox1[dec][(state[i]>>16)&0xff])<<16;
            state[i] = temp|(LBox2[dec][(state[i]>>8)&0xff]^LBox1[dec][state[i]&0xff]);
        }
        if(dec){
            Slayer(state,dec);
            state[0] ^= (LBox1[0][12-round])<<16;
        }
        for(i=0;i<4;i++){
            state[i] ^= key[i];
        }
    }
}
```

La couche de substitution utilisée par Fantomas_32b est donnée par le code suivant :

```
void Slayer(uint32_t X[4], uint8_t dec){
    uint32_t Temp[2];
    if(dec){
        X[0] ^= (X[0]>>16);
        X[0] ^= (X[1]&X[2])>>16;
        X[2] ^= (X[0]<<16);
        X[0] ^= (X[0]&X[1])<<16;
        X[1] ^= (X[1]<<16);
        X[1] ^= (X[0]&X[2])>>16;
        X[0] ^= (X[1]>>16);
        X[1] ^= X[0]&(X[0]<<16);
        X[2] ^= (X[0]>>16);
        X[3] ^= (X[0]<<16);
        X[3] ^= (X[1]>>16);
        Temp[0] = X[2];
        Temp[1] = X[3];
        X[2] ^= (((~Temp[1])>>16)&Temp[1]);
        X[3] ^= (((~Temp[1])&Temp[0])<<16);
        X[3] ^= ((~Temp[0])&(Temp[1]>>16));
        X[1] ^= 0x0000ffff;
        X[2] ^= 0xffff0000;
        X[0] ^= (X[2]<<16);
        X[0] ^= (X[3]>>16);
        X[1] ^= (X[3]<<16);
        X[0] ^= (X[0]>>16);
        X[0] ^= (X[1]&X[2])>>16;
        X[2] ^= (X[0]<<16);
        X[0] ^= (X[0]&X[1])<<16;
        X[1] ^= (X[1]<<16);
        X[1] ^= (X[0]&X[2])>>16;
        X[0] ^= (X[1]>>16);
        X[1] ^= X[0]&(X[0]<<16);
    }
    else{
        X[1] ^= X[0]&(X[0]<<16);
        X[0] ^= (X[1]>>16);
        X[1] ^= (X[0]&X[2])>>16;
        X[1] ^= (X[1]<<16);
        X[0] ^= (X[0]&X[1])<<16;
        X[2] ^= (X[0]<<16);
        X[0] ^= (X[1]&X[2])>>16;
        X[0] ^= (X[0]>>16);
        X[0] ^= (X[2]<<16);
        X[0] ^= (X[3]>>16);
        X[1] ^= (X[3]<<16);
        X[1] ^= 0x0000ffff;
        X[2] ^= 0xffff0000;
        Temp[0] = X[2];
        Temp[1] = X[3];
        X[2] ^= (((~Temp[1])>>16)&Temp[1]);
        X[3] ^= (((~Temp[1])&Temp[0])<<16);
        X[3] ^= ((~Temp[0])&(Temp[1]>>16));
        X[2] ^= (X[0]>>16);
        X[3] ^= (X[0]<<16);
        X[3] ^= (X[1]>>16);
        X[1] ^= X[0]&(X[0]<<16);
        X[0] ^= (X[1]>>16);
        X[1] ^= (X[0]&X[2])>>16;
        X[1] ^= (X[1]<<16);
        X[0] ^= (X[0]&X[1])<<16;
        X[2] ^= (X[0]<<16);
        X[0] ^= (X[1]&X[2])>>16;
        X[0] ^= (X[0]>>16);
    }
}
```

où $\sim X$ ou $\overline{X}$ est l'inversion binaire de X. Les tableaux utilisés par l'étage de Fantomas_32b sont donnés par le code suivant :

```
uint16_t LBox1[2][256] = {{
 0x0000,0xbfff,0x6e90,0xd16f,0x4137,0xfec8,0x2fa7,0x9058,0xd548,0x6ab7,0xbbd8,0x0427,
 0x947f,0x2b80,0xfaef,0x4510,0x5296,0xed69,0x3c06,0x83f9,0x13a1,0xac5e,0x7d31,0xc2ce,
 0x87de,0x3821,0xe94e,0x56b1,0xc6e9,0x7916,0xa879,0x1786,0xd183,0x6e7c,0xbf13,0x00ec,
 0x90b4,0x2f4b,0xfe24,0x41db,0x04cb,0xbb34,0x6a5b,0xd5a4,0x45fc,0xfa03,0x2b6c,0x9493,
 0x8315,0x3cea,0xed85,0x527a,0xc222,0x7ddd,0xacb2,0x134d,0x565d,0xe9a2,0x38cd,0x8732,
 0x176a,0xa895,0x79fa,0xc605,0xc2b1,0x7d4e,0xac21,0x13de,0x8386,0x3c79,0xed16,0x52e9,
 0x17f9,0xa806,0x7969,0xc696,0x56ce,0xe931,0x385e,0x87a1,0x9027,0x2fd8,0xfeb7,0x4148,
 0xd110,0x6eef,0xbf80,0x007f,0x456f,0xfa90,0x2bff,0x9400,0x0458,0xbba7,0x6ac8,0xd537,
 0x1332,0xaccd,0x7da2,0xc25d,0x5205,0xedfa,0x3c95,0x836a,0xc67a,0x7985,0xa8ea,0x1715,
 0x874d,0x38b2,0xe9dd,0x5622,0x41a4,0xfe5b,0x2f34,0x90cb,0x0093,0xbf6c,0x6e03,0xd1fc,
 0x94ec,0x2b13,0xfa7c,0x4583,0xd5db,0x6a24,0xbb4b,0x04b4,0x496c,0xf693,0x27fc,0x9803,
 0x085b,0xb7a4,0x66cb,0xd934,0x9c24,0x23db,0xf2b4,0x4d4b,0xdd13,0x62ec,0xb383,0x0c7c,
 0x1bfa,0xa405,0x756a,0xca95,0x5acd,0xe532,0x345d,0x8ba2,0xceb2,0x714d,0xa022,0x1fdd,
 0x8f85,0x307a,0xe115,0x5eea,0x98ef,0x2710,0xf67f,0x4980,0xd9d8,0x6627,0xb748,0x08b7,
 0x4da7,0xf258,0x2337,0x9cc8,0x0c90,0xb36f,0x6200,0xddff,0xca79,0x7586,0xa4e9,0x1b16,
 0x8b4e,0x34b1,0xe5de,0x5a21,0x1f31,0xa0ce,0x71a1,0xce5e,0x5e06,0xe1f9,0x3096,0x8f69,
 0x8bdd,0x3422,0xe54d,0x5ab2,0xcaea,0x7515,0xa47a,0x1b85,0x5e95,0xe16a,0x3005,0x8ffa,
 0x1fa2,0xa05d,0x7132,0xcecd,0xd94b,0x66b4,0xb7db,0x0824,0x987c,0x2783,0xf6ec,0x4913,
```

```
0x0c03,0xb3fc,0x6293,0xdd6c,0x4d34,0xf2cb,0x23a4,0x9c5b,0x5a5e,0xe5a1,0x34ce,0x8b31,
0x1b69,0xa496,0x75f9,0xca06,0x8f16,0x30e9,0xe186,0x5e79,0xce21,0x71de,0xa0b1,0x1f4e,
0x08c8,0xb737,0x6658,0xd9a7,0x49ff,0xf600,0x276f,0x9890,0xdd80,0x627f,0xb310,0x0cef,
0x9cb7,0x2348,0xf227,0x4dd8
},{
0x0000,0xb065,0x8935,0x3950,0x4a95,0xfaf0,0xc3a0,0x73c5,0xb489,0x04ec,0x3dbc,0x8dd9,
0xfe1c,0x4e79,0x7729,0xc74c,0x2857,0x9832,0xa162,0x1107,0x62c2,0xd2a7,0xebf7,0x5b92,
0x9cde,0x2cbb,0x15eb,0xa58e,0xd64b,0x662e,0x5f7e,0xef1b,0x26c5,0x96a0,0xaff0,0x1f95,
0x6c50,0xdc35,0xe565,0x5500,0x924c,0x2229,0x1b79,0xab1c,0xd8d9,0x68bc,0x51ec,0xe189,
0x0e92,0xbef7,0x87a7,0x37c2,0x4407,0xf462,0xcd32,0x7d57,0xba1b,0x0a7e,0x332e,0x834b,
0xf08e,0x40eb,0x79bb,0xc9de,0xc989,0x79ec,0x40bc,0xf0d9,0x831c,0x3379,0x0a29,0xba4c,
0x7d00,0xcd65,0xf435,0x4450,0x3795,0x87f0,0xbea0,0x0ec5,0xe1de,0x51bb,0x68eb,0xd88e,
0xab4b,0x1b2e,0x227e,0x921b,0x5557,0xe532,0xdc62,0x6c07,0x1fc2,0xafa7,0x96f7,0x2692,
0xef4c,0x5f29,0x6679,0xd61c,0xa5d9,0x15bc,0x2cec,0x9c89,0x5bc5,0xeba0,0xd2f0,0x6295,
0x1150,0xa135,0x9865,0x2800,0xc71b,0x777e,0x4e2e,0xfe4b,0x8d8e,0x3deb,0x04bb,0xb4de,
0x7392,0xc3f7,0xfaa7,0x4ac2,0x3907,0x8962,0xb032,0x0057,0x1173,0xa116,0x9846,0x2823,
0x5be6,0xeb83,0xd2d3,0x62b6,0xa5fa,0x159f,0x2ccf,0x9caa,0xef6f,0x5f0a,0x665a,0xd63f,
0x3924,0x8941,0xb011,0x0074,0x73b1,0xc3d4,0xfa84,0x4ae1,0x8dad,0x3dc8,0x0498,0xb4fd,
0xc738,0x775d,0x4e0d,0xfe68,0x37b6,0x87d3,0xbe83,0x0ee6,0x7d23,0xcd46,0xf416,0x4473,
0x833f,0x335a,0x0a0a,0xba6f,0xc9aa,0x79cf,0x409f,0xf0fa,0x1fe1,0xaf84,0x96d4,0x26b1,
0x5574,0xe511,0xdc41,0x6c24,0xab68,0x1b0d,0x225d,0x9238,0xe1fd,0x5198,0x68c8,0xd8ad,
0xd8fa,0x689f,0x51cf,0xe1aa,0x926f,0x220a,0x1b5a,0xab3f,0x6c73,0xdc16,0xe546,0x5523,
0x26e6,0x9683,0xafd3,0x1fb6,0xf0ad,0x40c8,0x7998,0xc9fd,0xba38,0x0a5d,0x330d,0x8368,
0x4424,0xf441,0xcd11,0x7d74,0x0eb1,0xbed4,0x8784,0x37e1,0xfe3f,0x4e5a,0x770a,0xc76f,
0xb4aa,0x04cf,0x3d9f,0x8dfa,0x4ab6,0xfad3,0xc383,0x73e6,0x0023,0xb046,0x8916,0x3973,
0xd668,0x660d,0x5f5d,0xef38,0x9cfd,0x2c98,0x15c8,0xa5ad,0x62e1,0xd284,0xebd4,0x5bb1,
0x2874,0x9811,0xa141,0x1124
}};
uint16_t LBox2[2][256]={{
0x0000,0x74c2,0xd324,0xa7e6,0xce28,0xbaea,0x1d0c,0x69ce,0x6856,0x1c94,0xbb72,0xcfb0,
0xa67e,0xd2bc,0x755a,0x0198,0xe489,0x904b,0x37ad,0x436f,0x2aa1,0x5e63,0xf985,0x8d47,
0x8cdf,0xf81d,0x5ffb,0x2b39,0x42f7,0x3635,0x91d3,0xe511,0x6839,0x1cfb,0xbb1d,0xcfdf,
0xa611,0xd2d3,0x7535,0x01f7,0x006f,0x74ad,0xd34b,0xa789,0xce47,0xba85,0x1d63,0x69a1,
0x8cb0,0xf872,0x5f94,0x2b56,0x4298,0x365a,0x91bc,0xe57e,0xe4e6,0x9024,0x37c2,0x4300,
0x2ace,0x5e0c,0xf9ea,0x8d28,0x5e44,0x2a86,0x8d60,0xf9a2,0x906c,0xe4ae,0x4348,0x378a,
0x3612,0x42d0,0xe536,0x91f4,0xf83a,0x8cf8,0x2b1e,0x5fdc,0xbacd,0xce0f,0x69e9,0x1d2b,
0x74e5,0x0027,0xa7c1,0xd303,0xd29b,0xa659,0x01bf,0x757d,0x1cb3,0x6871,0xcf97,0xbb55,
0x367d,0x42bf,0xe559,0x919b,0xf855,0x8c97,0x2b71,0x5fb3,0x5e2b,0x2ae9,0x8d0f,0xf9cd,
0x9003,0xe4c1,0x4327,0x37e5,0xd2f4,0xa636,0x01d0,0x7512,0x1cdc,0x681e,0xcff8,0xbb3a,
0xbaa2,0xce60,0x6986,0x1d44,0x748a,0x0048,0xa7ae,0xd36c,0x614b,0x1589,0xb26f,0xc6ad,
0xaf63,0xdba1,0x7c47,0x0885,0x091d,0x7ddf,0xda39,0xaefb,0xc735,0xb3f7,0x1411,0x60d3,
0x85c2,0xf100,0x56e6,0x2224,0x4bea,0x3f28,0x98ce,0xec0c,0xed94,0x9956,0x3eb0,0x4a72,
0x23bc,0x577e,0xf098,0x845a,0x0972,0x7db0,0xda56,0xae94,0xc75a,0xb398,0x147e,0x60bc,
0x6124,0x15e6,0xb200,0xc6c2,0xaf0c,0xdbce,0x7c28,0x08ea,0xedfb,0x9939,0x3edf,0x4a1d,
0x23d3,0x5711,0xf0f7,0x8435,0x85ad,0xf16f,0x5689,0x224b,0x4b85,0x3f47,0x98a1,0xec63,
0x3f0f,0x4bcd,0xec2b,0x98e9,0xf127,0x85e5,0x2203,0x56c1,0x5759,0x239b,0x847d,0xf0bf,
0x9971,0xedb3,0x4a55,0x3e97,0xdb86,0xaf44,0x08a2,0x7c60,0x15ae,0x616c,0xc68a,0xb248,
0xb3d0,0xc712,0x60f4,0x1436,0x7df8,0x093a,0xaedc,0xda1e,0x5736,0x23f4,0x8412,0xf0d0,
0x991e,0xeddc,0x4a3a,0x3ef8,0x3f60,0x4ba2,0xec44,0x9886,0xf148,0x858a,0x226c,0x56ae,
0xb3bf,0xc77d,0x609b,0x1459,0x7d97,0x0955,0xaeb3,0xda71,0xdbe9,0xaf2b,0x08cd,0x7c0f,
0x15c1,0x6103,0xc6e5,0xb227
},{
0x0000,0x82ad,0x4653,0xc4fe,0x550b,0xd7a6,0x1358,0x91f5,0x6c83,0xee2e,0x2ad0,0xa87d,
0x3988,0xbb25,0x7fdb,0xfd76,0x4339,0xc194,0x056a,0x87c7,0x1632,0x949f,0x5061,0xd2cc,
0x2fba,0xad17,0x69e9,0xeb44,0x7ab1,0xf81c,0x3ce2,0xbe4f,0xb903,0x3bae,0xff50,0x7dfd,
0xec08,0x6ea5,0xaa5b,0x28f6,0xd580,0x572d,0x93d3,0x117e,0x808b,0x0226,0xc6d8,0x4475,
0xfa3a,0x7897,0xbc69,0x3ec4,0xaf31,0x2d9c,0xe962,0x6bcf,0x96b9,0x1414,0xd0ea,0x5247,
```

```
0xc3b2,0x411f,0x85e1,0x074c,0xfffe,0x7d53,0xb9ad,0x3b00,0xaaf5,0x2858,0xeca6,0x6e0b,
0x937d,0x11d0,0xd52e,0x5783,0xc676,0x44db,0x8025,0x0288,0xbcc7,0x3e6a,0xfa94,0x7839,
0xe9cc,0x6b61,0xaf9f,0x2d32,0xd044,0x52e9,0x9617,0x14ba,0x854f,0x07e2,0xc31c,0x41b1,
0x46fd,0xc450,0x00ae,0x8203,0x13f6,0x915b,0x55a5,0xd708,0x2a7e,0xa8d3,0x6c2d,0xee80,
0x7f75,0xfdd8,0x3926,0xbb8b,0x05c4,0x8769,0x4397,0xc13a,0x50cf,0xd262,0x169c,0x9431,
0x6947,0xebea,0x2f14,0xadb9,0x3c4c,0xbee1,0x7a1f,0xf8b2,0x1669,0x94c4,0x503a,0xd297,
0x4362,0xc1cf,0x0531,0x879c,0x7aea,0xf847,0x3cb9,0xbe14,0x2fe1,0xad4c,0x69b2,0xeb1f,
0x5550,0xd7fd,0x1303,0x91ae,0x005b,0x82f6,0x4608,0xc4a5,0x39d3,0xbb7e,0x7f80,0xfd2d,
0x6cd8,0xee75,0x2a8b,0xa826,0xaf6a,0x2dc7,0xe939,0x6b94,0xfa61,0x78cc,0xbc32,0x3e9f,
0xc3e9,0x4144,0x85ba,0x0717,0x96e2,0x144f,0xd0b1,0x521c,0xec53,0x6efe,0xaa00,0x28ad,
0xb958,0x3bf5,0xff0b,0x7da6,0x80d0,0x027d,0xc683,0x442e,0xd5db,0x5776,0x9388,0x1125,
0xe997,0x6b3a,0xafc4,0x2d69,0xbc9c,0x3e31,0xfacf,0x7862,0x8514,0x07b9,0xc347,0x41ea,
0xd01f,0x52b2,0x964c,0x14e1,0xaaae,0x2803,0xecfd,0x6e50,0xffa5,0x7d08,0xb9f6,0x3b5b,
0xc62d,0x4480,0x807e,0x02d3,0x9326,0x118b,0xd575,0x57d8,0x5094,0xd239,0x16c7,0x946a,
0x059f,0x8732,0x43cc,0xc161,0x3c17,0xbeba,0x7a44,0xf8e9,0x691c,0xebb1,0x2f4f,0xade2,
0x13ad,0x9100,0x55fe,0xd753,0x46a6,0xc40b,0x00f5,0x8258,0x7f2e,0xfd83,0x397d,0xbbd0,
0x2a25,0xa888,0x6c76,0xeedb
}};
```

A.2.2 – Implémentation 16 bits appliquée sur 2 blocs

Nous avons utilisé l'implémentation 16 bits de référence de Fantomas appliquée de façon SIMD sur 2 blocs de données. Elle prend en entrée 2 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n + 15$ de chaque message), 2 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n + 15$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, 1 sinon. Son code est le suivant :

```
void Fantomas_16b(uint32_t state[8], uint32_t key[8], uint8_t dec){
    uint8_t round,i;
    uint32_t temp;
    for(i=0;i<8;i++){
        state[i] ^= key[i];
    }
    for(round=0;round<12;round++){
        if(dec^1){
            state[0] ^= LBox1[0][round+1]|(LBox1[0][round+1]<<16);
            Slayer(state,dec);
        }
        for(i=0;i<8;i++){
            temp = (LBox2[dec][(state[i]>>24)&0xff]^LBox1[dec][(state[i]>>16)&0xff])<<16;
            state[i] = temp|(LBox2[dec][(state[i]>>8)&0xff]^LBox1[dec][state[i]&0xff]);
        }
        if(dec){
            Slayer(state,dec);
            state[0] ^= LBox1[0][12-round]|(LBox1[0][12-round]<<16);
        }
        for(i=0;i<8;i++){
            state[i] ^= key[i];
        }
    }
}
```

La couche de substitution utilisée par Fantomas_16b est donnée par le code suivant :

```
void Slayer(uint32_t X[8], uint8_t dec){
    uint32_t Temp[3];
    if (dec){
        X[1] ^= X[0];
        X[1] ^= X[2]&X[4];
        X[4] ^= X[1];
        X[0] ^= X[1]&X[3];
        X[2] ^= X[3];
        X[3] ^= X[0]&X[4];
        X[1] ^= X[2];
        X[2] ^= X[0]&X[1];
        X[5] ^= X[0];
        X[6] ^= X[1];
        X[7] ^= X[2];
        Temp[0] = X[5];
        Temp[1] = X[6];
        Temp[2] = X[7];
        X[5] ^= (~Temp[1])&Temp[2];
        X[6] ^= (~Temp[2])&Temp[0];
        X[7] ^= (~Temp[0])&Temp[1];
        X[3] = ~X[3];
        X[4] = ~X[4];
        X[0] ^= X[5];
        X[1] ^= X[6];
        X[2] ^= X[7];
        X[1] ^= X[0];
        X[1] ^= X[2]&X[4];
        X[4] ^= X[1];
        X[0] ^= X[1]&X[3];
        X[2] ^= X[3];
        X[3] ^= X[0]&X[4];
        X[1] ^= X[2];
        X[2] ^= X[0]&X[1];
    }
    else{
        X[2] ^= X[0]&X[1];
        X[1] ^= X[2];
        X[3] ^= X[0]&X[4];
        X[2] ^= X[3];
        X[0] ^= X[1]&X[3];
        X[4] ^= X[1];
        X[1] ^= X[2]&X[4];
        X[1] ^= X[0];
        X[0] ^= X[5];
        X[1] ^= X[6];
        X[2] ^= X[7];
        X[3] = ~X[3];
        X[4] = ~X[4];
        Temp[0] = X[5];
        Temp[1] = X[6];
        Temp[2] = X[7];
        X[5] ^= (~Temp[1])&Temp[2];
        X[6] ^= (~Temp[2])&Temp[0];
        X[7] ^= (~Temp[0])&Temp[1];
        X[5] ^= X[0];
        X[6] ^= X[1];
        X[7] ^= X[2];
        X[2] ^= X[0]&X[1];
        X[1] ^= X[2];
        X[3] ^= X[0]&X[4];
        X[2] ^= X[3];
        X[0] ^= X[1]&X[3];
        X[4] ^= X[1];
        X[1] ^= X[2]&X[4];
        X[1] ^= X[0];
    }
}
```

où $\sim X$ ou $\overline{X}$ est l'inversion binaire de X. Les tableaux LBox1 et LBox2 utilisés par l'étage linéaire de Fantomas_16b sont les mêmes que ceux utilisés par Fantomas_32b donnés en annexe A.2.1.

Cette implémentation permet d'appliquer Fantomas sur 2 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer tous les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin d'exploiter une architecture 16 bits.

Afin d'appliquer le masquage sur Fantomas, il est possible d'utiliser le schéma décrit en section 2.6. Il suffit pour cela que chaque mot de 32 bits en entrée soit composé d'un mot de 16 bits du masque et d'un mot de 16 bits du bloc de données masquées comme illustré à la figure 2.11, de ne pas effectuer l'inversion binaire sur le masque, de ne pas effectuer le XOR entre le masque et la clé ou les constantes de tour, et que chaque opération & soit modifiée afin de calculer correctement la valeur des masques.

A.3 – Implémentations de GARFIELD

Les spécifications de GARFIELD sont données dans le chapitre 5. Le message clair, le chiffré ainsi que les clés sont stockés de gauche à droite dans les registres.

A.3.1 – Implémentation 32 bits

L'implémentation 32 bits de référence de GARFIELD prend en entrée un message clair ou un chiffré stocké dans les registres "state", la clé secrète stockée dans les registres "key" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Afin de réduire le coût de l'étage linéaire, les octets $B_0 \cdots B_{15}$ du message, de la clé et des constantes de tour sont stockés de la façon suivante :

$$\begin{aligned} \text{word}[0] &= B_0B_2B_4B_6, & \text{word}[1] &= B_1B_3B_5B_7, \\ \text{word}[2] &= B_8B_{10}B_{12}B_{14}, & \text{word}[3] &= B_9B_{11}B_{13}B_{15}. \end{aligned}$$

Son code est le suivant :

```
void GARFIELD_32b(uint32_t state[4], uint32_t key[4], uint8_t dec){
 uint32_t RC[4];
 uint8_t round, i;
 for(i=0;i<4;i++){
        state[i] ^= key[i];
 }
 if(dec){
     RC[0]=0x35cf6903;
     RC[1]=0x821cb650;
     RC[2]=0xea841eb8;
     RC[3]=0x9d37d16b;
     Llayer(state,dec);
     Slayer(state);
 }
 else{
     RC[0]=0x30ca64fe;
     RC[1]=0x7d17b14b;
     RC[2]=0xe57f19b3;
     RC[3]=0x9832cc66;
     Slayer(state);
     Llayer(state,dec);
 for(round=1;round<11;round++){
     for(i=0;i<4;i++){
     state[i] ^= key[i];
     state[i] ^= RC[i];
     if(dec){
         RC[i] = UADD8(RC[i],0xe3e3e3e3);
         Llayer(state,dec);
         Slayer(state);
     }
     else{
         RC[i] = UADD8(RC[i],0x1d1d1d1d);
         Slayer(state);
         Llayer(state,dec);
     }
 }
 for(i=0;i<4;i++){
     state[i] ^= key[i];
 }
}
```

GARFIELD_32b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code suivant :

```
uint32_t UADD8(uint32_t word1,uint32_t  word2){
 uint32_t t1, t2;
 t1 = word1&0x7f7f7f7f;
 t2 = word2&0x7f7f7f7f;
 return (t1+t2)^((word1^word2)&0x80808080);
}
```

La couche de substitution utilisée par GARFIELD_32b est donnée par le code suivant :

```
void Slayer(uint32_t X[4]){
    uint8_t a, b, i=2;
    uint32_t Temp[5];
    for(a=0;a<3;a++){
        for(b=0;b<2;b++){
            Temp[2] = (X[i]>>16)&X[i];
            Temp[2] ^=(X[i]>>8);
            Temp[1] = (Temp[2]<<8)&(X[i]<<16);
            Temp[2] ^=X[i];
            if(a&0x1){
                Temp[1] ^=(X[i]>>8);
                Temp[0] = (X[i]<<8)|(X[i]<<16);
                Temp[0] ^=(X[i]<<24);
                Temp[3] = (Temp[0]&(Temp[1]<<8))>>24;
                Temp[3] ^=(X[i]>>8);
            }
            else{
                Temp[0] =X[i]^(X[i]<<24);
                Temp[4] = (Temp[0]>>24)^(X[i]>>8);
                Temp[0] |=(Temp[2]<<16);
                Temp[3] =X[i]^(Temp[2]>>8);
                Temp[3] &=Temp[4];
            }
            Temp[2] ^=(Temp[1]>>8);
            i^=1;
            X[i^2] ^= (Temp[0]&0xff000000);
            X[i^2] ^= (Temp[1]&0xff0000);
            X[i^2] ^= (Temp[2]&0xff00);
            X[i^2] ^= (Temp[3]&0xff);
        }
        i^=2;
    }
}
```

Enfin, son étage linéaire est donné par le code suivant :

```
void Llayer(uint32_t state[4], uint8_t dec){
 uint8_t word=0;
 uint32_t M0,M1;
 while(word<4){
     M0= state[word+dec^1];
     M1= state[word+dec];
     if(dec){
         M1= ((M1<<3)&0x00f800f8)|((M1>>5)&0x00070007)|(M1&0xff00ff00);
         M0= ((M0<<3)&0xf800f800)|((M0>>5)&0x07000700)|(M0&0x00ff00ff);
         M0^=M1;
     }
     M1^=M0;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M0^=M1;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M1^=M0;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M0^=M1;
     if(dec){
         M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     }
```

```
22      else{
23          M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
24      }
25      M1^=M0;
26      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
27      M0^=M1;
28      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
29      M1^=M0;
30      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
31      M0^=M1;
32      if(dec^1){
33          M1^=M0;
34          M0= ((M0<<5)&0x00e000e0)|((M0>>3)&0x001f001f)|(M0&0xff00ff00);
35          M1= ((M1<<5)&0xe000e000)|((M1>>3)&0x1f001f00)|(M1&0x00ff00ff);
36      }
37      state[word+dec^1]=M0;
38      state[word+dec]=M1;
39      word+=2;
40  }
41 }
```

A.3.2 — Implémentation 16 bits appliquée sur 2 blocs

L'implémentation 16 bits de référence de GARFIELD appliquée de façon SIMD sur 2 blocs de données prend en entrée 2 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n+15$ de chaque message), 2 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n+15$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
1  void GARFIELD_16b(uint32_t state[8], uint32_t key[8], uint8_t dec){
2   uint32_t RC;
3   uint8_t round, i;
4   for(i=0;i<8;i++){
5       state[i] ^= key[i];
6   }
7   if(dec){
8       RC = 0x9be89be8;
9       Llayer(state,dec);
10      Slayer(state);
11  }
12  else{
13      RC = 0x96e396e3;
14      Slayer(state);
15      Llayer(state,dec);
16  }
17  for(round=1;round<11;round++){
18      for(i=0;i<8;i++){
19          RC = UADD8(RC,0x9a9a9a9a);
20          state[i] ^= key[i];
21          state[i] ^= RC;
22      }
23      if(dec){
24          Llayer(state,dec);
25          Slayer(state);
26          RC = UADD8(RC,0x13131313);
27      }
28      else{
29          Slayer(state);
30          Llayer(state,dec);
31          RC = UADD8(RC,0x4d4d4d4d);
32      }
33  }
34  for(i=0;i<8;i++){
35      state[i] ^= key[i];
36  }
37 }
```

GARFIELD_16b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code donné en annexe A.3.1.

La couche de substitution utilisée par GARFIELD_16b est donnée par le code suivant :

```
void Slayer(uint32_t X[8]){
    uint8_t a, i=4;
    uint32_t Temp[5];
    for(a=0;a<3;a++){
        Temp[2] = X[i] &X[i+2];
        Temp[2] ^=X[i+1];
        Temp[1] = Temp[2]&X[i+3];
        Temp[2] ^=X[i+2];
        if (a&0x1){
            Temp[1] ^=X[i];
            Temp[0] = X[i+1]|X[i+2];
            Temp[0] ^=X[i+3];
            Temp[3] = Temp[0]&Temp[1];
            Temp[3] ^=X[i+2];
        }
        else{
            Temp[0] = X[i]^X[i+3];
            Temp[4] = Temp[0]^X[i+2];
            Temp[0] = Temp[0]|Temp[2];
            Temp[3] = X[i+3]^Temp[2];
            Temp[3] = Temp[3]&Temp[4];
        }
        Temp[2] ^=Temp[1];
        i^=4;
        X[i]  ^= Temp[0];
        X[i+1]  ^= Temp[1];
        X[i+2]  ^= Temp[2];
        X[i+3]  ^= Temp[3];
    }
}
```

Enfin, son étage linéaire est donné par le code suivant :

```
void Llayer(uint32_t state[8], uint8_t dec){
 uint8_t word=0;
 uint32_t M0,M1,T;
 while(word<8){
     M0= state[word]&0xff00ff00;
     M1= (state[word]&0x00ff00ff)<<8;
     if(dec^1^(word>3)){
         T=M0;
         M0=M1;
         M1=T;
     }
         if(dec){
             if(word&0x1){
                 M1= ((M1<<3)&0xf8f8f8f8)|((M1>>5)&0x07070707);
             }
             else{
                 M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
             }
             M0^=M1;
         }
     M1^=M0;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M0^=M1;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M1^=M0;
     M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
     M0^=M1;
```

```
28      if (dec){
29              M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
30      }
31      else{
32              M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
33      }
34      M1^=M0;
35      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
36      M0^=M1;
37      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
38      M1^=M0;
39      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
40      M0^=M1;
41      if (dec^1){
42              M1^=M0;
43              if (word&0x1){
44                  M0= ((M0<<5)&0xe0e0e0e0)|((M0>>3)&0x1f1f1f1f);
45              }
46              else{
47                  M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
48              }
49          }
50      if (dec^1^(word>3)){
51          T=M0;
52          M0=M1;
53          M1=T;
54      }
55      state[word] = M0|(M1>>8);
56      word+=1;
57  }
58 }
```

Cette implémentation permet d'appliquer GARFIELD sur 2 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin d'exploiter une architecture 16 bits.

Afin d'appliquer le masquage sur GARFIELD, il est possible d'utiliser le schéma décrit en section 2.6. Il suffit pour cela que chaque mot de 32 bits en entrée soit composé d'un mot de 16 bits du masque et d'un mot de 16 bits du bloc de données masquées comme illustré à la figure 2.11, de ne pas effectuer le XOR entre le masque et la clé ou les constantes de tour, et que chaque opération & et | soit modifiée afin de calculer correctement la valeur des masques.

A.3.3 — Implémentation 8 bits appliquée sur 4 blocs

L'implémentation 8 bits de référence de GARFIELD appliquée de façon SIMD sur 4 blocs de données prend en entrée 4 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 15$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque message), 4 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 15$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou

égal à 1 sinon. Son code est le suivant :

```
void GARFIELD_8b(uint32_t state[16], uint32_t key[16], uint8_t dec){
 uint32_t RC;
 uint8_t round, i;
 for(i=0;i<16;i++){
     state[i] ^= key[i];
 }
 if(dec){
     RC = 0xe8e8e8e8;
     Llayer(state,dec);
     Slayer(state);
 }
 else{
     RC = 0xe3e3e3e3;
     Slayer(state);
     Llayer(state,dec);
 }
 for(round=1;round<11;round++){
     for(i=0;i<16;i++){
         RC = UADD8(RC,0x4d4d4d4d);
         state[i] ^= key[i];
         state[i] ^= RC;
     }
     if(dec){
         RC = UADD8(RC,0x13131313);
         Llayer(state,dec);
         Slayer(state);
     }
     else{
         RC = UADD8(RC,0x4d4d4d4d);
         Slayer(state);
         Llayer(state,dec);
     }
 }
 for(i=0;i<16;i++){
     state[i] ^= key[i];
 }
}
```

GARFIELD_8b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code donné en annexe A.3.1.

La couche de substitution utilisée par GARFIELD_8b est donnée par le code suivant :

```
void Slayer(uint32_t X[16]){
    uint8_t a, b, i=8;
    uint32_t Temp[5];
    for(a=0;a<3;a++){
        for(b=0;b<2;b++){
            Temp[2] = X[i]&X[i+4];
            Temp[2] ^=X[i+2];
            Temp[1] = Temp[2]&X[i+6];
            Temp[2] ^=X[i+4];
            if(a&0x1){
                Temp[1] ^=X[i];
                Temp[0] = X[i+2]|X[i+4];
                Temp[0] ^=X[i+6];
                Temp[3] = Temp[0]&Temp[1];
                Temp[3] ^=X[i+4];
            }
            else{
                Temp[0] = X[i]^X[i+6];
                Temp[4] = Temp[0]^X[i+4];
                Temp[0] |=Temp[2];
                Temp[3] = X[i+6]^Temp[2];
                Temp[3] &=Temp[4];
            }
            Temp[2] ^=Temp[1];
            X[i^8] ^= Temp[0];
            X[i^8+2] ^=Temp[1];
            X[i^8+4] ^=Temp[2];
            X[i^8+6] ^=Temp[3];
            i^=1;
        }
        i^=8;
    }
}
```

Enfin, son étage linéaire est donné par le code suivant :

```
void Llayer(uint32_t state[16], uint8_t dec){
 uint8_t word=0;
 uint32_t M0,M1,T;
 while(word<16){
      M0= state[word+dec^1];
      M1= state[word+dec];
      if(round>7){
          T=M0;
          M0=M1;
          M1=T;
      }
         if(dec){
             if(word&0x2){
                 M1= ((M1<<3)&0xf8f8f8f8)|((M1>>5)&0x07070707);
             }
             else{
                 M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
             }
             M0^=M1;
         }
      M1^=M0;
      M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
      M0^=M1;
      M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
      M1^=M0;
      M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
      M0^=M1;
      if(dec){
             M0= ((M0<<3)&0xf8f8f8f8)|((M0>>5)&0x07070707);
      }
      else{
             M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
      }
      M1^=M0;
      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
      M0^=M1;
      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
      M1^=M0;
      M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
      M0^=M1;
      if(dec^1){
             M1^=M0;
             if(word&0x2){
                 M0= ((M0<<5)&0xe0e0e0e0)|((M0>>3)&0x1f1f1f1f);
             }
             else{
                 M1= ((M1<<5)&0xe0e0e0e0)|((M1>>3)&0x1f1f1f1f);
             }
         }
      if(word>7){
          T=M0;
          M0=M1;
          M1=T;
      }
      state[word+dec^1] =M0;
      state[word+dec] =M1;
      word+=2;
 }
}
```

Cette implémentation permet d'appliquer GARFIELD sur 4 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin de l'appliquer sur 2 blocs en entrée et d'exploiter une architecture 16 bits, ou de remplacer les mots de 32 bits par des mots de 8 bits, et d'effectuer les opérations seulement sur 8 bits, afin d'exploiter une architecture 8 bits.

Pour appliquer l'IRC qui est décrit dans le chapitre 4, il suffit que chaque mot de 32 bits soit composé de deux copies d'un octet du bloc de données et d'un octet de chacun des 2 blocs de référence comme illustré à la figure 4.3. Enfin, pour combiner le masquage et l'IRC, il suffit de remplacer le premier bloc de référence par un octet du masque comme illustré à la figure 4.10, de ne pas effectuer le XOR entre le masque et la clé ou les constantes de tour, et que chaque opération & et | soit modifiée afin de calculer correctement la valeur des masques.

A.4 Implémentations de PRIDE

Les spécifications de PRIDE sont données en section 2.3.2.1. Le message clair, le chiffré ainsi que la clé sont stockés de gauche à droite dans les registres. Il est à noter que les 8 premiers octets de la clé de chiffrement doivent contenir $\mathcal{P}(K_0)$ dans les implémentations que nous avons utilisées.

A.4.1 Implémentation 32 bits

Nous avons proposé une implémentation 32 bits de PRIDE à partir de son code de référence afin d'exploiter pleinement une architecture 32 bits. Elle prend en entrée un message clair ou un chiffré stocké dans les registres "state", la clé secrète stockée dans les registres "key" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
1  void PRIDE_32b(uint32_t state[2], uint32_t key[4], uint8_t dec){
2      uint32_t round, Rkey[2];
3      Rkey[0] = key[2];
4      Rkey[1] = key[3];
5      // XOR avec P(k_0)
6      state[0] ^= key[0];
7      state[1] ^= key[1];
8      if(dec){
9          Rkey[0] = UADD8(Rkey[0],0x001400e4);
10         Rkey[1] = UADD8(Rkey[1],0x00540064);
11     }
```

```
    for(round=1;round<20;round++){
        if(dec){
            Slayer(state);
        }
        else{
            Rkey[0] = UADD8(Rkey[0],0x00c100a5);
            Rkey[1] = UADD8(Rkey[1],0x005100c5);
        }
        state[0] ^= Rkey[0];
        state[1] ^= Rkey[1];
        if(dec^1){
            Slayer(state);
        }
        else{
            Rkey[0] = UADD8(Rkey[0],0x003f005b);
            Rkey[1] = UADD8(Rkey[1],0x00af003b);
        }
        Llayer(state,dec);
    }
    if(dec){
        Slayer(state);
    }
    else{
        Rkey[0] = UADD8(Rkey[0],0x00c100a5);
        Rkey[1] = UADD8(Rkey[1],0x005100c5);
    }
    state[0] ^= Rkey[0];
    state[1] ^= Rkey[1];
    if(dec^1){
        Slayer(state);
    }
    // XOR avec P(k0)
    state[0] ^= key[0];
    state[1] ^= key[1];
}
```

PRIDE_32b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code donné en annexe A.3.1.

La couche de substitution utilisée par PRIDE_32b est donnée par le code suivant :

```
void Slayer(uint32_t X[2]){
    uint32_t A, B, C, D;
    A = (X[0]&0xffff0000&(X[0]<<16))^(X[1]&0xffff0000);
    B = (X[0]&0x0000ffff&(X[1]>>16))^(X[1]&0x0000ffff);
    C = (A&(B<<16))^(X[0]&0xffff0000);
    D = (B&(C>>16))^(X[0]&0x0000ffff);
    X[0] = A|B;
    X[1] = C|D;
}
```

Enfin, son étage linéaire est donné par le code suivant :

```
void Llayer(uint32_t Z[2],uint8_t dec){
    uint32_t Temp0[4],Temp1[2],Bit0,Bit1;
    if(dec){
        Temp0[2]= Z[0]&0x0000ff00;
        Temp0[3]= Z[0]&0x000000ff;
        Bit0 = Z[0]&(1<<8);
        Bit1 = Z[0]&1;
```

```
        Temp0[2]= ((Temp0[2]>>1)&0x0000ff00)^(Bit0<<7);
        Temp0[3]= ((Temp0[3]>>1)&0x000000ff)^(Bit1 <<7);
        Temp0[3]^= (Temp0[2]>>8);
        Z[0]  ^= Temp0[3];
    }
    Temp0[0]= Z[0]&0xff000000;
    Temp0[1]= Z[0]&0x00ff0000;
    Temp1[0]= (Z[0]<<4)&0xf0f000f0;
    Temp1[1]= (Z[0]>>4)&0x0f0f000f;
    Z[0]  = (Temp1[0]^Temp1[1])^(Z[0]&0x0000ff00);
    Temp1[0]= (Z[0]<<8)&0xff000000;
    Z[0]  ^= Temp1[0];
    Temp0[0]^= (Z[0]&0xff000000);
    Z[0]  = (Z[0]&0xff00ffff)^(Temp0[0]>>8);
    Z[0]  ^= (Temp0[1]<<8);
    if (dec^1){
        Bit0  = Z[0]&(1<<15);
        Bit1  = Z[0]&1;
        Temp0[2]= Z[0]&0x0000ff00;
        Temp0[3]= Z[0]&0x000000ff;
        Temp0[2]= ((Temp0[2]<<1)&0x0000ff00)^(Bit0>>7);
        Temp0[3]= ((Temp0[3]>>1)&0x000000ff)^(Bit1<<7);
        Z[0]  ^= (Temp0[3]<<8);
        Temp0[3]= Z[0]&0x0000ff00;
        Z[0]  ^= Temp0[2];
        Z[0]  ^= (Temp0[3]>>8);
    }
    else{
        Z[0]  = (Z[0]&0xffff00ff)^(Temp0[3]<<8);
        Bit0  = Z[0]&(1<<8);
        Z[0]  = (Z[0]&0xffff00ff)^(((Z[0]&0x0000ff00)>>1)&0x0000ff00)^(Bit0<<7);
        Z[0]  ^= Temp0[2];
    }
    if (dec){
        Temp0[0]= Z[1]&0xff000000;
        Temp0[1]= Z[1]&0x00ff0000;
        Bit0  = Z[1]&(1<<24);
        Bit1  = Z[1]&(1<<16);
        Temp0[0]= ((Temp0[0]>>1)&0xff000000)^(Bit0<<7);
        Temp0[1]= ((Temp0[1]>>1)&0x00ff0000)^(Bit1<<7);
        Temp0[1]^= (Temp0[0]>>8);
        Z[1]  ^= Temp0[1];
        Z[1]  = (Z[1]&0x00ffffff)^(Temp0[1]<<8);
        Bit0  = Z[1]&(1<<24);
        Z[1]  = (Z[1]&0x00ffffff)^(((Z[1]>>1)&0xff000000)^(Bit0<<7));
        Z[1]  ^= Temp0[0];
    }
    Temp0[2]= Z[1]&0x0000ff00;
    Temp0[3]= Z[1]&0x000000ff;
    Temp1[0]= (Z[1]<<4)&0x0f000f0f;
    Temp1[1]= (Z[1]>>4)&0x0f000f0f;
    Z[1]  = (Temp1[0]^Temp1[1])^(Z[1]&0x00ff0000);
    if (dec^1){
        Bit0  = Z[1]&(1<<31);
        Bit1  = Z[1]&(1<<16);
        Temp0[0]= Z[1]&0xff000000;
        Temp0[1]= Z[1]&0x00ff0000;
        Temp0[0]= ((Temp0[0]<<1)&0xff000000)^(Bit0>>7);
        Temp0[1]= ((Temp0[1]>>1)&0x00ff0000)^(Bit1<<7);
        Z[1]  ^= (Temp0[1]<<8);
        Temp0[1]= Z[1]&0xff000000;
        Z[1]  ^= Temp0[0];
        Z[1]  ^= (Temp0[1]>>8);
    }
```

```
    Z[1] ^= ((Z[1]&0x000000ff)<<8);
    Temp0[3]^= ((Z[1]&0x0000ff00)>>8);
    Z[1] = (Z[1]&0xffffff00)^Temp0[3];
    Z[1] ^= Temp0[2];
}
```

A.4.2 — Implémentation 16 bits appliquée sur 2 blocs

L'implémentation 16 bits de PRIDE appliquée de façon SIMD sur 2 blocs de données que nous avons proposée prend en entrée 2 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 3$ correspond à la concaténation des bits $16n$ à $16n + 15$ de chaque message), 2 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n + 15$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void PRIDE_16b(uint32_t state[4], uint32_t key[8], uint8_t dec){
    uint8_t round,i;
    uint32_t Rkey[4];
    for(i=0;i<4;i++){
        Rkey[i] = key[i+4];
        // XOR avec P(k0)
        state[i] ^= key[i];
    }
    if(dec){
        Rkey[0] = UADD8(Rkey[0],0x00140014);
        Rkey[1] = UADD8(Rkey[1],0x00e400e4);
        Rkey[2] = UADD8(Rkey[2],0x00540054);
        Rkey[3] = UADD8(Rkey[3],0x00640064);
    }
    for(round=1;round<20;round++){
        if(dec){
            Slayer(state);
        }
        else{
            Rkey[0] = UADD8(Rkey[0],0x00c100c1);
            Rkey[1] = UADD8(Rkey[1],0x00a500a5);
            Rkey[2] = UADD8(Rkey[2],0x00510051);
            Rkey[3] = UADD8(Rkey[3],0x00c500c5);
        }
        for(i=0;i<4;i++){
            state[i] ^= Rkey[i];
        }
        if(dec^1){
            Slayer(state);
        }
        else{
            Rkey[0] = UADD8(Rkey[0],0x003f003f);
            Rkey[1] = UADD8(Rkey[1],0x005b005b);
            Rkey[2] = UADD8(Rkey[2],0x00af00af);
            Rkey[3] = UADD8(Rkey[3],0x003b003b);
        }
        Llayer(state,dec);
    }
```

```
    if (dec){
        Slayer(state);
    }
    else{
        Rkey[0] = UADD8(Rkey[0],0x00c100c1);
        Rkey[1] = UADD8(Rkey[1],0x00a500a5);
        Rkey[2] = UADD8(Rkey[2],0x00510051);
        Rkey[3] = UADD8(Rkey[3],0x00c500c5);
    }
    for(i=0;i<4;i++){
        state[i] ^= Rkey[i];
    }
    if (dec^1){
        Slayer(state);
    }
    for(i=0;i<4;i++){
        // XOR avec P(k0)
        state[i] ^= key[i];
    }
}
```

PRIDE_16b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code donné en annexe A.3.1.

La couche de substitution utilisée par PRIDE_16b est donnée par le code suivant :

```
void Slayer(uint32_t X[4]){
    uint32_t Temp0,Temp1;
    Temp0=X[0];
    Temp1=X[1];
    X[0] = X[0]&X[1];
    X[0] ^= X[2];
    X[1] = X[1]&X[2];
    X[1] ^= X[3];
    X[2] = X[0];
    X[3] = X[1];
    X[2] = X[2]&X[3];
    X[2] ^= Temp0;
    X[3] = X[2]&X[3];
    X[3] ^= Temp1;
}
```

Enfin, son étage linéaire est donné par le code suivant :

```
void Llayer(uint32_t Z[4], uint8_t dec){
    uint32_t Temp0,Temp1,Temp2;
    Temp0=Z[0];
    Temp2=(Z[0]<<4)&0xf0f0f0f0;
    Temp2|=(Z[0]>>4)&0x0f0f0f0f;
    Z[0] = Temp2;
    Z[0] ^= (Z[0]<<8)&0xff00ff00;
    Temp0^=(Z[0]&0xff00ff00);
    Z[0] &=0xff00ff00;
    Z[0] |= (Temp0>>8)&0x00ff00ff;
    Z[0] ^= (Temp0<<8)&0xff00ff00;
    if (dec){
        Temp0=((Z[1]<<7)&0x80808080)|((Z[1]>>1)&0x7f7f7f7f);
        Temp0^=(Temp0>>8)&0x00ff00ff;
```

```
        Z[1] ^= (Temp0&0x00ff00ff);
        Temp1= (Z[1]<<4)&0x00f000f0;
        Temp1|=(Z[1]>>4)&0x000f000f;
        Z[1] = Temp1|((Temp0<<8)&0xff00ff00);
        Z[1] = (Z[1]&0x00ff00ff)|((Z[1]<<7)&0x80008000)|((Z[1]>>1)&0x7f007f00);
        Z[1] ^= (Temp0&0xff00ff00);
        Temp0= ((Z[2]<<7)&0x80808080)|((Z[2]>>1)&0x7f7f7f7f);
        Temp0^=(Temp0>>8)&0x00ff00ff;
        Z[2] ^= (Temp0&0x00ff00ff);
        Z[2] = (Z[2]&0x00ff00ff)|((Temp0<<8)&0xff00ff00);
        Z[2] = (Z[2]&0x00ff00ff)|((Z[2]<<7)&0x80008000)|((Z[2]>>1)&0x7f007f00);
        Z[2] ^= (Temp0&0xff00ff00);
        Temp0= (Z[2]<<4)&0xf000f000;
        Temp0|=(Z[2]>>4)&0x0f000f00;
        Z[2] = Temp0|(Z[2]&0x00ff00ff);
    }
    else{
        Temp0= (Z[1]<<4)&0x00f000f0;
        Temp0|=(Z[1]>>4)&0x000f000f;
        Z[1] = Temp0|(Z[1]&0xff00ff00);
        Temp0= ((Z[1]<<1)&0xfe00fe00)|((Z[1]>>7)&0x01000100);
        Temp1= ((Z[1]<<7)&0x00800080)|((Z[1]>>1)&0x007f007f);
        Z[1] ^= (Temp1<<8);
        Temp1=Z[1]&0xff00ff00;
        Z[1] ^= Temp0;
        Z[1] ^= (Temp1>>8);
        Temp0= (Z[2]<<4)&0xf000f000;
        Temp0|=(Z[2]>>4)&0x0f000f00;
        Z[2] = Temp0|(Z[2]&0x00ff00ff);
        Temp0= ((Z[2]<<1)&0xfe00fe00)|((Z[2]>>7)&0x01000100);
        Temp1= ((Z[2]<<7)&0x00800080)|((Z[2]>>1)&0x007f007f);
        Z[2] ^= (Temp1<<8);
        Temp1=Z[2]&0xff00ff00;
        Z[2] ^= Temp0;
        Z[2] ^= (Temp1>>8);
    }
    Temp0=Z[3];
    Temp2= (Z[3]<<4)&0xf0f0f0f0;
    Temp2|=(Z[3]>>4)&0x0f0f0f0f;
    Z[3] = Temp2;
    Z[3] ^= (Z[3]<<8)&0xff00ff00;
    Temp0^=(Z[3]>>8)&0x00ff00ff;
    Z[3] &=0xff00ff00;
    Z[3] |= (Temp0&0x00ff00ff);
    Z[3] ^= (Temp0&0xff00ff00);
}
```

Cette implémentation permet d'appliquer PRIDE sur 2 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin d'exploiter une architecture 16 bits.

Afin d'appliquer le masquage sur PRIDE, il est possible d'utiliser le schéma décrit en section 2.6. Il suffit pour cela que chaque mot de 32 bits en entrée soit composé d'un mot de 16 bits du masque et d'un mot de 16 bits du bloc de données masquées comme illustré à la figure 2.11, de ne pas effectuer le XOR entre le masque et la clé de tour, et que chaque opération & soit modifiée afin de calculer correctement la valeur des masques.

A.4.3 — Implémentation 8 bits appliquée sur 4 blocs

L'implémentation 8 bits de PRIDE appliquée de façon SIMD sur 4 blocs de données que nous avons proposée prend en entrée 4 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque message), 4 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 15$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void PRIDE_8b(uint32_t state[8], uint32_t key[16], uint8_t dec){
    uint8_t round,i;
    uint32_t Rkey[8];
    for(i=0;i<8;i++){
        Rkey[i] = key[i+8];
        // XOR avec P(k_0)
        state[i] ^= key[i];
    }
    if(dec){
        Rkey[1] = UADD8(Rkey[1],0x14141414);
        Rkey[3] = UADD8(Rkey[3],0xe4e4e4e4);
        Rkey[5] = UADD8(Rkey[5],0x54545454);
        Rkey[7] = UADD8(Rkey[7],0x64646464);
    }
    for(round=1;round<20;round++){
        if(dec){
            Slayer(state);
        }
        else{
            Rkey[1] = UADD8(Rkey[1],0xc1c1c1c1);
            Rkey[3] = UADD8(Rkey[3],0xa5a5a5a5);
            Rkey[5] = UADD8(Rkey[5],0x51515151);
            Rkey[7] = UADD8(Rkey[7],0xc5c5c5c5);
        }
        for(i=0;i<8;i++){
            state[i] ^= Rkey[i];
        }
        if(dec^1){
            Slayer(state);
        }
        else{
            Rkey[1] = UADD8(Rkey[1],0x3f3f3f3f);
            Rkey[3] = UADD8(Rkey[3],0x5b5b5b5b);
            Rkey[5] = UADD8(Rkey[5],0xafafafaf);
            Rkey[7] = UADD8(Rkey[7],0x3b3b3b3b);
        }
        Llayer(state,dec);
    }
    if(dec){
        Slayer(state);
    }
    else{
        Rkey[1] = UADD8(Rkey[1],0xc1c1c1c1);
        Rkey[3] = UADD8(Rkey[3],0xa5a5a5a5);
        Rkey[5] = UADD8(Rkey[5],0x51515151);
        Rkey[7] = UADD8(Rkey[7],0xc5c5c5c5);
    }
```

```
48      for(i=0;i<8;i++){
49          state[i] ^= Rkey[i];
50      }
51      if(dec^1){
52          Slayer(state);
53      }
54      for(i=0;i<8;i++){
55       // XOR avec $\mathcal{P}(k_0)$
56          state[i] ^= key[i];
57      }
58  }
```

PRIDE_8b utilise l'opération d'addition sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM le permettant, ou appliquée sinon par le code donné en annexe A.3.1.

La couche de substitution utilisée par PRIDE_8b est donnée par le code suivant :

```
1  void Slayer(uint32_t X[8]){
2      uint8_t i;
3      uint32_t Temp0,Temp1;
4      for(i=0;i<2;i++){
5          Temp0=X[i];
6          Temp1=X[i+2];
7          X[i] &=X[i+2];
8          X[i] ^= X[i+4];
9          X[i+2] &=X[i+4];
10         X[i+2] ^= X[i+6];
11         X[i+4] = X[i];
12         X[i+6] = X[i+2];
13         X[i+4] &=X[i+6];
14         X[i+4] ^= Temp0;
15         X[i+6] &=X[i+4];
16         X[i+6] ^= Temp1;
17     }
18 }
```

Enfin, son étage linéaire est donné par le code suivant :

```
1  void Llayer(uint32_t Z[8], uint8_t dec){
2    uint32_t Temp0,Temp1,Temp2;
3      Temp0=Z[0];
4      Temp1=Z[1];
5      Temp2=(Z[0]<<4)&0xf0f0f0f0;
6      Temp2|=(Z[0]>>4)&0x0f0f0f0f;
7      Z[0] = Temp2;
8      Temp2=(Z[1]<<4)&0xf0f0f0f0;
9      Temp2|=(Z[1]>>4)&0x0f0f0f0f;
10     Z[1] = Temp2;
11     Z[0] ^= Z[1];
12     Temp0^=Z[0];
13     Z[1] = Temp0;
14     Z[0] ^= Temp1;
15     if(dec){
16         Temp0=((Z[2]<<7)&0x80808080)|((Z[2]>>1)&0x7f7f7f7f);
17         Temp1=((Z[3]<<7)&0x80808080)|((Z[3]>>1)&0x7f7f7f7f);
18         Temp1^=Temp0;
19         Z[3] ^= Temp1;
```

```
        Temp2=(Z[3]<<4)&0xf0f0f0f0;
        Temp2|=(Z[3]>>4)&0x0f0f0f0f;
        Z[3] = Temp2;
        Z[2] = Temp1;
        Z[2] = ((Z[2]<<7)&0x80808080)|((Z[2]>>1)&0x7f7f7f7f);
        Z[2] ^= Temp0;
        Temp0=((Z[4]<<7)&0x80808080)|((Z[4]>>1)&0x7f7f7f7f);
        Temp1=((Z[5]<<7)&0x80808080)|((Z[5]>>1)&0x7f7f7f7f);
        Temp1^=Temp0;
        Z[5] ^= Temp1;
        Z[4] = Temp1;
        Z[4] = ((Z[4]<<7)&0x80808080)|((Z[4]>>1)&0x7f7f7f7f);
        Z[4] ^= Temp0;
        Temp2=(Z[4]<<4)&0xf0f0f0f0;
        Temp2|=(Z[4]>>4)&0x0f0f0f0f;
        Z[4] = Temp2;
    }
    else{
        Temp0=(Z[3]<<4)&0xf0f0f0f0;
        Temp0|=(Z[3]>>4)&0x0f0f0f0f;
        Z[3] = Temp0;
        Temp2=((Z[2]&0xff00ff00)<<1)&0xff00ff00;
        Temp2|=((Z[2]&0xff00ff00)>>7)&0xff00ff00;
        Temp0=((Z[2]&0x00ff00ff)<<1)&0x00ff00ff;
        Temp0|=((Z[2]&0x00ff00ff)>>7)&0x00ff00ff;
        Temp0|=Temp2;
        Temp2=((Z[3]&0xff00ff00)>>1)&0xff00ff00;
        Temp2|=((Z[3]&0xff00ff00)<<7)&0xff00ff00;
        Temp1=((Z[3]&0x00ff00ff)>>1)&0x00ff00ff;
        Temp1|=((Z[3]&0x00ff00ff)<<7)&0x00ff00ff;
        Temp1|=Temp2;
        Z[2] ^= Temp1;
        Temp1=Z[2];
        Z[2] ^= Temp0;
        Z[3] ^= Temp1;
        Temp0=(Z[4]<<4)&0xf0f0f0f0;
        Temp0|=(Z[4]>>4)&0x0f0f0f0f;
        Z[4] = Temp0;
        Temp2=((Z[4]&0xff00ff00)<<1)&0xff00ff00;
        Temp2|=((Z[4]&0xff00ff00)>>7)&0xff00ff00;
        Temp0=((Z[4]&0x00ff00ff)<<1)&0x00ff00ff;
        Temp0|=((Z[4]&0x00ff00ff)>>7)&0x00ff00ff;
        Temp0|=Temp2;
        Temp2=((Z[5]&0xff00ff00)>>1)&0xff00ff00;
        Temp2|=((Z[5]&0xff00ff00)<<7)&0xff00ff00;
        Temp1=((Z[5]&0x00ff00ff)>>1)&0x00ff00ff;
        Temp1|=((Z[5]&0x00ff00ff)<<7)&0x00ff00ff;
        Temp1|=Temp2;
        Z[4] ^= Temp1;
        Temp1=Z[4];
        Z[4] ^= Temp0;
        Z[5] ^= Temp1;
    }
    Temp0=Z[6];
    Temp1=Z[7];
```

```
    Temp2=(Z[6]<<4)&0xf0f0f0f0;
    Temp2|=(Z[6]>>4)&0x0f0f0f0f;
    Z[6] = Temp2;
    Temp2=(Z[7]<<4)&0xf0f0f0f0;
    Temp2|=(Z[7]>>4)&0x0f0f0f0f;
    Z[7] = Temp2;
    Z[6] ^= Z[7];
    Temp1^=Z[6];
    Z[7] = Temp1;
    Z[6] ^= Temp0;
}
```

Cette implémentation permet d'appliquer PRIDE sur 4 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin de l'appliquer sur 2 blocs en entrée et d'exploiter une architecture 16 bits, ou de remplacer les mots de 32 bits par des mots de 8 bits, et d'effectuer les opérations seulement sur 8 bits, afin d'exploiter une architecture 8 bits.

Pour appliquer l'IRC qui est décrit dans le chapitre 4, il suffit que chaque mot de 32 bits soit composé de deux copies d'un octet du bloc de données et d'un octet de chacun des 2 blocs de référence comme illustré à la figure 4.3. Enfin, pour combiner le masquage et l'IRC, il suffit de remplacer le premier bloc de référence par un octet du masque comme illustré sur la figure 4.10, de ne pas effectuer le XOR entre le masque et la clé ou les constantes de tour, et que chaque opération & et | soit modifiée afin de calculer correctement la valeur des masques.

A.5 Implémentations de Robin

Les spécifications de Robin sont données dans [Grosso et al. 2015]. Le message clair, le chiffré ainsi que les clés sont stockés de gauche à droite dans les registres.

A.5.1 Implémentation 32 bits

Nous avons proposé une implémentation 32 bits de Robin à partir de son code de référence afin d'exploiter pleinement une architecture 32 bits. Elle prend en entrée un message clair ou un chiffré stocké dans les registres "state", la clé secrète stockée dans les registres "key" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon.

Son code est le suivant :

```
void Robin_32b(uint32_t state[4], uint32_t key[4], uint8_t dec){
    uint8_t round,i;
    uint32_t temp;
    for(i=0;i<4;i++){
        state[i] ^= key[i];
    }
    for(round=0;round<16;round++){
        if(dec^1){
            state[0] ^= (LBox1[round+1]<<16);
            Slayer(state);
        }
        for(i=0;i<4;i++){
            temp = (LBox2[(state[i]>>24)&0xff]^LBox1[(state[i]>>16)&0xff])<<16;
            state[i] = temp|(LBox2[(state[i]>>8)&0xff]^LBox1[state[i]&0xff]);
        }
        if(dec){
            Slayer(state);
            state[0] ^= (LBox1[0][16-round])<<16;
        }
        for(i=0;i<4;i++){
            state[i] ^= key[i];
        }
    }
}
```

La couche de substitution utilisée par Robin_32b est donnée par le code suivant :

```
void Slayer(uint32_t X[4]){
    uint8_t a, i=2;
    uint32_t Temp[2];
    for(a=0;a<3;a++){
        Temp[0] = X[i]&(X[i]<<16);
        Temp[0] ^=X[i+1];
        Temp[1] = (X[i]<<16)|X[i+1];
        Temp[1] ^=(X[i+1]<<16);
        Temp[1] &=0xffff0000;
        Temp[1] |=(Temp[0]>>16)&X[i+1];
        Temp[1] ^=(X[i]>>16);
        Temp[0] &=0xffff0000;
        Temp[0] |=(Temp[1]>>16)&(X[i]>>16);
        Temp[0] ^=(X[i]&0x0000ffff);
        i^=2;
        X[i] ^= Temp[0];
        X[i+1] ^= Temp[1];
    }
}
```

Enfin, les tableaux utilisés par l'étage linéaire de Robin_32b sont les suivants :

```
uint16_t LBox1[256]= {
 0x0000,0xfffe,0xccc1,0x333f,0xaaa1,0x555f,0x6660,0x999e,0x9991,0x666f,0x5550,0xaaae,
 0x3330,0xccce,0xfff1,0x000f,0x6689,0x9977,0xaa48,0x55b6,0xcc28,0x33d6,0x00e9,0xff17,
 0xff18,0x00e6,0x33d9,0xcc27,0x55b9,0xaa47,0x9978,0x6686,0x5585,0xaa7b,0x9944,0x66ba,
 0xff24,0x00da,0x33e5,0xcc1b,0xcc14,0x33ea,0x00d5,0xff2b,0x66b5,0x994b,0xaa74,0x558a,
 0x330c,0xccf2,0xffcd,0x0033,0x99ad,0x6653,0x556c,0xaa92,0xaa9d,0x5563,0x665c,0x99a2,
 0x003c,0xffc2,0xccfd,0x3303,0x3383,0xcc7d,0xff42,0x00bc,0x9922,0x66dc,0x55e3,0xaa1d,
 0xaa12,0x55ec,0x66d3,0x992d,0x00b3,0xff4d,0xcc72,0x338c,0x550a,0xaaf4,0x99cb,0x6635,
 0xffab,0x0055,0x336a,0xcc94,0xcc9b,0x3365,0x005a,0xffa4,0x663a,0x99c4,0xaafb,0x5505,
 0x6606,0x99f8,0xaac7,0x5539,0xcca7,0x3359,0x0066,0xff98,0xff97,0x0069,0x3356,0xcca8,
```

```
0x5536,0xaac8,0x99f7,0x6609,0x008f,0xff71,0xcc4e,0x33b0,0xaa2e,0x55d0,0x66ef,0x9911,
0x991e,0x66e0,0x55df,0xaa21,0x33bf,0xcc41,0xff7e,0x0080,0x007f,0xff81,0xccbe,0x3340,
0xaade,0x5520,0x661f,0x99e1,0x99ee,0x6610,0x552f,0xaad1,0x334f,0xccb1,0xff8e,0x0070,
0x66f6,0x9908,0xaa37,0x55c9,0xcc57,0x33a9,0x0096,0xff68,0xff67,0x0099,0x33a6,0xcc58,
0x55c6,0xaa38,0x9907,0x66f9,0x55fa,0xaa04,0x993b,0x66c5,0xff5b,0x00a5,0x339a,0xcc64,
0xcc6b,0x3395,0x00aa,0xff54,0x66ca,0x9934,0xaa0b,0x55f5,0x3373,0xcc8d,0xffb2,0x004c,
0x99d2,0x662c,0x5513,0xaaed,0xaae2,0x551c,0x6623,0x99dd,0x0043,0xffbd,0xcc82,0x337c,
0x33fc,0xcc02,0xff3d,0x00c3,0x995d,0x66a3,0x559c,0xaa62,0xaa6d,0x5593,0x66ac,0x9952,
0x00cc,0xff32,0xcc0d,0x33f3,0x5575,0xaa8b,0x99b4,0x664a,0xffd4,0x002a,0x3315,0xcceb,
0xcce4,0x331a,0x0025,0xffdb,0x6645,0x99bb,0xaa84,0x557a,0x6679,0x9987,0xaab8,0x5546,
0xccd8,0x3326,0x0019,0xffe7,0xffe8,0x0016,0x3329,0xccd7,0x5549,0xaab7,0x9988,0x6676,
0x00f0,0xff0e,0xcc31,0x33cf,0xaa51,0x55af,0x6690,0x996e,0x9961,0x669f,0x55a0,0xaa5e,
0x33c0,0xcc3e,0xff01,0x00ff
};
uint16_t LBox2[256]= {
0x0000,0xe069,0xd055,0x303c,0xb033,0x505a,0x6066,0x800f,0x700f,0x9066,0xa05a,0x4033,
0xc03c,0x2055,0x1069,0xf000,0x0e69,0xee00,0xde3c,0x3e55,0xbe5a,0x5e33,0x6e0f,0x8e66,
0x7e66,0x9e0f,0xae33,0x4e5a,0xce55,0x2e3c,0x1e00,0xfe69,0x0d55,0xed3c,0xdd00,0x3d69,
0xbd66,0x5d0f,0x6d33,0x8d5a,0x7d5a,0x9d33,0xad0f,0x4d66,0xcd69,0x2d00,0x1d3c,0xfd55,
0x033c,0xe355,0xd369,0x3300,0xb30f,0x5366,0x635a,0x8333,0x7333,0x935a,0xa366,0x430f,
0xc300,0x2369,0x1355,0xf33c,0x0b33,0xeb5a,0xdb66,0x3b0f,0xbb00,0x5b69,0x6b55,0x8b3c,
0x7b3c,0x9b55,0xab69,0x4b00,0xcb0f,0x2b66,0x1b5a,0xfb33,0x055a,0xe533,0xd50f,0x3566,
0xb569,0x5500,0x653c,0x8555,0x7555,0x953c,0xa500,0x4569,0xc566,0x250f,0x1533,0xf55a,
0x0666,0xe60f,0xd633,0x365a,0xb655,0x563c,0x6600,0x8669,0x7669,0x9600,0xa63c,0x4655,
0xc65a,0x2633,0x160f,0xf666,0x080f,0xe866,0xd85a,0x3833,0xb83c,0x5855,0x6869,0x8800,
0x7800,0x9869,0xa855,0x483c,0xc833,0x285a,0x1866,0xf80f,0x070f,0xe766,0xd75a,0x3733,
0xb73c,0x5755,0x6769,0x8700,0x7700,0x9769,0xa755,0x473c,0xc733,0x275a,0x1766,0xf70f,
0x0966,0xe90f,0xd933,0x395a,0xb955,0x593c,0x6900,0x8969,0x7969,0x9900,0xa93c,0x4955,
0xc95a,0x2933,0x190f,0xf966,0x0a5a,0xea33,0xda0f,0x3a66,0xba69,0x5a00,0x6a3c,0x8a55,
0x7a55,0x9a3c,0xaa00,0x4a69,0xca66,0x2a0f,0x1a33,0xfa5a,0x0433,0xe45a,0xd466,0x340f,
0xb400,0x5469,0x6455,0x843c,0x743c,0x9455,0xa469,0x4400,0xc40f,0x2466,0x145a,0xf433,
0x0c3c,0xec55,0xdc69,0x3c00,0xbc0f,0x5c66,0x6c5a,0x8c33,0x7c33,0x9c5a,0xac66,0x4c0f,
0xcc00,0x2c69,0x1c55,0xfc3c,0x0255,0xe23c,0xd200,0x3269,0xb266,0x520f,0x6233,0x825a,
0x725a,0x9233,0xa20f,0x4266,0xc269,0x2200,0x123c,0xf255,0x0169,0xe100,0xd13c,0x3155,
0xb15a,0x5133,0x610f,0x8166,0x7166,0x910f,0xa133,0x415a,0xc155,0x213c,0x1100,0xf169,
0x0f00,0xef69,0xdf55,0x3f3c,0xbf33,0x5f5a,0x6f66,0x8f0f,0x7f0f,0x9f66,0xaf5a,0x4f33,
0xcf3c,0x2f55,0x1f69,0xff00
};
```

A.5.2 — Implémentation 16 bits appliquée sur 2 blocs

Nous avons utilisé l'implémentation 16 bits de référence de Robin appliquée de façon SIMD sur 2 blocs de données. Elle prend en entrée 2 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n+15$ de chaque message), 2 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 7$ correspond à la concaténation des bits $16n$ à $16n+15$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void Robin_16b(uint32_t state[8], uint32_t key[8], uint8_t dec){
    uint8_t round,i;
    uint32_t temp;
    for(i=0;i<8;i++){
        state[i] ^= key[i];
    }
```

```
    for(round=0;round<16;round++){
        if(dec^1){
            state[0] ^= LBox1[round+1]|(LBox1[round+1]<<16);
            Slayer(state);
        }
        for(i=0;i<8;i++){
            temp = (LBox2[(state[i]>>24)&0xff]^LBox1[(state[i]>>16)&0xff])<<16;
            state[i] = temp|(LBox2[(state[i]>>8)&0xff]^LBox1[state[i]&0xff]);
        }
        if(dec){
            Slayer(state);
            state[0] ^= LBox1[16-round]|(LBox1[16-round]<<16);
        }
        for(i=0;i<8;i++){
            state[i] ^= key[i];
        }
    }
}
```

La couche de substitution utilisée par Robin_16b est donnée par le code suivant :

```
void Slayer(uint32_t X[8]){
    uint8_t a, i=4;
    uint32_t Temp[4];
    for(a=0;a<3;a++){
        Temp[0] = X[i]&X[i+1];
        Temp[0] ^=X[i+2];
        Temp[2] = X[i+1]|X[i+2];
        Temp[2] ^=X[i+3];
        Temp[3] = Temp[0]&X[i+3];
        Temp[3] ^=X[i];
        Temp[1] = Temp[2]&X[i];
        Temp[1] ^=X[i+1];
        i^=4;
        X[i] ^= Temp[0];
        X[i+1] ^= Temp[1];
        X[i+2] ^= Temp[2];
        X[i+3] ^= Temp[3];
    }
}
```

Enfin, les tableaux LBox1 et LBox2 utilisés par l'étage linéaire de Robin_16b sont les mêmes que ceux utilisés par Robin_32b donnés en annexe A.5.1.

Cette implémentation permet d'appliquer Robin sur 2 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer tous les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin d'exploiter une architecture 16 bits.

Afin d'appliquer le masquage sur Robin, il est possible d'utiliser le schéma décrit en section 2.6. Il suffit pour cela que chaque mot de 32 bits en entrée soit composé d'un mot de 16 bits du masque et d'un mot de 16 bits du bloc de données masquées comme illustré à la figure 2.11, de ne pas effectuer le XOR entre le masque et la clé ou les constantes de tour, et que chaque opération & soit modifiée afin de calculer correctement la valeur des masques.

A.6 – Implémentations de SPECK

Les spécifications de SPECK sont données dans [BEAULIEU et al. 2015]. Le message clair, le chiffré ainsi que la clé sont stockés de gauche à droite dans les registres. Il est à noter que la clé de chiffrement est modifiée par l'implémentation, et doit être égale à la dernière clé de tour du chiffrement en entrée du déchiffrement.

A.6.1 – Implémentation 32 bits

Nous avons proposé une implémentation 32 bits de SPECK à partir de son code de référence afin d'exploiter pleinement une architecture 32 bits. Elle prend en entrée un message clair ou un chiffré stocké dans les registres "state", la clé secrète ou la dernière clé de tour stockée dans les registres "key" et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void SPECK_32b(uint32_t state[4], uint32_t key[4], uint8_t dec){
    uint32_t i,round;
    for(round=0;round<32;round++){
        if(dec){
            key[1] ^= 31-round;
            key[3] ^= 31-round;
            SPECKround(key,dec);
            for(i=0;i<2;i++){
                state[i] ^= key[i+2];
                state[i+2] ^= key[i+2];
            }
            SPECKround(state,dec);
        }
        else{
            SPECKround(state,dec);
            for(i=0;i<2;i++){
                state[i] ^= key[i+2];
                state[i+2] ^= key[i+2];
            }
            SPECKround(key,dec);
            key[1] ^= round;
            key[3] ^= round;
        }
    }
}
```

La fonction de tour utilisée par SPECK_32b est donnée par le code suivant :

```
void SPECKround(uint32_t state[4], uint8_t dec){
    uint32_t t0, t1, t2, t3, b=0x7fffffff;
    if(dec){
        state[2] ^= state[0];
        state[3] ^= state[1];
        t0 = state[2]&0x00000007;
        t1 = state[3]&0x00000007;
        state[2] = (state[2]>>3)|(t1<<29);
        state[3] = (state[3]>>3)|(t0<<29);
        t2 = ~state[2];
        t3 = ~state[3];
        if(t3==0xffffffff){
            t2 += 1;
        }
        t3 += 1;
        t0 = (state[1]&b)+(t3&b);
        state[0] += t2;
        if(((state[1]&t3)>b)||((state[1]&t0)>b)||((t0&t3)>b)){
            state[0] += 1;
        }
```

```
            state[1] += t3;
            t0 = state[0]&0xff000000;
            t1 = state[1]&0xff000000;
            state[0] = (state[0]<<8)|(t1>>24);
            state[1] = (state[1]<<8)|(t0>>24);
    }
    else{
            t0 = state[0]&0xff;
            t1 = state[1]&0xff;
            state[0] = (state[0]>>8)|(t1<<24);
            state[1] = (state[1]>>8)|(t0<<24);
            t0=(state[1]&b)+(state[3]&b);
            state[0] += state[2];
            if(((state[1]&state[3])>b)||((state[1]&t0)>b)||((state[3]&t0)>b)){
                state[0] += 1;
            }
            state[1] += state[3];
            t0 = state[2]&0xe0000000;
            t1 = state[3]&0xe0000000;
            state[2] = (state[2]<<3)|(t1>>29);
            state[3] = (state[3]<<3)|(t0>>29);
            state[2] ^= state[0];
            state[3] ^= state[1];
    }
}
```

A.6.2 — Implémentation 8 bits appliquée sur 4 blocs

L'implémentation 8 bits de SPECK appliquée de façon SIMD sur 4 blocs de données que nous avons proposée prend en entrée 4 messages clairs ou chiffrés stockés dans les registres "state" (le registre $0 \leq n \leq 15$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque message), 4 clés secrètes stockées dans les registres "key" (le registre $0 \leq n \leq 15$ correspond à la concaténation des bits $8n$ à $8n+7$ de chaque clé) et le paramètre "dec" égal à 0 si la fonction de chiffrement est appliquée, ou égal à 1 sinon. Son code est le suivant :

```
void SPECK_8b(uint32_t state[16], uint32_t key[16], uint8_t dec){
    uint32_t i,round;
    for(round=0;round<32;round++){
        if(dec){
            key[7] ^= 31-round;
            key[15] ^= 31-round;
            SPECKround(key,dec);
            for(i=0;i<8;i++){
                state[i] ^= key[i+8];
                state[i+8] ^= key[i+8];
            }
            SPECKround(state,dec);
        }
        else{
            SPECKround(state,dec);
            for(i=0;i<8;i++){
                state[i] ^= key[i+8];
                state[i+8] ^= key[i+8];
            }
            SPECKround(key,dec);
            key[7] ^= round;
            key[15] ^= round;
        }
    }
}
```

La fonction de tour utilisée par SPECK_8b est donnée par le code suivant :

```
void SPECKround(uint32_t state[16], uint8_t dec){
    uint32_t i,j,k,t0,t1,temp[8],result[8];
    if(dec){
        for(i=0;i<8;i++){
            state[i+8] ^= state[i];
        }
        t0 = (state[15]<<5)&0xe0e0e0e0;
        for(i=15;i>8;i--){
            state[i] = (state[i]>>3)&0x1f1f1f1f;
            state[i] |= (state[i-1]<<5)&0xe0e0e0e0;
        }
        state[8] = (state[8]>>3)&0x1f1f1f1f;
        state[8] |= t0;
        for(i=0;i<8;i++){
            result[i] = ~state[i+8];
            temp[i] = 0x01010101;
            for(j=0;j<8;j++){
             temp[i] &= result[i]>>j;
            }
        }
        for(i=0;i<7;i++){
            t0 = 0x01010101;
            for(j=i+1;j<8;j++){
                t0 &=temp[j];
            }
            result[i] = UADD8(result[i],t0);
        }
        result[7] = UADD8(result[7],0x01010101);
        for(i=7;i>0;i--){
                t0 = (state[i]&0x7f7f7f7f)+(result[i]&0x7f7f7f7f);
                t1 = t0&state[i]&result[i];
             t1 ^= t0|state[i]|result[i];
             t1 ^= t0^state[i]^result[i];
             t1 = (t1>>7)&0x01010101;
             temp[0] = 0x01010101;
             for(j=0;j<8;j++){
                 temp[0] &= state[i-1]>>j;
             }
                temp[0] &=t1;
                state[i-1] = UADD8(state[i-1],t1);
                for(j=1;j<i;j++){
                 temp[j] = 0x01010101;
                 for(k=0;k<8;k++){
                     temp[j] &= state[i-j-1]>>k;
                 }
                    temp[j] &=temp[j-1];
                    state[i-j-1] = UADD8(state[i-j-1],temp[j-1]);
                }
                state[i] = UADD8(state[i],result[i]);
        }
        state[0] = UADD8(state[0],result[0]);
        t0 = state[0];
        for(i=0;i<7;i++){
            state[i] = state[i+1];
        }
        state[7] = t0;
    }
```

```
58      else{
59              t0 = state[7];
60              for(i=7;i>0;i--){
61                  state[i] = state[i-1];
62              }
63              state[0] = t0;
64              for(i=7;i>0;i--){
65                      t0 = (state[i]&0x7f7f7f7f)+(state[i+8]&0x7f7f7f7f);
66                      t1 = t0&state[i]&state[i+8];
67                  t1 ^= t0|state[i]|state[i+8];
68                  t1 ^= t0^state[i]^state[i+8];
69                  t1 = (t1>>7)&0x01010101;
70                  temp[0] = 0x01010101;
71                  for(j=0;j<8;j++){
72                      temp[0] &=state[i-1]>>j;
73                  }
74                      temp[0] &=t1;
75                      state[i-1] = UADD8(state[i-1],t1);
76                      for(j=1;j<i;j++){
77                       temp[j] = 0x01010101;
78                       for(k=0;k<8;k++){
79                           temp[j] &= state[i-j-1]>>k;
80                       }
81                          temp[j] &=temp[j-1];
82                          state[i-j-1] = UADD8(state[i-j-1],temp[j-1]);
83                      }
84                      state[i] = UADD8(state[i],state[i+8]);
85              }
86              state[0] = UADD8(state[0],state[8]);
87              t0 = (state[8]>>5)&0x07070707;
88              for(i=8;i<15;i++){
89                  state[i] = (state[i]<<3)&0xf8f8f8f8;
90                  state[i] |= (state[i+1]>>5)&0x07070707;
91              }
92              state[15] = (state[15]<<3)&0xf8f8f8f8;
93              state[15] |= t0;
94              for(i=0;i<8;i++){
95                  state[i+8] ^= state[i];
96              }
97      }
98  }
```

où $\sim X$ ou $\overline{X}$ est l'inversion binaire de X, et la fonction UADD8 peut-être appliquée par l'instruction SIMD UADD8 sur les architectures ARM qui le permettent, ou appliquée sinon par le code donné en annexe A.3.1.

Cette implémentation permet d'appliquer SPECK sur 4 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin de l'appliquer sur 2 blocs en entrée et d'exploiter une architecture 16 bits, ou de remplacer les mots de 32 bits par des mots de 8 bits, et d'effectuer les opérations seulement sur 8 bits, afin d'exploiter une architecture 8 bits.

Pour appliquer l'IRC qui est décrit dans le chapitre 4, il suffit que chaque mot de 32 bits soit composé de deux copies d'un octet du bloc de données et d'un octet de chacun des 2 blocs de référence comme illustré à la figure 4.3.

Il est à noter que l'addition sur 64 bits demande beaucoup plus d'instructions pour être réalisée de façon SIMD sur 4 octets d'un mot de 32 bits.

A.7 — Implémentations de TRIVIUM

Les spécifications de TRIVIUM sont données en section 4.4.2.1. L'état interne est stocké de gauche à droite dans les registres.

A.7.1 — Implémentation 32 bits

Nous avons proposé une implémentation 32 bits optimisée de TRIVIUM afin d'exploiter pleinement une architecture 32 bits. Tout d'abord, sa fonction d'initialisation Init_32b prend en entrée l'état interne stocké dans les registres "IS", la clé secrète stockée dans les registres "key" et un IV stocké dans les registres "iv". Elle stocke alors la clé secrète et l'IV dans l'état interne et effectue les 36 tours d'initialisation (4·288/32) requis avant la génération des mots de la suite chiffrante. Il est à noter que la clé secrète et l'IV doivent être complétés par 16 zéros. Son code est le suivant :

```
extern uint8_t k;
void Init_32b(uint32_t IS[9], uint32_t key[3], uint32_t iv[3]){
    uint8_t i;
    k=0;
    for(i=0;i<9;i++){
        IS[i]=0;
    }
    IS[8]=0x07;
    for(i=0;i<3;i++){
        IS[i]=key[i];
    }
    IS[2]=iv[0]>>29;
    for(i=0;i<2;i++){
        IS[i+3]=(iv[i]<<3|iv[i+1]>>29);
    }
    IS[5]=iv[2]<<3;
    for(i=0;i<36;i++){
        NextWord_32b(IS);
    }
}
```

La fonction NextWord_32b de génération de la suite chiffrante, qui est également utilisée par la fonction Init_32b, est donnée par le code suivant :

```
static const uint8_t mod9[18]= {0,1,2,3,4,5,6,7,8,0,1,2,3,4,5,6,7,8};
uint8_t k;
uint32_t NextWord(uint32_t IS[9]){
    uint32_t t1,t2,t3,z;
    // Étape 1 : Calcul des registres temporaires et de la sortie
    // t1 ← s34···s65 ⊕ s61···s92
    t1=(IS[T[1+k]]<<2|IS[T[2+k]]>>30)^(IS[T[1+k]]<<28|IS[T[2+k]]>>4);
    // t2 ← s130···s161 ⊕ s145···s176
    t2=(IS[T[4+k]]<<2|IS[T[5+k]]>>30)^(IS[T[4+k]]<<17|IS[T[5+k]]>>15);
```

```
    // t3 ← s211···s242 ⊕ s256···s287
    t3=(IS[T[6+k]]<<19|IS[T[7+k]]>>13)^IS[T[8+k]];
    z=t1^t2^t3;
    // t1 ← t1 ⊕ (s59···s90&s60···s91) ⊕ s139···s170
    t1^=(IS[T[1+k]]<<27|IS[T[2+k]]>>5)&(IS[T[1+k]]<<28|IS[T[2+k]]>>4);
    t1^=(IS[T[4+k]]<<11|IS[T[5+k]]>>21);
    // t2 ← t2 ⊕ (s143···s174&s144···s175) ⊕ s232···s263
    t2^=(IS[T[4+k]]<<15|IS[T[5+k]]>>17)&(IS[T[4+k]]<<16|IS[T[5+k]]>>16);
    t2^=(IS[T[7+k]]<<8|IS[T[8+k]]>>24);
    // t3 ← t3 ⊕ (s254···s285&s255···s286) ⊕ s37···s68
    t3^=(IS[T[7+k]]<<30|IS[T[8+k]]>>2)&(IS[T[7+k]]<<31|IS[T[8+k]]>>1);
    t3^=(IS[T[1+k]]<<5|IS[T[2+k]]>>27);
    // Étape 2 : Décalage des octets de l'état interne
    k+=8;
    k=mod9[k];
    // Étape 3 : Mise à jour des octets de l'état interne
    // s0···s31 ← t3
    IS[T[0+k]]=t3;
    // s93···s124 ← t1.
    IS[T[2+k]]=(IS[T[2+k]]&0xfffffff8)|(t1>>29);
    IS[T[3+k]]=(IS[T[3+k]]&0x00000007)|(t1<<3);
    // s177···s208 ← t2
    IS[T[5+k]]=(IS[T[5+k]]&0xffff8000)|(t2>>17);
    IS[T[6+k]]=(IS[T[6+k]]&0x00007fff)|(t2<<15);
    return z;
}
```

A.7.2 — Implémentation 8 bits appliquée sur 4 blocs

Nous avons proposé une implémentation 8 bits optimisée de TRIVIUM appliquée de façon SIMD sur 4 blocs de données. Tout d'abord, sa fonction d'initialisation Init_8b prend en entrée l'état interne stocké dans les registres "IS", la clé secrète stockée dans les registres "key" et un IV stocké dans les registres "iv". Elle stocke alors la clé secrète et l'IV dans l'état interne et effectue les 144 tours d'initialisation (4·288/8) requis avant la génération des mots de la suite chiffrante. Son code est le suivant :

```
extern uint8_t k;
void Init_8b(uint32_t IS[36], uint32_t key[10], uint32_t iv[10]){
    uint8_t i;
    k=0;
    for(i=0;i<36;i++){
        IS[i]=0;
    }
    IS[35]=0x07070707;
    for(i=0;i<10;i++){
        IS[i]=key[i];
    }
    IS[11]=RSHIFT8(iv[0],5);
    for(i=0;i<9;i++){
        IS[i+12]=LSHIFT8(iv[i],3)|RSHIFT8(iv[i+1],5);
    }
    IS[21]=LSHIFT8(iv[9],3);
    for(i=0;i<144;i++){
        NextWord_8b(IS);
    }
}
```

Ensuite, la fonction NextWord_8b de génération de la suite chiffrante, qui est également utilisée par la fonction Init_8b, est donnée par le code suivant :

```
static const uint8_t mod36[72]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,
 21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,
 18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35};
uint8_t k;
#define AT(i) (mod36[(i)+(k)])
uint32_t NextWord_8b(uint32_t IS[36]){
      uint32_t t1,t2,t3,t4,z;
      // Étape 1 : Calcul des registres temporaires et de la sortie
      // t1=s65+s92.
      t1=LSHIFT8(IS[AT(7)],2)|RSHIFT8(IS[AT(8)],6);
      t1^=LSHIFT8(IS[AT(10)],5)|RSHIFT8(IS[AT(11)],3);
      // t2=s161+s176.
      t2=LSHIFT8(IS[AT(19)],2)|RSHIFT8(IS[AT(20)],6);
      t2^=LSHIFT8(IS[AT(21)],1)|RSHIFT8(IS[AT(22)],7);
      // t3=s242+s287.
      t3=(LSHIFT8(IS[AT(29)],3)|RSHIFT8(IS[AT(30)],5))^IS[AT(35)];
      z=t1^t2^t3;
      // t1=t1+s90s91+s170.
      t4=LSHIFT8(IS[AT(10)],3)|RSHIFT8(IS[AT(11)],5);
      t4&=LSHIFT8(IS[AT(10)],4)|RSHIFT8(IS[AT(11)],4);
      t1^=t4^(LSHIFT8(IS[AT(20)],3)|RSHIFT8(IS[AT(21)],5));
      // t2=t2+s174s175+s263.
      t2^=((LSHIFT8(IS[AT(20)],7)|RSHIFT8(IS[AT(21)],1))&IS[AT(21)])^IS[AT(32)];
      // t3=t3+s285s286+s68.
      t4=LSHIFT8(IS[AT(34)],6)|RSHIFT8(IS[AT(35)],2);
      t4&=LSHIFT8(IS[AT(34)],7)|RSHIFT8(IS[AT(35)],1)
      t3^=t4^(LSHIFT8(IS[AT(7)],5)|RSHIFT8(IS[AT(8)],3));
      // Étape 2 : Décalage des octets de l'état interne
      k+=35;
      k=mod36[k];
      // Étape 3 : Mise à jour des octets de l'état interne
      // s0=t3.
      IS[AT(0)]=t3;
      // s93=t1.
      IS[AT(11)]=(IS[AT(11)]&0xf8f8f8f8)|RSHIFT8(t1,5);
      IS[AT(12)]=LSHIFT8(t1,3)|(IS[AT(12)]&0x07070707);
      // s177=t2.
      IS[AT(22)]=(IS[AT(22)]&0x80808080)|RSHIFT8(t2,1);
      IS[AT(23)]=LSHIFT8(t2,7)|(IS[AT(23)]&0x7f7f7f7f);
      return z;
}
```

Enfin, Init_8b et NextWord_8b utilisent l'opération de décalage sur un octet de façon SIMD sur les mots de 32 bits, qui peut-être appliquée par le code suivant :

```
uint32_t LSHIFT8(uint32_t word, uint8_t shift){
      static const uint32_t mask[9] = {0xffffffff,0xfefefefe,0xfcfcfcfc,0xf8f8f8f8,
      0xf0f0f0f0,0xe0e0e0e0,0xc0c0c0c0,0x80808080,0x00000000};
      if(shift<9){
            return (word<<shift)&mask[shift];
      }
      else{
            return 0;
      }
}
```

```
uint32_t RSHIFT8(uint32_t word, uint8_t shift){
    static const uint32_t mask[9] = {0xffffffff,0x7f7f7f7f,0x3f3f3f3f,0x1f1f1f1f,
    0x0f0f0f0f,0x07070707,0x03030303,0x01010101,0x00000000};
    if(shift<9){
        return (word>>shift)&mask[shift];
    }
    else{
        return 0;
    }
}
```

Cette implémentation permet d'appliquer TRIVIUM sur 4 blocs en entrée, et exploite donc une architecture 32 bits, mais il est également possible de remplacer les mots de 32 bits par des mots de 16 bits, et d'effectuer les opérations seulement sur 16 bits, afin de l'appliquer sur 2 blocs en entrée et d'exploiter une architecture 16 bits, ou de remplacer les mots de 32 bits par des mots de 8 bits, et d'effectuer les opérations seulement sur 8 bits, afin d'exploiter une architecture 8 bits.

Pour appliquer l'IRC qui est décrit dans le chapitre 4, il suffit que chaque mot de 32 bits soit composé de deux copies d'un octet du bloc de données et d'un octet de chacun des 2 blocs de référence comme illustré à la figure 4.3.

Enfin, pour combiner le masquage et l'IRC, il suffit de remplacer le premier bloc de référence par un octet du masque comme illustré à la figure 4.10 et que chaque opération & et | soit modifiée afin de calculer correctement la valeur des masques. Il est à noter qu'il est nécessaire d'enlever le masque de chaque octet de la suite chiffrante au fur et à mesure que ces derniers sont générés.

Références

[Adomnicai et al. 2016] - Adomnicai, A., Lac, B., Canteaut, A., Fournier, J. J. A., Masson, L., Sirdey, R. et Tria, A. (2016). “On the importance of considering physical attacks when implementing lightweight cryptography”. In : *Lightweight Cryptography Workshop - LCW 2016*. Gaithersburg, Maryland : National Institute of Standards & Technology (cf. p. 18, 19, 33, 44).

[Adomnicai et al. 2017] - Adomnicai, A., Fournier, J. J. A. et Masson, L. (2017). “Bricklayer Attack : A Side-Channel Analysis on the ChaCha Quarter Round”. In : *INDOCRYPT 2017*. Sous la dir. d’Arpita Patra et Nigel P. Smart. T. 10698. LNCS. Chennai, India : Springer, Heidelberg, Germany, p. 65–84 (cf. p. 38, 42).

[Agoyan et al. 2010] - Agoyan, M., Dutertre, J.-M., Naccache, D., Robisson, B. et Tria, A. (2010). “When Clocks Fail : On Critical Paths and Clock Faults”. In : *CARDIS 2010*. Sous la dir. de Dieter Gollmann, Jean-Louis Lanet et Julien Iguchi-Cartigny. T. 6035. LNCS. Passau, Germany : Springer, Heidelberg, Germany, p. 182–193 (cf. p. 48).

[Agrawal et al. 2003] - Agrawal, D., Archambeault, B., Rao, J. R. et Rohatgi, P. (2003). “The EM Side-Channel(s)”. In : *CHES 2002*. Sous la dir. de Burton S. Kaliski Jr., Çetin Kaya Koç et Christof Paar. T. 2523. LNCS. Redwood Shores, CA, USA : Springer, Heidelberg, Germany, p. 29–45 (cf. p. 22).

[Ahn et al. 2016] - Ahn, S. et Choi, D. (2015). “An Improved Masking Scheme for S-Box Software Implementations”. In : *WISA 2015*. Sous la dir. d’Howon Kim et Dooho Choi. T. 9503. LNCS. Jeju Island, Korea : Springer, Heidelberg, Germany, p. 200–212 (cf. p. 41).

[Akkar et al. 2001] - Akkar, M.-L. et Giraud, C. (2001). “An Implementation of DES and AES, Secure against Some Attacks”. In : *CHES 2001*. Sous la dir. de Çetin Kaya Koç, David Naccache et Christof Paar. T. 2162. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 309–318 (cf. p. 83).

[Albrecht et al. 2014a] - Albrecht, M. R., Driessen, B., Bilge Kavun, E., Leander, G., Paar, C. et Yalçin, T. (2014a). *Block Ciphers - Focus On The Linear Layer (feat. PRIDE) : Full Version*. Cryptology ePrint Archive, Report 2014/453 : http://eprint.iacr.org/2014/453 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 117, 118).

[Albrecht et al. 2014b] - Albrecht, M. R., Driessen, B., Kavun, E. B., Leander, G., Paar, C. et Yalçin, T. (2014b). "Block Ciphers - Focus on the Linear Layer (feat. PRIDE)". In : *CRYPTO 2014, Part I*. Sous la dir. de Juan A. Garay et Rosario Gennaro. T. 8616. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 57–76 (cf. p. 11, 14, 17, 18, 32–34, 54, 118).

[ANSSI 2014] - ANSSI (2014). *Référentiel Général de Sécurité version 2.0 - Annexe B1*. ANSSI : https://www.ssi.gouv.fr/uploads/2014/11/RGS_v-2-0_B1.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 13).

[Aoki et al. 2001] - Aoki, K., Ichikawa, T., Kanda, M., Matsui, M., Moriai, S., Nakajima, J. et Tokita, T. (2001). "Camellia : A 128-Bit Block Cipher Suitable for Multiple Platforms - Design and Analysis". In : *SAC 2000*. Sous la dir. de Douglas R. Stinson et Stafford E. Tavares. T. 2012. LNCS. Waterloo, Ontario, Canada : Springer, Heidelberg, Germany, p. 39–56 (cf. p. 11).

[Aumasson et al. 2009] - Aumasson, J.-P., Dinur, I., Meier, W. et Shamir, A. (2009). "Cube Testers and Key Recovery Attacks on Reduced-Round MD6 and Trivium". In : *FSE 2009*. Sous la dir. d'Orr Dunkelman. T. 5665. LNCS. Leuven, Belgium : Springer, Heidelberg, Germany, p. 1–22 (cf. p. 105, 130).

[Babbage et al. 2008] - Babbage, S. et Dodd, M. (2008). "The MICKEY Stream Ciphers". In : *New Stream Cipher Designs - The eSTREAM Finalists*. Sous la dir. de Matthew J. B. Robshaw et Olivier Billet. T. 4986. LNCS. Springer, Heidelberg, Germany, p. 191–209 (cf. p. 10, 14, 92).

[Bai et al. 2009] - Bai, X., Huang, L., Wang, Y. et Xu, Y. (2009). "Differential Power Analysis Attack on CLEFIA Block Cipher". In : *2009 International Conference on Computational Intelligence and Software Engineering*. Wuhan, China : IEEE Computer Society, p. 1–4 (cf. p. 38).

[Banik et al. 2015] - Banik, S., Bogdanov, A., Isobe, T., Shibutani, K., Hiwatari, H., Akishita, T. et Regazzoni, F. (2015). "Midori : A Block Cipher for Low Energy". In : *ASIACRYPT 2015, Part II*. Sous la dir. de Tetsu Iwata et Jung Hee Cheon. T. 9453. LNCS. Auckland, New Zealand : Springer, Heidelberg, Germany, p. 411–436 (cf. p. 14).

[Bar-El et al. 2006] - Bar-El, H., Choukri, H., Naccache, D., Tunstall, M. et Whelan, C. (2006). "The Sorcerer's Apprentice Guide to Fault Attacks". In : *Proceedings of the IEEE* 94.2, p. 370–382 (cf. p. 46).

[Barenghi et al. 2012] - Barenghi, A. et Trichina, E. (2012). “Fault Attacks on Stream Ciphers”. In : *Fault Analysis in Cryptography*. Sous la dir. de Marc Joye et Michael Tunstall. Information Security and Cryptography. Springer, Heidelberg, Germany, p. 239–255 (cf. p. 81).

[Barreto et al. 2000] - Barreto, P. et Rijmen, V. (2000). “The Khazad legacy-level block cipher”. In : *First open NESSIE Workshop*. Leuven, Belgium (cf. p. 11).

[Barton 2017] - Barton, D. (2017). *Unsecured IoT – A Dangerous Gambit !* A2N : http://www.a2n.net/2017/03/20/unsecured-iot/ (en ligne, dernier accès le 29 octobre 2018) (cf. p. 2).

[Bassham,III et al. 2010] - Bassham III, L. E., Rukhin, A. L., Soto, J., Nechvatal, J. R., Smid, M. E., Barker, E. B., Leigh, S. D., Levenson, M., Vangel, M., Banks, D. L., Heckert, N. A., Dray, J. F. et Vo, S. (2010). “SP 800-22 Rev. 1a. A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications”. In : *Technical Report*. Gaithersburg, MD, United States : National Institute of Standards & Technology (cf. p. 10, 143).

[Battistello et al. 2016] - Battistello, A., Coron, J.-S., Prouff, E. et Zeitoun, R. (2016). “Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme”. In : *CHES 2016*. Sous la dir. de Benedikt Gierlichs et Axel Y. Poschmann. T. 9813. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 23–39 (cf. p. 41).

[Beaulieu et al. 2015] - Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B. et Wingers, L. (2015). “The SIMON and SPECK lightweight block ciphers”. In : *Proceedings of the 52nd Annual Design Automation Conference 2015*. San Francisco, CA, USA : ACM, 175 :1–175 :6 (cf. p. 11, 14, 143, 178).

[Beierle et al. 2016] - Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P. et Sim, S. M. (2016). “The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS”. In : *CRYPTO 2016, Part II*. Sous la dir. de Matthew Robshaw et Jonathan Katz. T. 9815. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 123–153 (cf. p. 14).

[Beierle et al. 2017] - Beierle, C., Canteaut, A., Leander, G. et Rotella, Y. (2017). “Proving Resistance Against Invariant Attacks : How to Choose the Round Constants”. In : *CRYPTO 2017, Part II*. Sous la dir. de Jonathan Katz et Hovav Shacham. T. 10402. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 647–678 (cf. p. 132).

[Bellare et al. 1996] - Bellare, M., Canetti, R. et Krawczyk, H. (1996). “Keying Hash Functions for Message Authentication”. In : *CRYPTO’96*. Sous la dir. de Neal Koblitz. T. 1109. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 1–15 (cf. p. 7).

[Berger et al. 2015] - Berger, T. P., Francq, J. et Minier, M. (2015). “CUBE Cipher : A Family of Quasi-Involutive Block Ciphers Easy to Mask”. In : *C2SI 2015*. Sous la dir. de Said El Hajji, Abderrahmane Nitaj, Claude Carlet et El Mamoun Souidi. T. 9084. LNCS. Rabat, Morocco : Springer, Heidelberg, Germany, p. 89–105 (cf. p. 32, 54, 59).

[Bernstein 2005] - Bernstein, D. J. (2005). *Cache-timing attacks on AES*. cr.yp.to : https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 27).

[Bernstein 2007] - Bernstein, D. J. et Lange, T. (2007). *eBACS : ECRYPT Benchmarking of Cryptographic Systems*. eBACS : https://bench.cr.yp.to/ (en ligne, dernier accès le 29 octobre 2018) (cf. p. 12).

[Bernstein 2008] - Bernstein, D. J. (2008). “The Salsa20 Family of Stream Ciphers”. In : *New Stream Cipher Designs - The eSTREAM Finalists*. Sous la dir. de Matthew J. B. Robshaw et Olivier Billet. T. 4986. LNCS. Springer, Heidelberg, Germany, p. 84–97 (cf. p. 10, 14).

[Bernstein 2014] - Bernstein, D. J. (2014). *CAESAR : Competition for Authenticated Encryption*. CAESAR competition : https://competitions.cr.yp.to/caesar.html (en ligne, dernier accès le 29 octobre 2018) (cf. p. 69).

[Bertoni et al. 2002] - Bertoni, G., Breveglieri, L., Koren, I., Maistri, P. et Piuri, V. (2002). “A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard”. In : *DFT 2002*. Vancouver, BC, Canada : IEEE Computer Society, p. 51–59 (cf. p. 82).

[Bertoni et al. 2003] - Bertoni, G., Breveglieri, L., Fragneto, P., Macchetti, M. et Marchesin, S. (2003). “Efficient Software Implementation of AES on 32-Bit Platforms”. In : *CHES 2002*. Sous la dir. de Burton S. Kaliski Jr., Çetin Kaya Koç et Christof Paar. T. 2523. LNCS. Redwood Shores, CA, USA : Springer, Heidelberg, Germany, p. 159–171 (cf. p. 146).

[Berzati et al. 2009] - Berzati, A., Canovas, C., Dumas, J.-G. et Goubin, L. (2009). “Fault Attacks on RSA Public Keys : Left-To-Right Implementations Are Also Vulnerable”. In : *CT-RSA 2009*. Sous la dir. de Marc Fischlin. T. 5473. LNCS. San Francisco, CA, USA : Springer, Heidelberg, Germany, p. 414–428 (cf. p. 53, 81).

[Biehl et al. 2000] - Biehl, I., Meyer, B. et Müller, V. (2000). “Differential Fault Attacks on Elliptic Curve Cryptosystems”. In : *CRYPTO 2000*. Sous la dir. de Mihir Bellare. T. 1880. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 131–146 (cf. p. 53).

[Biham et al. 1991] - Biham, E. et Shamir, A. (1991). “Differential Cryptanalysis of DES-like Cryptosystems”. In : *CRYPTO’90*. Sous la dir. d’Alfred J. Menezes et Scott A. Vanstone. T. 537. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 2–21 (cf. p. 114).

[Biham et al. 1997] - Biham, E. et Shamir, A. (1997). “Differential Fault Analysis of Secret Key Cryptosystems”. In : *CRYPTO’97*. Sous la dir. de Burton S. Kaliski Jr. T. 1294. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 513–525 (cf. p. 52).

[Biryukov et al. 1999] - Biryukov, A. et Wagner, D. (1999). “Slide Attacks”. In : *FSE’99*. Sous la dir. de Lars R. Knudsen. T. 1636. LNCS. Rome, Italy : Springer, Heidelberg, Germany, p. 245–259 (cf. p. 122).

[Biryukov et al. 2016] - Biryukov, A., Dinu, D. et Grosssch ädl, J. (2016). “Correlation Power Analysis of Lightweight Block Ciphers : From Theory to Practice”. In : *ACNS 2016*. Sous la dir. de Mark Manulis, Ahmad-Reza Sadeghi et Steve Schneider. T. 9696. LNCS. Guildford, UK : Springer, Heidelberg, Germany, p. 537–557 (cf. p. 31).

[Biryukov et al. 2017] - Biryukov, A. et Perrin, L. P. (2017). *State of the Art in Lightweight Symmetric Cryptography*. Open Repository and Bibliography : http://orbilu.uni.lu/handle/10993/31319 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 13, 14).

[Bogdanov et al. 2007] - Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y. et Vikkelsoe, C. (2007). “PRESENT : An Ultra-Lightweight Block Cipher”. In : *CHES 2007*. Sous la dir. de Pascal Paillier et Ingrid Verbauwhede. T. 4727. LNCS. Vienna, Austria : Springer, Heidelberg, Germany, p. 450–466 (cf. p. 11, 14, 61).

[Boneh et al. 1997] - Boneh, D., DeMillo, R. A. et Lipton, R. J. (1997). “On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract)”. In : *EUROCRYPT’97*. Sous la dir. de Walter Fumy. T. 1233. LNCS. Konstanz, Germany : Springer, Heidelberg, Germany, p. 37–51 (cf. p. 51).

[Borghoff et al. 2012] - Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knezevic, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S. et Yalçın, T. (2012). “PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract”. In : *ASIACRYPT 2012*. Sous la dir. de Xiaoyun Wang et Kazue Sako. T. 7658. LNCS. Beijing, China : Springer, Heidelberg, Germany, p. 208–225 (cf. p. 11, 14).

[Boscher et al. 2008] - Boscher, A. et Handschuh, H. (2008). “Masking Does Not Protect Against Differential Fault Attacks”. In : *FDTC 2008*. Sous la dir. de Luca Breveglieri, Shay Gueron, Israel Koren, David Naccache et Jean-Pierre Seifert. Washington, DC, USA : IEEE Computer Society, p. 35–40 (cf. p. 87).

[Bouarroudj-Berkani 2008] - Bouarroudj-Berkani, M. (2008). “Study of thermomechanical fatigue of power electronic modules under high environemental temperatures for automotive applications”. Thèse. École normale supérieure de Cachan - ENS Cachan, France (cf. p. 23).

[Boura et al. 2011] - Boura, C., Canteaut, A. et De Cannière, C. (2011). “Higher-Order Differential Properties of Keccak and Luffa”. In : *FSE 2011*. Sous la dir. d’Antoine Joux. T. 6733. LNCS. Lyngby, Denmark : Springer, Heidelberg, Germany, p. 252–269 (cf. p. 129, 130).

[Boura et al. 2013] - Boura, C. et Canteaut, A. (2013). “On the Influence of the Algebraic Degree of F^{-1} on the Algebraic Degree of $G \circ F$”. In : *IEEE Trans. Information Theory* 59.1, p. 691–702 (cf. p. 130).

[Brassard et al. 1996] - Brassard, G. et Bratley, P. (1996). *Fundamentals of Algorithmics*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc. (cf. p. 27).

[Briceno et al. 1999] - Briceno, M., Goldberg, I. et Wagner, D. (1999). *A pedagogical implementation of A5/1*. Scard : http://www.scard.org/gsm/a51.html (en ligne, dernier accès le 29 octobre 2018) (cf. p. 10, 14).

[Brier et al. 2004] - Brier, E., Clavier, C. et Olivier, F. (2004). “Correlation Power Analysis with a Leakage Model”. In : *CHES 2004*. Sous la dir. de Marc Joye et Jean-Jacques Quisquater. T. 3156. LNCS. Cambridge, Massachusetts, USA : Springer, Heidelberg, Germany, p. 16–29 (cf. p. 29).

[Brouchier et al. 2009] - Brouchier, J., Kean, T., Marsh, C. et Naccache, D. (2009). “Temperature Attacks”. In : *IEEE Security & Privacy* 7.2, p. 79–82 (cf. p. 20, 23, 47).

[Canteaut 2012] - Canteaut, A. (2012). *Turing à l’assaut d’Enigma*. Interstices.info : https://interstices.info/jcms/int_70884/turing-a-l-assaut-d-enigma (en ligne, dernier accès le 29 octobre 2018) (cf. p. 5).

[Canteaut et al. 2015] - Canteaut, A., Duval, S. et Leurent, G. (2015). *Construction of Lightweight S-Boxes using Feistel and MISTY structures (Full Version)*. Cryptology ePrint Archive, Report 2015/711 : http://eprint.iacr.org/2015/711 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 115).

[Canteaut et al. 2016a] - Canteaut, A., Carpov, S., Fontaine, C., Lepoint, T., Naya-Plasencia, M., Paillier, P. et Sirdey, R. (2016a). “Stream Ciphers : A Practical Solution for Efficient Homomorphic-Ciphertext Compression”. In : *FSE 2016*. Sous la dir. de Thomas Peyrin. T. 9783. LNCS. Bochum, Germany : Springer, Heidelberg, Germany, p. 313–333 (cf. p. 105).

[Canteaut et al. 2016b] - Canteaut, A., Duval, S. et Leurent, G. (2016b). “Construction of Lightweight S-Boxes Using Feistel and MISTY Structures”. In : *SAC 2015*. Sous la dir. d’Orr Dunkelman et Liam Keliher. T. 9566. LNCS. Sackville, NB, Canada : Springer, Heidelberg, Germany, p. 373–393 (cf. p. 115, 117).

[Canteaut et al. 2017] - Canteaut, A., Carpov, S., Fontaine, C., Fournier, J. J. A., Lac, B., Naya-Plasencia, M., Sirdey, R. et Tria, A. (2017). “End-to-end data security for IoT : from a cloud of encryptions to encryption in the cloud”. In : *Computer & Electronics Security Applications Rendez-vous - C&ESAR 2017*. Rennes, France (cf. p. 18, 19).

[Carlet et al. 2012] - Carlet, C., Goubin, L., Prouff, E., Quisquater, M. et Rivain, M. (2012). “Higher-Order Masking Schemes for S-Boxes”. In : *FSE 2012*. Sous la dir. d’Anne Canteaut. T. 7549. LNCS. Washington, DC, USA : Springer, Heidelberg, Germany, p. 366–384 (cf. p. 41).

[Carmon et al. 2016] - Carmon, E., Seifert, J.-P. et Wool, A. (2016). “Simple Photonic Emission Attack with Reduced Data Complexity”. In : *COSADE 2016*. Sous la dir. de François-Xavier Standaert et Elisabeth Oswald. T. 9689. LNCS. Graz, Austria : Springer, Heidelberg, Germany, p. 35–51 (cf. p. 24).

[Carpi et al. 2013] - Carpi, R. B., Picek, S., Batina, L., Menarini, F., Jakobovic, D. et Golub, M. (2013). “Glitch It If You Can : Parameter Search Strategies for Successful Fault Injection”. In : *CARDIS 2013*. Sous la dir. d’Aurélien Francillon et Pankaj Rohatgi. T. 8419. LNCS. Berlin, Germany : Springer, Heidelberg, Germany, p. 236–252 (cf. p. 46).

[Chari et al. 1999] - Chari, S., Jutla, C. S., Rao, J. R. et Rohatgi, P. (1999). “Towards Sound Approaches to Counteract Power-Analysis Attacks”. In : *CRYPTO’99*. Sous la dir. de Michael J. Wiener. T. 1666. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 398–412 (cf. p. 29).

[Chari et al. 2003] - Chari, S., Rao, J. R. et Rohatgi, P. (2003). “Template Attacks”. In : *CHES 2002*. Sous la dir. de Burton S. Kaliski Jr., Çetin Kaya Koç et Christof Paar. T. 2523. LNCS. Redwood Shores, CA, USA : Springer, Heidelberg, Germany, p. 13–28 (cf. p. 30).

[Clavier et al. 2000] - Clavier, C., Coron, J.-S. et Dabbous, N. (2000). “Differential Power Analysis in the Presence of Hardware Countermeasures”. In : *CHES 2000*. Sous la dir. de Çetin Kaya Koç et Christof Paar. T. 1965. LNCS. Worcester, Massachusetts, USA : Springer, Heidelberg, Germany, p. 252–263 (cf. p. 39).

[Coron et al. 2003] - Coron, J.-S. et Tchulkine, A. (2003). “A New Algorithm for Switching from Arithmetic to Boolean Masking”. In : *CHES 2003*. Sous la dir. de Colin D. Walter, Çetin Kaya Koç et Christof Paar. T. 2779. LNCS. Cologne, Germany : Springer, Heidelberg, Germany, p. 89–97 (cf. p. 42).

[CypherPunks 1994] - CypherPunks (1994). *RC4 Source Code*. CypherPunks : http://cypherpunks.venona.com/date/1994/09/msg00304.html (en ligne, dernier accès le 29 octobre 2018) (cf. p. 10).

[Daemen 1995] - Daemen, J. (1995). “Cipher and hash function design strategies based on linear and differential cryptanalysis”. Thèse. K.U. Leuven, Belgium (cf. p. 118).

[Daemen et al. 2000] - Daemen, J., Peeters, M., Van Assche, G. et Rijmen, V. (2000). “The NOEKEON block cipher”. In : *First open NESSIE Workshop*. Leuven, Belgium (cf. p. 14).

[Daemen et al. 2002] - Daemen, J. et Rijmen, V. (2002). “Specification of Rijndael”. In : *The Design of Rijndael : AES — The Advanced Encryption Standard*. Springer, Heidelberg, Germany, p. 31–51 (cf. p. 8, 128, 146).

[Daemen et al. 2006] - Daemen, J. et Rijmen, V. (2006). “Understanding Two-Round Differentials in AES”. In : *SCN 06*. Sous la dir. de Roberto De Prisco et Moti Yung. T. 4116. LNCS. Maiori, Italy : Springer, Heidelberg, Germany, p. 78–94 (cf. p. 128).

[Danger et al. 2009] - Danger, J. L., Guilley, S., Bhasin, S. et Nassar, M. (2009). “Overview of Dual rail with Precharge logic styles to thwart implementation-level attacks on hardware cryptoprocessors”. In : *SCS 2009*. Medenine, Tunisia : IEEE Computer Society, p. 1–8 (cf. p. 38).

[De Cannière 2006] - De Cannière, C. (2006). “Trivium : A Stream Cipher Construction Inspired by Block Cipher Design Principles”. In : *ISC 2006*. Sous la dir. de Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis et Bart Preneel. T. 4176. LNCS. Samos Island, Greece : Springer, Heidelberg, Germany, p. 171–186 (cf. p. 10, 14, 104).

[De Cannière et al. 2009] - De Cannière, C., Dunkelman, O. et Knezevic, M. (2009). “KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers”. In : *CHES 2009*. Sous la dir. de Christophe Clavier et Kris Gaj. T. 5747. LNCS. Lausanne, Switzerland : Springer, Heidelberg, Germany, p. 272–288 (cf. p. 14).

[Debande et al. 2012] - Debande, N., Souissi, Y., Aabid, M. A. E., Guilley, S. et Danger, J.-L. (2012). “Wavelet transform based pre-processing for side channel analysis”. In : *MICRO 2012*. Vancouver, BC, Canada : IEEE Computer Society, p. 32–38 (cf. p. 26).

[Dehbaoui et al. 2012] - Dehbaoui, A., Dutertre, J.-M., Robisson, B. et Tria, A. (2012). “Electromagnetic Transient Faults Injection on a Hardware and a Software Implementations of AES”. In : *FDTC 2012*. Sous la dir. de Guido Bertoni et Benedikt Gierlichs. Leuven, Belgium : IEEE Computer Society, p. 7–15 (cf. p. 47).

[Diffie et al. 1976] - Diffie, W. et Hellman, M. (1976). “New directions in cryptography”. In : *IEEE Trans. Information Theory* 22.6, p. 644–654 (cf. p. 13).

[Ding et al. 2009] - Ding, G. L., Chu, J., Yuan, L. et Zhao, Q. (2009). “Correlation Electromagnetic Analysis for Cryptographic Device”. In : *PACCS 2009*. Chengdu, China : IEEE Computer Society, p. 388–391 (cf. p. 30).

[Dinu et al. 2016] - Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J. et Biryukov, A. (2016). “Design Strategies for ARX with Provable Bounds : Sparx and LAX”. In : *ASIACRYPT 2016, Part I*. Sous la dir. de Jung Hee Cheon et Tsuyoshi Takagi. T. 10031. LNCS. Hanoi, Vietnam : Springer, Heidelberg, Germany, p. 484–513 (cf. p. 14, 143).

[Dinur et al. 2009] - Dinur, I. et Shamir, A. (2009). “Cube Attacks on Tweakable Black Box Polynomials”. In : *EUROCRYPT 2009*. Sous la dir. d’Antoine Joux. T. 5479. LNCS. Cologne, Germany : Springer, Heidelberg, Germany, p. 278–299 (cf. p. 105).

[Djellid-Ouar et al. 2006] - Djellid-Ouar, A., Cathebras, G. et Bancel, F. (2006). “Supply voltage glitches effects on CMOS circuits”. In : *DTIS 2006*. Tunis, Tunisia : IEEE Computer Society, p. 257–261 (cf. p. 46).

[Dobraunig et al. 2016] - Dobraunig, C., Eichlseder, M., Korak, T., Lomné, V. et Mendel, F. (2016). “Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes”. In : *ASIACRYPT 2016, Part I*. Sous la dir. de Jung Hee Cheon et Tsuyoshi Takagi. T. 10031. LNCS. Hanoi, Vietnam : Springer, Heidelberg, Germany, p. 369–395 (cf. p. 83).

[Drioton 1953] - Drioton, É. (1953). “Les principes de la cryptographie égyptienne”. In : *Comptes rendus des séances de l’Académie des Inscriptions et Belles-Lettres* 97.3, p. 355–364 (cf. p. 4).

[Dutertre et al. 2011] - Dutertre, J.-M., Fournier, J. J. A., Mirbaha, A. P., Naccache, D., Rigaud, J. B., Robisson, B. et Tria, A. (2011). “Review of fault injection mechanisms and consequences on countermeasures design”. In : *DTIS 2011*. IEEE Computer Society, p. 1–6 (cf. p. 49).

[Dutertre et al. 2012] - Dutertre, J.-M., Mirbaha, A. P., Naccache, D., Ribotta, A. L., Tria, A. et Vaschalde, T. (2012). “Fault Round Modification Analysis of the advanced encryption standard”. In : *HOST 2012*. San Francisco, CA, USA : IEEE Computer Society, p. 140–145 (cf. p. 53).

[Dworkin 2007] - Dworkin, M. J. (2007). “SP 800-38D. Recommendation for Block Cipher Modes of Operation : Galois/Counter Mode (GCM) and GMAC”. In : *Technical Report*. Gaithersburg, MD, United States : National Institute of Standards & Technology (cf. p. 7).

[ECRYPT 2008] - ECRYPT (2008). *eSTREAM : the ECRYPT Stream Cipher Project*. ECRYPT : http://www.ecrypt.eu.org/stream/index.html (en ligne, dernier accès le 29 octobre 2018) (cf. p. 104).

[Eisenbarth et al. 2010] - Eisenbarth, T., Paar, C. et Weghenkel, B. (2010). “Building a Side Channel Based Disassembler”. In : *Transactions on Computational Science X - Special Issue on Security in Computing, Part I*. Sous la dir. de Marina L. Gavrilova, Chih Jeng Kenneth Tan et Edward D. Moreno. T. 6340. LNCS. Springer, Heidelberg, Germany, p. 78–99 (cf. p. 27).

[El-Baze et al. 2016] - El-Baze, D., Rigaud, J.-B. et Maurine, P. (2016). “A fully-digital EM pulse detector”. In : *DATE 2016*. Sous la dir. de Luca Fanucci et Jürgen Teich. Dresden, Germany : IEEE Computer Society, p. 439–444 (cf. p. 82).

[ElGamal 1984] - ElGamal, T. (1984). “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”. In : *CRYPTO’84*. Sous la dir. de G. R. Blakley et David Chaum. T. 196. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 10–18 (cf. p. 8).

[ETSI/SAGE 2006] - ETSI/SAGE (2006). *Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2 : SNOW 3G Specification*. ETSI/SAGE : http://cryptome.org/uea2-uia2/snow_3g_spec.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 14).

[Euler 1763] - Euler, L. (1763). “Theoremata arithmetica nova methodo demonstrata”. In : *Novi Commentarii academiae scientiarum Petropolitanae* 8, p. 74–104 (cf. p. 8).

[FANG et al. 2015] - FANG, X., LUO, P., FEI, Y. et LEESER, M. (2015). “Balance power leakage to fight against side-channel analysis at gate level in FPGAs”. In : *ASAP 2015*. Toronto, ON, Canada : IEEE Computer Society, p. 154–155 (cf. p. 38).

[FISCHER et al. 2006] - FISCHER, W., GAMMEL, B. M., KNIFFLER, O. et VELTEN, J. (2007). “Differential Power Analysis of Stream Ciphers”. In : *CT-RSA 2007*. Sous la dir. de Masayuki ABE. T. 4377. LNCS. San Francisco, CA, USA : Springer, Heidelberg, Germany, p. 257–270 (cf. p. 105).

[FLUHRER 2001] - FLUHRER, S. et LUCKS, S. (2001). “Analysis of the E0 Encryption System”. In : *SAC 2001*. Sous la dir. de Serge VAUDENAY et Amr M. YOUSSEF. T. 2259. LNCS. Toronto, Ontario, Canada : Springer, Heidelberg, Germany, p. 38–48 (cf. p. 10, 14).

[FOUQUE et al. 2013] - FOUQUE, P.-A. et VANNET, T. (2014). “Improving Key Recovery to 784 and 799 Rounds of Trivium Using Optimized Cube Attacks”. In : *FSE 2013*. Sous la dir. de Shiho MORIAI. T. 8424. LNCS. Singapore : Springer, Heidelberg, Germany, p. 502–517 (cf. p. 105).

[GANDOLFI et al. 2001] - GANDOLFI, K., MOURTEL, C. et OLIVIER, F. (2001). “Electromagnetic Analysis : Concrete Results”. In : *CHES 2001*. Sous la dir. de Çetin Kaya KOÇ, David NACCACHE et Christof PAAR. T. 2162. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 251–261 (cf. p. 20).

[GÉRARD et al. 2013] - GÉRARD, B., GROSSO, V., NAYA-PLASENCIA, M. et STANDAERT, F.-X. (2013). “Block Ciphers That Are Easier to Mask : How Far Can We Go ?” In : *CHES 2013*. Sous la dir. de Guido BERTONI et Jean-Sébastien CORON. T. 8086. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 383–399 (cf. p. 14, 15, 42).

[GIERLICHS et al. 2008] - GIERLICHS, B., BATINA, L., TUYLS, P. et PRENEEL, B. (2008). “Mutual Information Analysis”. In : *CHES 2008*. Sous la dir. d’Elisabeth OSWALD et Pankaj ROHATGI. T. 5154. LNCS. Washington, D.C., USA : Springer, Heidelberg, Germany, p. 426–442 (cf. p. 29).

[GIERLICHS et al. 2010] - GIERLICHS, B., BATINA, L., PRENEEL, B. et VERBAUWHEDE, I. (2010). “Revisiting Higher-Order DPA Attacks : Multivariate Mutual Information Analysis”. In : *CT-RSA 2010*. Sous la dir. de J. PIEPRZYK. San Francisco, CA, USA : Springer, Heidelberg, Germany, p. 221–234 (cf. p. 30).

[GIRAUD 2004] - GIRAUD, C. (2004). “DFA on AES”. In : *AES 2004*. Sous la dir. d’Hans DOBBERTIN, Vincent RIJMEN et Aleksandra SOWA. T. 3373. LNCS. Bonn, Germany : Springer, Heidelberg, Germany, p. 27–41 (cf. p. 52).

[GOLOMB 1981] - GOLOMB, S. W. (1981). *Shift Register Sequences*. Laguna Hills, CA, USA : Aegean Park Press (cf. p. 10).

[Gong et al. 2012] - Gong, Z., Nikova, S. et Law, Y. W. (2011). “KLEIN : A New Family of Lightweight Block Ciphers”. In : *RFIDSec 2011*. Sous la dir. d’Ari Juels et Christof Paar. T. 7055. LNCS. Amherst, MA, USA : Springer, Heidelberg, Germany, p. 1–18 (cf. p. 11, 14).

[Goubin 2001] - Goubin, L. (2001). “A Sound Method for Switching between Boolean and Arithmetic Masking”. In : *CHES 2001*. Sous la dir. de Çetin Kaya Koç, David Naccache et Christof Paar. T. 2162. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 3–15 (cf. p. 42).

[Goudarzi et al. 2016] - Goudarzi, D. et Rivain, M. (2016). “On the Multiplicative Complexity of Boolean Functions and Bitsliced Higher-Order Masking”. In : *CHES 2016*. Sous la dir. de Benedikt Gierlichs et Axel Y. Poschmann. T. 9813. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 457–478 (cf. p. 42).

[Goudarzi et al. 2017] - Goudarzi, D. et Rivain, M. (2017). “How Fast Can Higher-Order Masking Be in Software ?” In : *EUROCRYPT 2017, Part I*. Sous la dir. de Jean-Sébastien Coron et Jesper Buus Nielsen. T. 10210. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 567–597 (cf. p. 42).

[Grosso et al. 2014a] - Grosso, V., Leurent, G., Standaert, F.-X. et Varici, K. (2015). “LS-Designs : Bitslice Encryption for Efficient Masked Software Implementations”. In : *FSE 2014*. Sous la dir. de Carlos Cid et Christian Rechberger. T. 8540. LNCS. London, UK : Springer, Heidelberg, Germany, p. 18–37 (cf. p. 14, 15, 17, 32, 42, 54, 58, 153, 174).

[Grosso et al. 2014b] - Grosso, V., Leurent, G., Standaert, F.-X., Varici, K., Durvaux, F., Gaspar, L. et Kerckhof, S. (2014). *SCREAM & iSCREAM*. Soumission à la compétition CAESAR. HAL : https://hal.inria.fr/hal-01093512 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 69, 76, 77).

[Guilley et al. 2004] - Guilley, S., Hoogvorst, P. et Pacalet, R. (2004). “Differential Power Analysis Model and Some Results”. In : *CARDIS 2004*. Sous la dir. de Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte et Anas Abou El Kalam. T. 153. IFIP. Toulouse, France : Kluwer et Springer, Heidelberg, Germany, p. 127–142 (cf. p. 20).

[Guilley et al. 2008] - Guilley, Sylvain, Sauvage, Laurent, Danger, Jean-Luc, Graba, Tarik et Mathieu, Yves (2008). “Evaluation of Power-Constant Dual-Rail Logic as a Protection of Cryptographic Applications in FPGAs”. In : *SSIRI 2008*. Yokohama, Japan : IEEE Computer Society, p. 16–23 (cf. p. 38).

[Guilley et al. 2010] - Guilley, S., Sauvage, L., Danger, J.-L. et Selmane, N. (2010). “Fault Injection Resilience”. In : *FDTC 2010*. Sous la dir. de Luca Breveglieri, Marc Joye, Israel Koren, David Naccache et Ingrid Verbauwhede. Santa Barbara, CA, USA : IEEE Computer Society, p. 51–65 (cf. p. 84, 85).

[Guo et al. 2011] - Guo, J., Peyrin, T., Poschmann, A. et Robshaw, M. (2011). “The LED Block Cipher”. In : *CHES 2011*. Sous la dir. de Bart Preneel et Tsuyoshi Takagi. T. 6917. LNCS. Nara, Japan : Springer, Heidelberg, Germany, p. 326–341 (cf. p. 14).

[Hanson 1975] - Hanson, R. J. (1975). “Stably Updating Mean and Standard Deviation of Data”. In : *Commun. ACM* 18.1, p. 57–58 (cf. p. 25).

[Hell et al. 2007] - Hell, M., Johansson, T. et Meier, W. (2007). “Grain : a stream cipher for constrained environments”. In : *IJWMC* 2.1, p. 86–93 (cf. p. 10, 14).

[Herbst et al. 2006] - Herbst, C., Oswald, E. et Mangard, S. (2006). “An AES Smart Card Implementation Resistant to Power Analysis Attacks”. In : *ACNS 2006*. Sous la dir. de Jianying Zhou, Moti Yung et Feng Bao. T. 3989. LNCS. Singapore, p. 239–252 (cf. p. 39).

[Hinsley 1992] - Hinsley, H. (1992). *The Influence of ULTRA in the Second World War*. King’s College London : https://www.kcl.ac.uk/library/archivespec/documents/archivesdocs/1992-lecture.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 6).

[Hojsík et al. 2008] - Hojsík, M. et Rudolf, B. (2008). “Differential Fault Analysis of Trivium”. In : *FSE 2008*. Sous la dir. de Kaisa Nyberg. T. 5086. LNCS. Lausanne, Switzerland : Springer, Heidelberg, Germany, p. 158–172 (cf. p. 105).

[Hong et al. 2001] - Hong, S., Lee, S., Lim, J., Sung, J., Cheon, D. H. et Cho, I. (2001). “Provable Security against Differential and Linear Cryptanalysis for the SPN Structure”. In : *FSE 2000*. Sous la dir. de Bruce Schneier. T. 1978. LNCS. New York, NY, USA : Springer, Heidelberg, Germany, p. 273–283 (cf. p. 128).

[Hong et al. 2006] - Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B.-S., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J. et Chee, S. (2006). “HIGHT : A New Block Cipher Suitable for Low-Resource Device”. In : *CHES 2006*. Sous la dir. de Louis Goubin et Mitsuru Matsui. T. 4249. LNCS. Yokohama, Japan : Springer, Heidelberg, Germany, p. 46–59 (cf. p. 14).

[Hong et al. 2013] - Hong, D., Lee, J.-K., Kim, D.-C., Kwon, D., Ryu, K. H. et Lee, D.-G. (2013). “LEA : A 128-Bit Block Cipher for Fast Encryption on Common Processors”. In : *WISA 2013*. Sous la dir. d’Yongdae Kim, Heejo Lee et Adrian Perrig. T. 8267. LNCS. Jeju Island, Korea : Springer, Heidelberg, Germany, p. 3–27 (cf. p. 14).

[Howgrave-Graham et al. 2001] - Howgrave-Graham, N., Dyer, J. G. et Gennaro, R. (2001). “Pseudo-random Number Generation on the IBM 4758 Secure Crypto Coprocessor”. In : *CHES 2001*. Sous la dir. de Çetin Kaya Koç, David Naccache et Christof Paar. T. 2162. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 93–102 (cf. p. 143).

[Hu et al. 2012] - Hu, Y., Gao, J., Liu, Q. et Zhang, Y. (2012). “Fault analysis of Trivium”. In : *Des. Codes Cryptography* 62.3, p. 289–311 (cf. p. 105).

[Hutter et al. 2013] - Hutter, M. et Schmidt, J.-M. (2013). “The Temperature Side Channel and Heating Fault Attacks”. In : *CARDIS 2013*. Sous la dir. d’Aurélien Francillon et Pankaj Rohatgi. T. 8419. LNCS. Berlin, Germany : Springer, Heidelberg, Germany, p. 219–235 (cf. p. 24, 47).

[Ishai et al. 2003] - Ishai, Y., Sahai, A. et Wagner, D. (2003). “Private Circuits : Securing Hardware against Probing Attacks”. In : *CRYPTO 2003*. Sous la dir. de Dan Boneh. T. 2729. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 463–481 (cf. p. 40).

[Jaffe et al. 2003] - Jaffe, J. M., Kocher, P. et Jun, B. C. (2003). *Balanced cryptographic computational method and apparatus for leak minimizational in smartcards and other cryptosystems*. Google Patents, US Patent 6,510,518 : https://www.google.com/patents/US6510518 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 39).

[Jia et al. 2012] - Jia, Y., Hu, Y., Wang, F. et Wang, H. (2012). “Correlation power analysis of Trivium”. In : *Security and Communication Networks* 5.5, p. 479–484 (cf. p. 38).

[Joule 1841] - Joule, J. P. (1841). “XXXVIII. On the heat evolved by metallic conductors of electricity, and in the cells of a battery during electrolysis”. In : *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 19.124, p. 260–277 (cf. p. 23).

[Journault et al. 2017a] - Journault, A., Standaert, F.-X. et Varici, K. (2017a). “Improving the security and efficiency of block ciphers based on LS-designs”. In : *Des. Codes Cryptography* 82.1-2, p. 495–509 (cf. p. 14).

[Journault et al. 2017b] - Journault, A. et Standaert, F.-X. (2017b). “Very High Order Masking : Efficient Implementation and Security Evaluation”. In : *CHES 2017*. Sous la dir. de Wieland Fischer et Naofumi Homma. T. 10529. LNCS. Taipei, Taiwan : Springer, Heidelberg, Germany, p. 623–643 (cf. p. 40).

[Karaklajic et al. 2013] - Karaklajic, D., Schmidt, J. M. et Verbauwhede, I. (2013). “Hardware Designer’s Guide to Fault Attacks”. In : *IEEE Trans. VLSI Syst.* 21.12, p. 2295–2306 (cf. p. 82).

[KARN et al. 1995] - KARN, P., METZGER, P. et SIMPSON, W. (1995). “The ESP Triple DES Transform”. In : *RFC* 1851, p. 1–11 (cf. p. 11).

[KARRI et al. 2003] - KARRI, R., KUZNETSOV, G. et GOESSEL, M. (2003). “Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers”. In : *CHES 2003*. Sous la dir. de Colin D. WALTER, Çetin Kaya KOÇ et Christof PAAR. T. 2779. LNCS. Cologne, Germany : Springer, Heidelberg, Germany, p. 113–124 (cf. p. 82).

[KAZMI et al. 2017] - KAZMI, A. R., AFZAL, M., AMJAD, M. F., ABBAS, H. et YANG, X. (2017). “Algebraic Side Channel Attack on Trivium and Grain Ciphers”. In : *IEEE Access* 5, p. 23958–23968 (cf. p. 105).

[KELIHER et al. 2007] - KELIHER, L. et SUI, J. (2007). “Exact maximum expected differential and linear probability for two-round Advanced Encryption Standard”. In : *IET Information Security* 1.2, p. 53–57 (cf. p. 128).

[KERCKHOFFS 1883] - KERCKHOFFS, A. (1883). “La cryptographie militaire”. In : *Journal des sciences militaires* vol. IX, p. 5–38 (cf. p. 6).

[KNUDSEN 1995] - KNUDSEN, L. R. (1995). “Truncated and Higher Order Differentials”. In : *FSE’94*. Sous la dir. de Bart PRENEEL. T. 1008. LNCS. Leuven, Belgium : Springer, Heidelberg, Germany, p. 196–211 (cf. p. 130).

[KNUTH 1997] - KNUTH, D. E. (1998). *The art of computer programming, Volume II : Seminumerical Algorithms, 3rd Edition*. Addison-Wesley (cf. p. 10).

[KOCHER 1996] - KOCHER, P. (1996). “Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems”. In : *CRYPTO’96*. Sous la dir. de Neal KOBLITZ. T. 1109. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 104–113 (cf. p. 20, 38, 83).

[KOCHER et al. 1999] - KOCHER, P. C., JAFFE, J. et JUN, B. (1999). “Differential Power Analysis”. In : *CRYPTO’99*. Sous la dir. de Michael J. WIENER. T. 1666. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 388–397 (cf. p. 20, 27, 29, 39).

[KONUMA et al. 2005] - KONUMA, S. et ICHIKAWA, S. (2005). “Design and Evaluation of Hardware Pseudo-Random Number Generator MT19937”. In : *IEICE Transactions* 88-D.12, p. 2876–2879 (cf. p. 143).

[KRAWCZYK et al. 1997] - KRAWCZYK, H., BELLARE, M. et CANETTI, R. (1997). “HMAC : Keyed-Hashing for Message Authentication”. In : *RFC* 2104, p. 1–11 (cf. p. 7).

[Kumar et al. 2017] - Kumar, S. V. D., Patranabis, S., Breier, J., Mukhopadhyay, D., Bhasin, S., Chattopadhyay, A. et Baksi, A. (2017). “A Practical Fault Attack on ARX-Like Ciphers with a Case Study on ChaCha20”. In : *FDTC 2017*. Taipei, Taiwan : IEEE Computer Society, p. 33–40 (cf. p. 81).

[Lac et al. 2016] - Lac, B., Beunardeau, M., Canteaut, A., Fournier, J. J. A. et Sirdey, R. (2016). “A First DFA on PRIDE : From Theory to Practice”. In : *CRiSIS 2016*. Sous la dir. de Frédéric Cuppens, Nora Cuppens, Jean-Louis Lanet et Axel Legay. T. 10158. LNCS. Roscoff, France : Springer, Heidelberg, Germany, p. 214–238 (cf. p. 18, 45).

[Lac et al. 2017] - Lac, B., Canteaut, A., Fournier, J. et Sirdey, R. (2017). “DFA on LS-Designs with a Practical Implementation on SCREAM”. In : *COSADE 2017*. Sous la dir. de Sylvain Guilley. T. 10348. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 223–247 (cf. p. 18, 45, 88).

[Lac et al. 2018] - Lac, B., Canteaut, A., Fournier, J. J. A. et Sirdey, R. (2018). “Thwarting Fault Attacks Against Lightweight Cryptography Using SIMD Instructions”. In : *International Symposium on Circuits and Systems - ISCAS 2018*. Florence, Italie : IEEE Computer Society, p. 1–5 (cf. p. 18, 89).

[Lai 1994] - Lai, X. (1994). “Higher order derivatives and differential cryptanalysis”. In : *Communications and Cryptography : Two Sides of One Tapestry*. Springer, Heidelberg, Germany, p. 227–233 (cf. p. 130).

[Lallemand et al. 2017] - Lallemand, V. et Rasoolzadeh, S. (2017). “Differential Cryptanalysis of 18-Round PRIDE”. In : *INDOCRYPT 2017*. Sous la dir. d’Arpita Patra et Nigel P. Smart. T. 10698. LNCS. Chennai, India : Springer, Heidelberg, Germany, p. 126–146 (cf. p. 35, 110, 112, 142).

[Lashermes et al. 2012] - Lashermes, R., Reymond, G., Dutertre, J. M., Fournier, J., Robisson, B. et Tria, A. (2012). “A DFA on AES Based on the Entropy of Error Distributions”. In : *FDTC 2012*. Sous la dir. de Guido Bertoni et Benedikt Gierlichs. Leuven, Belgium : IEEE Computer Society, p. 34–43 (cf. p. 52, 65).

[Le Bouder et al. 2014] - Le Bouder, H., Thomas, G., Linge, Y. et Tria, A. (2014). “On Fault Injections in Generalized Feistel Networks”. In : *FDTC 2014*. Sous la dir. d’Assia Tria et Dooho Choi. Busan, South Korea : IEEE Computer Society, p. 83–93 (cf. p. 81).

[Leander et al. 2011] - Leander, G., Abdelraheem, M. A., AlKhzaimi, H. et Zenner, E. (2011). “A Cryptanalysis of PRINTcipher : The Invariant Subspace Attack”. In : *CRYPTO 2011*. Sous la dir. de Phillip Rogaway. T. 6841. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 206–221 (cf. p. 122, 130, 142).

[Leander et al. 2015] - Leander, G., Minaud, B. et Ronjom, S. (2015). “A Generic Approach to Invariant Subspace Attacks : Cryptanalysis of Robin, iSCREAM and Zorro”. In : *EUROCRYPT 2015, Part I*. Sous la dir. d’Elisabeth Oswald et Marc Fischlin. T. 9056. LNCS. Sofia, Bulgaria : Springer, Heidelberg, Germany, p. 254–283 (cf. p. 122, 127, 130, 131, 142).

[Li et al. 2005] - Li, H., Markettos, T. et Moore, S. (2005). “Security Evaluation Against Electromagnetic Analysis at Design Time”. In : *CHES 2005*. Sous la dir. de Josyula R. Rao et Berk Sunar. T. 3659. LNCS. Edinburgh, UK : Springer, Heidelberg, Germany, p. 280–292.

[Li et al. 2006] - Li, C.-Y., Chen, J.-S. et Chang, T.-Y. (2006). “A chaos-based pseudo random number generator using timing-based reseeding method”. In : *ISCAS 2006*. Island of Kos, Greece : IEEE Computer Society (cf. p. 143).

[Lim et al. 2005] - Lim, C. H. et Korkishko, T. (2005). “mCrypton - A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors”. In : *WISA 2005*. Sous la dir. de JooSeok Song, Taekyoung Kwon et Moti Yung. T. 3786. LNCS. Jeju Island, Korea : Springer, Heidelberg, Germany, p. 243–258 (cf. p. 14).

[Liskov et al. 2002] - Liskov, M., Rivest, R. L. et Wagner, D. (2002). “Tweakable Block Ciphers”. In : *CRYPTO 2002*. Sous la dir. de Moti Yung. T. 2442. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 31–46 (cf. p. 77).

[Liu 2017] - Liu, M. (2017). “Degree Evaluation of NFSR-Based Cryptosystems”. In : *CRYPTO 2017, Part III*. Sous la dir. de Jonathan Katz et Hovav Shacham. T. 10403. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 227–249 (cf. p. 105).

[Lo et al. 2017] - Lo, O., Buchanan, W. J. et Carson, D. (2017). “Power analysis attacks on the AES-128 S-box using differential power analysis (DPA) and correlation power analysis (CPA)”. In : *Journal of Cyber Security Technology* 1.2, p. 88–107 (cf. p. 31).

[Luo et al. 2015] - Luo, P., Zhang, L., Fei, Y. et Ding, A. A. (2015). “Towards secure cryptographic software implementation against side-channel power analysis attacks”. In : *ASAP 2015*. Toronto, ON, Canada : IEEE Computer Society, p. 144–148 (cf. p. 39).

[Masoomi et al. 2010] - Masoomi, M., Masoumi, M. et Ahmadian, M. (2010). “A practical differential power analysis attack against an FPGA implementation of AES cryptosystem”. In : *2010 International Conference on Information Society*. London, UK : IEEE Computer Society, p. 308–312 (cf. p. 31).

[Matsui 1994] - Matsui, M. (1994). “Linear Cryptanalysis Method for DES Cipher”. In : *EUROCRYPT’93*. Sous la dir. de Tor Helleseth. T. 765. LNCS. Lofthus, Norway : Springer, Heidelberg, Germany, p. 386–397 (cf. p. 114).

[Matsui 1997] - Matsui, M. (1997). “New Block Encryption Algorithm MISTY”. In : *FSE’97*. Sous la dir. d’Eli Biham. T. 1267. LNCS. Haifa, Israel : Springer, Heidelberg, Germany, p. 54–68 (cf. p. 11).

[Maxwell 1873] - Maxwell, J. C. (1873). *A treatise on electricity and magnetism*. Oxford Clarendon Press : https://gallica.bnf.fr/ark:/12148/bpt6k95176j.image (en ligne, dernier accès le 29 octobre 2018) (cf. p. 22).

[Menezes et al. 1997] - A. J. Menezes, P. C. Van Oorschot et Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press (cf. p. 7).

[Messerges et al. 1999] - Messerges, T. S., Dabbish, E. A. et Sloan, R. H. (1999). “Power Analysis Attacks of Modular Exponentiation in Smartcards”. In : *CHES’99*. Sous la dir. de Çetin Kaya Koç et Christof Paar. T. 1717. LNCS. Worcester, Massachusetts, USA : Springer, Heidelberg, Germany, p. 144–157 (cf. p. 83).

[Messerges 2000] - Messerges, T. S. (2000). “Using Second-Order Power Analysis to Attack DPA Resistant Software”. In : *CHES 2000*. Sous la dir. de Çetin Kaya Koç et Christof Paar. T. 1965. LNCS. Worcester, Massachusetts, USA : Springer, Heidelberg, Germany, p. 238–251 (cf. p. 29).

[Messerges 2001] - Messerges, T. S. (2001). “Securing the AES Finalists Against Power Analysis Attacks”. In : *FSE 2000*. Sous la dir. de Bruce Schneier. T. 1978. LNCS. New York, NY, USA : Springer, Heidelberg, Germany, p. 150–164 (cf. p. 40).

[Mestiri et al. 2013] - Mestiri, H., Benhadjyoussef, N., Machhout, M. et Tourki, R. (2013). “A Robust Fault Detection Scheme for the Advanced Encryption Standard”. In : *IJCNIS* 5.6, p. 49–55 (cf. p. 84).

[Moore et al. 2002] - Moore, S., Anderson, R., Cunningham, P., Mullins, R. et Taylor, G. (2002). “Improving Smart Card Security Using Self-Timed Circuits”. In : *ASYNC 2002*. Manchester, UK : IEEE Computer Society, p. 211–218 (cf. p. 82).

[Moro 2014] - Moro, N. (2014). “Security of assembly programs against fault attacks on embedded processors”. Thèse. Université Pierre et Marie Curie - Paris VI, France (cf. p. 50, 51).

[Mounier et al. 2012] - Mounier, B., Ribotta, A.-L., Fournier, J. J. A., Agoyan, M. et Tria, A. (2012). “EM Probes Characterisation for Security Analysis”. In : *Cryptography and Security : From Theory to Applications - Essays Dedicated to Jean-Jacques Quisquater on the Occasion of His 65th Birthday*. Sous la dir. de David Naccache. T. 6805. LNCS. Springer, Heidelberg, Germany, p. 248–264 (cf. p. 22).

[Muijrers et al. 2011] - Muijrers, R. A., Van Woudenberg, J. G. J. et Batina, L. (2011). “RAM : Rapid Alignment Method”. In : *CARDIS 2011*. Sous la dir. d’Emmanuel Prouff. T. 7079. LNCS. Leuven, Belgium : Springer, Heidelberg, Germany, p. 266–282 (cf. p. 26).

[Newell et al. 2013] - Newell, G. R. et Morin, T. (2013). *The Right and Wrong Way to Implement Cryptographic Algorithms in Embedded Electronic Systems*. EDN NETWORK : https://www.edn.com/Home/PrintView?contentItemId=4410267 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 26).

[NIST 1977] - NIST (1977). *Federal Information Processing Standards Publication (FIPS 46). Data Encryption Standard (DES)*. National Institute of Standards & Technology : https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 11).

[NIST 1995] - NIST (1995). *Federal Information Processing Standards Publication (FIPS 180). SECURE HASH STANDARD*. National Institute of Standards & Technology : https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 7).

[NIST 2001] - NIST (2001). *Federal Information Processing Standards Publication (FIPS 197). Advanced Encryption Standard (AES)*. National Institute of Standards & Technology : http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf (en ligne, dernier accès le 29 octobre 2018) (cf. p. 8, 11).

[Nozaki et al. 2001] - Nozaki, H., Motoyama, M., Shimbo, A. et Kawamura, S. (2001). “Implementation of RSA Algorithm Based on RNS Montgomery Multiplication”. In : *CHES 2001*. Sous la dir. de Çetin Kaya Koç, David Naccache et Christof Paar. T. 2162. LNCS. Paris, France : Springer, Heidelberg, Germany, p. 364–376 (cf. p. 51).

[Osvik et al. 2006] - Osvik, D. A., Shamir, A. et Tromer, E. (2006). “Cache Attacks and Countermeasures : The Case of AES”. In : *CT-RSA 2006*. Sous la dir. de David Pointcheval. T. 3860. LNCS. San Jose, CA, USA : Springer, Heidelberg, Germany, p. 1–20 (cf. p. 27, 31).

[Oswald et al. 2013] - Oswald, David et Paar, Christof (2012). “Improving Side-Channel Analysis with Optimal Linear Transforms”. In : *CARDIS 2012*. Sous la dir. de Stefan Mangard. T. 7771. LNCS. Graz, Austria : Springer, Heidelberg, Germany, p. 219–233 (cf. p. 25).

[Paillier 1999] - Paillier, P. (1999). “Public-Key Cryptosystems Based on Composite Degree Residuosity Classes”. In : *EUROCRYPT’99*. Sous la dir. de Jacques Stern. T. 1592. LNCS. Prague, Czech Republic : Springer, Heidelberg, Germany, p. 223–238 (cf. p. 8).

[Patrick et al. 2016] - Patrick, C., Yuce, B., Ghalaty, N. et Schaumont, P. (2016). “Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy”. In : *SAC 2016*. Sous la dir. de Roberto Avanzi et Howard M. Heys. T. 10532. LNCS. St. John’s, NL, Canada : Springer, Heidelberg, Germany, p. 231–244 (cf. p. 87, 88, 125).

[Peeters et al. 2007] - Peeters, E., Standaert, F.-X. et Quisquater, J.-J. (2007). “Power and electromagnetic analysis : Improved model, consequences and comparisons”. In : *Integration* 40.1, p. 52–60 (cf. p. 22, 28).

[Petrank et al. 2000] - Petrank, E. et Rackoff, C. (2000). “CBC MAC for Real-Time Data Sources”. In : *J. Cryptology* 13.3, p. 315–338 (cf. p. 7).

[Piret et al. 2003] - Piret, G. et Quisquater, J.-J. (2003). “A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD”. In : *CHES 2003*. Sous la dir. de Colin D. Walter, Çetin Kaya Koç et Christof Paar. T. 2779. LNCS. Cologne, Germany : Springer, Heidelberg, Germany, p. 77–88 (cf. p. 52).

[Piret et al. 2012] - Piret, G., Roche, T. et Carlet, C. (2012). “PICARO - A Block Cipher Allowing Efficient Higher-Order Side-Channel Resistance”. In : *ACNS 12*. Sous la dir. de Feng Bao, Pierangela Samarati et Jianying Zhou. T. 7341. LNCS. Singapore : Springer, Heidelberg, Germany, p. 311–328 (cf. p. 15, 42).

[Pornin 2001] - Pornin, Thomas (2001). “Implantation et optimisation des primitives cryptographiques”. Thèse. Université Paris Diderot - Paris VII, France (cf. p. 32, 88).

[Preneel 2008] - Preneel, B. (2008). “The State of Hash Functions and the NIST SHA-3 Competition”. In : *Inscrypt 2008*. Sous la dir. de Moti Yung, Peng Liu et Dongdai Lin. T. 5487. LNCS. Beijing, China : Springer, Heidelberg, Germany, p. 1–11 (cf. p. 7).

[Quisquater et al. 2002] - Quisquater, J.-J. et Samyde, D. (2002). “Eddy current for Magnetic Analysis with Active Sensor”. In : *Esmart 2002, Nice, France* (cf. p. 47).

[Rechberger et al. 2004] - Rechberger, C. et Oswald, E. (2004). “Practical Template Attacks”. In : *WISA 04*. Sous la dir. de Chae Hoon Lim et Moti Yung. T. 3325. LNCS. Jeju Island, Korea : Springer, Heidelberg, Germany, p. 440–456 (cf. p. 30).

[Rijmen et al. 1996] - Rijmen, V., Daemen, J., Preneel, B., Bosselaers, A. et De Win, E. (1996). “The Cipher SHARK”. In : *FSE’96*. Sous la dir. de Dieter Gollmann. T. 1039. LNCS. Cambridge, UK : Springer, Heidelberg, Germany, p. 99–111 (cf. p. 11).

[Rius et al. 2003] - Rius, J., Peidro, A., Manich, S. et Rodriguez-Sánchez, R. (2003). “Power and Energy Consumption of CMOS Circuits : Measurement Methods and Experimental Results”. In : *PATMOS 2003*. Sous la dir. de Jorge Juan-Chico et Enrico Macii. T. 2799. LNCS. Torino, Italy : Springer, Heidelberg, Germany, p. 80–89 (cf. p. 22).

[Rivain et al. 2009] - Rivain, M., Prouff, E. et Doget, J. (2009). “Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers”. In : *CHES 2009*. Sous la dir. de Christophe Clavier et Kris Gaj. T. 5747. LNCS. Lausanne, Switzerland : Springer, Heidelberg, Germany, p. 171–188 (cf. p. 39, 40).

[Rivain et al. 2010] - Rivain, M. et Prouff, E. (2010). “Provably Secure Higher-Order Masking of AES”. In : *CHES 2010*. Sous la dir. de Stefan Mangard et François-Xavier Standaert. T. 6225. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 413–427 (cf. p. 40, 41).

[Rivest et al. 1978] - Rivest, R. L., Shamir, A. et Adleman, L. M. (1978). “A Method for Obtaining Digital Signature and Public-Key Cryptosystems”. In : *Communications of the Association for Computing Machinery* 21.2, p. 120–126 (cf. p. 8).

[Rivest 1992] - Rivest, R. L. (1992). “The MD5 Message-Digest Algorithm”. In : *RFC* 1321, p. 1–21 (cf. p. 7).

[Rivest 1995] - Rivest, R. L. (1995). “The RC5 Encryption Algorithm”. In : *FSE’94*. Sous la dir. de Bart Preneel. T. 1008. LNCS. Leuven, Belgium : Springer, Heidelberg, Germany, p. 86–96 (cf. p. 11).

[Rivest et al. 1998] - Rivest, R. L., Robshaw, M. J. B., Sidney, R. et Yin, Y. L. (1998). “The RC6 Block Cipher”. In : *First Advanced Encryption Standard (AES) Conference*, p. 16 (cf. p. 11).

[Rogaway et al. 2004] - Rogaway, P. et Shrimpton, T. (2004). “Cryptographic Hash-Function Basics : Definitions, Implications, and Separations for Preimage Resistance, Second-Preimage Resistance, and Collision Resistance”. In : *FSE 2004*. Sous la dir. de Bimal K. Roy et Willi Meier. T. 3017. LNCS. New Delhi, India : Springer, Heidelberg, Germany, p. 371–388 (cf. p. 7).

[Ronen et al. 2017] - Ronen, E., Shamir, A., Weingarten, A.-O. et O’Flynn, C. (2017). “IoT Goes Nuclear : Creating a ZigBee Chain Reaction”. In : *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA : IEEE Computer Society Press, p. 195–212 (cf. p. 15).

[Roscian et al. 2013] - Roscian, C., Sarafianos, A., Dutertre, J.-M. et Tria, A. (2013). “Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells”. In : *FDTC 2013*. Sous la dir. de Wieland Fischer et Jörn-Marc Schmidt. Los Alamitos, CA, USA : IEEE Computer Society, p. 89–98 (cf. p. 49).

[Schlösser et al. 2012] - Schlösser, A., Nedospasov, D., Krämer, J., Orlic, S. et Seifert, J.-P. (2012). “Simple Photonic Emission Analysis of AES - Photonic Side Channel Analysis for the Rest of Us”. In : *CHES 2012*. Sous la dir. d’Emmanuel Prouff et Patrick Schaumont. T. 7428. LNCS. Leuven, Belgium : Springer, Heidelberg, Germany, p. 41–57 (cf. p. 20, 24).

[Schmidt et al. 2007] - Schmidt, J.-M. et Hutter, M. (2007). “Optical and EM Fault-Attacks on CRT-based RSA : Concrete Results”. In : *Austrochip 2007*. Graz, Austria : Verlag der Technischen Universität Graz (cf. p. 47).

[Schneier 1994] - Schneier, B. (1994). “Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)”. In : *FSE’93*. Sous la dir. de Ross J. Anderson. T. 809. LNCS. Cambridge, UK : Springer, Heidelberg, Germany, p. 191–204 (cf. p. 11).

[Schneier et al. 1999] - Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C. et Ferguson, N. (1999). *The Twofish Encryption Algorithm : A 128-bit Block Cipher*. New York, NY, USA : John Wiley & Sons, Inc. (cf. p. 11).

[Schramm et al. 2006] - Schramm, K. et Paar, C. (2006). “Higher Order Masking of the AES”. In : *CT-RSA 2006*. Sous la dir. de David Pointcheval. T. 3860. LNCS. San Jose, CA, USA : Springer, Heidelberg, Germany, p. 208–225 (cf. p. 40).

[Schwabe et al. 2016] - Schwabe, P. et Stoffelen, K. (2016). “All the AES You Need on Cortex-M3 and M4”. In : *SAC 2016*. Sous la dir. de Roberto Avanzi et Howard M. Heys. T. 10532. LNCS. St. John’s, NL, Canada : Springer, Heidelberg, Germany, p. 180–194 (cf. p. 125, 148).

[Shannon 1949] - Shannon, C. E. (1949). “Communication Theory of Secrecy Systems”. In : *Bell System Technical Journal* 28.4, p. 656–715 (cf. p. 5).

[Shibutani et al. 2011] - Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T. et Shirai, T. (2011). “Piccolo : An Ultra-Lightweight Blockcipher”. In : *CHES 2011*. Sous la dir. de Bart Preneel et Tsuyoshi Takagi. T. 6917. LNCS. Nara, Japan : Springer, Heidelberg, Germany, p. 342–357 (cf. p. 14).

[Shirai et al. 2007] - Shirai, T., Shibutani, K., Akishita, T., Moriai, S. et Iwata, T. (2007). “The 128-Bit Blockcipher CLEFIA (Extended Abstract)”. In : *FSE 2007*. Sous la dir. d’Alex Biryukov. T. 4593. LNCS. Luxembourg, Luxembourg : Springer, Heidelberg, Germany, p. 181–195 (cf. p. 11, 14).

[Skorobogatov et al. 2003] - Skorobogatov, S. P. et Anderson, R. J. (2003). “Optical Fault Induction Attacks”. In : *CHES 2002*. Sous la dir. de Burton S. Kaliski Jr., Çetin Kaya Koç et Christof Paar. T. 2523. LNCS. Redwood Shores, CA, USA : Springer, Heidelberg, Germany, p. 2–12 (cf. p. 49).

[Skorobogatov 2009] - Skorobogatov, S. P. (2009). “Local Heating Attacks on Flash Memory Devices”. In : *HOST 2009*. Sous la dir. de Mohammad Tehranipoor et Jim Plusquellic. San Francisco, CA, USA : IEEE Computer Society, p. 1–6 (cf. p. 47).

[Suzaki et al. 2013] - Suzaki, T., Minematsu, K., Morioka, S. et Kobayashi, E. (2013). “TWINE : A Lightweight Block Cipher for Multiple Platforms”. In : *SAC 2012*. Sous la dir. de Lars R. Knudsen et Huapeng Wu. T. 7707. LNCS. Windsor, Ontario, Canada : Springer, Heidelberg, Germany, p. 339–354 (cf. p. 14).

[Thiebeauld et al. 2018] - Thiebeauld, H., Gagnerot, G., Wurcker, A. et Clavier, C. (2018). “SCATTER : A New Dimension in Side-Channel”. In : *COSADE 2018*. Sous la dir. de Junfeng Fan et Benedikt Gierlichs. T. 10815. LNCS. Singapore : Springer, Heidelberg, Germany, p. 135–152 (cf. p. 26).

[Tiri et al. 2004] - Tiri, K. et Verbauwhede, I. (2004). “A Logic Level Design Methodology for a Secure DPA Resistant ASIC or FPGA Implementation”. In : *DATE 2004*. Paris, France : IEEE Computer Society, p. 246–251 (cf. p. 38).

[Todo 2015] - Todo, Y. (2015). “Structural Evaluation by Generalized Integral Property”. In : *EUROCRYPT 2015, Part I*. Sous la dir. d’Elisabeth Oswald et Marc Fischlin. T. 9056. LNCS. Sofia, Bulgaria : Springer, Heidelberg, Germany, p. 287–314 (cf. p. 130).

[Todo et al. 2016] - Todo, Y., Leander, G. et Sasaki, Y. (2016). “Nonlinear Invariant Attack - Practical Attack on Full SCREAM, iSCREAM, and Midori64”. In : *ASIACRYPT 2016, Part II*. Sous la dir. de Jung Hee Cheon et Tsuyoshi Takagi. T. 10032. LNCS. Hanoi, Vietnam : Springer, Heidelberg, Germany, p. 3–33 (cf. p. 122, 127, 132, 142).

[Todo et al. 2017] - Todo, Y., Isobe, T., Hao, Y. et Meier, W. (2017). “Cube Attacks on Non-Blackbox Polynomials Based on Division Property”. In : *CRYPTO 2017, Part III*. Sous la dir. de Jonathan Katz et Hovav Shacham. T. 10403. LNCS. Santa Barbara, CA, USA : Springer, Heidelberg, Germany, p. 250–279 (cf. p. 105).

[Tsoi et al. 2003] - Tsoi, K. H., Leung, K. H. et Leong, P. H. W. (2003). “Compact FPGA-based True and Pseudo Random Number Generators”. In : *FCCM 2003*. Napa, CA, USA : IEEE Computer Society, p. 51–61 (cf. p. 143).

[Tummeltshammer et al. 2009] - Tummeltshammer, P. et Steininger, A. (2009). “On the role of the power supply as an entry for common cause faults - An experimental analysis”. In : *DDECS 2009*. Liberec, Czech Republic : IEEE Computer Society, p. 152–157 (cf. p. 47).

[Tunstall et al. 2007] - Tunstall, M. et Benoît, O. (2007). “Efficient Use of Random Delays in Embedded Software”. In : *WISTP 2007*. Sous la dir. de Damien Sauveron, Constantinos Markantonakis, Angelos Bilas et Jean-Jacques Quisquater. T. 4462. LNCS. Heraklion, Crete, Greece : Springer, Heidelberg, Germany, p. 27–38 (cf. p. 39).

[Van Der Meulen 2017] - Van Der Meulen, R. (2017). *Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016*. Gartner Press Release : https://www.gartner.com/newsroom/id/3598917 (en ligne, dernier accès le 29 octobre 2018) (cf. p. 3).

[Van Woudenberg et al. 2011a] - Van Woudenberg, J. G. J., Witteman, M. F. et Bakker, B. (2011a). “Improving Differential Power Analysis by Elastic Alignment”. In : *CT-RSA 2011*. Sous la dir. d’Aggelos Kiayias. T. 6558. LNCS. San Francisco, CA, USA : Springer, Heidelberg, Germany, p. 104–119 (cf. p. 26).

[Van Woudenberg et al. 2011b] - Van Woudenberg, J. G. J., Witteman, M. F. et Menarini, F. (2011b). “Practical Optical Fault Injection on Secure Microcontrollers”. In : *FDTC 2011*. Sous la dir. de Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache et Junko Takahashi. Tokyo, Japan : IEEE Computer Society, p. 91–99 (cf. p. 53, 82).

[Verbauwhede et al. 2011] - Verbauwhede, I., Karaklajic, D. et Schmidt, J.-M. (2011). “The Fault Attack Jungle - A Classification Model to Guide You”. In : *FDTC 2011*. Sous la dir. de Luca Breveglieri, Sylvain Guilley, Israel Koren, David Naccache et Junko Takahashi. Tokyo, Japan : IEEE Computer Society, p. 3–8 (cf. p. 49).

[Veyrat-Charvillon et al. 2012] - Veyrat-Charvillon, N., Medwed, M., Kerckhof, S. et Standaert, F.-X. (2012). “Shuffling against Side-Channel Attacks : A Comprehensive Study with Cautionary Note”. In : *ASIACRYPT 2012*. Sous la dir. de Xiaoyun Wang et Kazue Sako. T. 7658. LNCS. Beijing, China : Springer, Heidelberg, Germany, p. 740–757 (cf. p. 40).

[Veyrat-Charvillon et al. 2013] - Veyrat-Charvillon, N., Gérard, B., Renauld, M. et Standaert, F.-X. (2013). “An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks”. In : *SAC 2012*. Sous la dir. de Lars R. Knudsen et Huapeng Wu. T. 7707. LNCS. Windsor, Ontario, Canada : Springer, Heidelberg, Germany, p. 390–406 (cf. p. 28).

[VIEGA et al. 2003] - VIEGA, J. (2003). “Practical Random Number Generation in Software”. In : *ACSAC 2003*. Las Vegas, NV, USA : IEEE Computer Society, p. 129–140 (cf. p. 143).

[VIJAYKUMAR 2012] - VIJAYKUMAR, A. (2012). “DPA Resistance of Cryptographic Circuits Considering Temperature and Process Variations”. Thèse. University of Cincinnati, OH, USA (cf. p. 24).

[WATANABE et al. 2008] - WATANABE, D., IDEGUCHI, K., KITAHARA, J., MUTO, K., FURUICHI, H. et KANEKO, T. (2008). “Enocoro-80 : A Hardware Oriented Stream Cipher”. In : *ARES 2008*. Barcelona , Spain : IEEE Computer Society, p. 1294–1300 (cf. p. 14).

[WU et al. 2011] - WU, W. et ZHANG, L. (2011). “LBlock : A Lightweight Block Cipher”. In : *ACNS 2011*. Sous la dir. de Javier LÓPEZ et Gene TSUDIK. T. 6715. LNCS. Nerja, Spain, p. 327–344 (cf. p. 14).

[YEN et al. 2000] - YEN, S.-M. et JOYE, M. (2000). “Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis”. In : *IEEE Trans. Computers* 49.9, p. 967–970 (cf. p. 50).

[ZHANG et al. 2010] - ZHANG, J., GU, D., GUO, Z. et ZHANG, L. (2010). “Differential power cryptanalysis attacks against PRESENT implementation”. In : *ICACTE 2010*. T. 6. Chengdu, China : IEEE Computer Society, p. V6-61-V6-65 (cf. p. 31).

[ZHANG et al. 2015] - ZHANG, W., BAO, Z., LIN, D., RIJMEN, V., YANG, B. et VERBAUWHEDE, I. (2015). “RECTANGLE : a bit-slice lightweight block cipher suitable for multiple platforms”. In : *SCIENCE CHINA Information Sciences* 58.12, p. 1–15 (cf. p. 14).

[ZUSSA et al. 2013] - ZUSSA, L., DUTERTRE, J.-M., CLÉDIÈRE, J. et TRIA, A. (2013). “Power supply glitch induced faults on FPGA : An in-depth analysis of the injection mechanism”. In : *IOLTS 2013*. Chania, Crete, Greece : IEEE Computer Society, p. 110–115 (cf. p. 46, 49).

École Nationale Supérieure des Mines
de Saint-Étienne

NNT : *Communiqué le jour de la soutenance*

Benjamin LAC

LIGHTWEIGHT CRYPTOGRAPHY INTRINSICALLY RESISTANT TO PHYSICAL ATTACKS FOR THE INTERNET OF THINGS.

Speciality: Microelectronics.

Keywords: Lightweight cryptography · Physical attacks · IRC · GARFIELD.

Abstract:

The Internet of Things has many application areas and offers huge potentials for businesses, industries and users. Our study deals with the cryptographic requirements and the security issues of connected objects, which specificities are the large number of data they handle every day, and the fact they are often fielded in hostile environment, physically accessible to any type of potential attacker.

Side-channel attacks and fault attacks are the two main categories of physical attacks. In our research works, we analyze these different techniques of physical attacks and the existing countermeasures to thwart them, and we introduce two new attack paths that we have experimentally validated in laboratory on a recent family of symmetric encryption schemes: the interleaving structures.

In order to meet the security needs and the high performance constraints of the connected objects, we propose a new generic software countermeasure based on redundancy to thwart most of the physical attacks that we called the IRC. We then study the deployment of the IRC on the existing encryption schemes, and its resistance to physical attacks.

Finally, we introduce GARFIELD: a new lightweight block cipher adapted to the IRC in order to ensure a good compromise between security and performance. We check its resistance to conventional mathematical attacks and we propose several implementations with different security levels, for the applications of the Internet of Things, for which we analyze the resulting performances and the validity in practice.

École Nationale Supérieure des Mines
de Saint-Étienne

NNT : *Communiqué le jour de la soutenance.*

Benjamin LAC

CRYPTOGRAPHIE LÉGÈRE INTRINSÈQUEMENT RÉSISTANTE AUX ATTAQUES PHYSIQUES POUR L'INTERNET DES OBJETS.

Spécialité : Microélectronique.

Mots clefs : Cryptographie légère · Attaques physiques · IRC · GARFIELD.

Résumé :

L'Internet des objets a de nombreux domaines applicatifs et offre ainsi un potentiel immense pour les entreprises, les industries et les utilisateurs. Notre étude porte sur les besoins en cryptographie et les enjeux de sécurité des objets connectés, dont les particularités sont à la fois le nombre important de données qu'ils manipulent, et le fait qu'ils soient souvent en milieu hostile, accessibles physiquement à tout type d'attaquant potentiel.

Les attaques par observation et par perturbation sont les deux principales catégories d'attaques physiques. Dans nos travaux de recherche, nous analysons ces différentes techniques d'attaques et les contre-mesures existantes, et nous introduisons deux nouveaux chemins d'attaques que nous avons validés expérimentalement en laboratoire sur une famille récente de chiffrements symétriques : les structures entrelacées.

Afin de répondre aux besoins en matière de sécurité et aux fortes contraintes de performances des objets connectés, nous proposons une nouvelle contre-mesure logicielle générique basée sur la redondance que nous avons nommée l'IRC. Nous étudions donc le déploiement de l'IRC sur les schémas de chiffrement existants, et sa résistance face aux attaques physiques.

Finalement, nous introduisons GARFIELD : un nouveau chiffrement par blocs à bas coût adapté à l'IRC pour assurer un bon compromis entre sécurité et performance. Nous vérifions sa résistance aux attaques mathématiques classiques et nous proposons des implémentations avec différents niveaux de sécurité face aux attaques physiques, pour les applications de l'Internet des objets, dont nous analysons les performances et la validité en pratique.