Protection des dispositifs de chiffrement symétriques contre les injections multiples de fautes

Pierre-Antoine Tissot

N°d'ordre NNT : **2023STET049**

THÈSE de DOCTORAT
DE L'UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Membre de l'Université de LYON

École Doctorale N° 488
Sciences Ingénierie et Santé

Spécialité de doctorat :
Microélectronique

Soutenue publiquement le 11/12/2023, par :
Pierre-Antoine Tissot

Protection des dispositifs de chiffrement symétriques contre les injections multiples de fautes

---

Devant le jury composé de :

Boura, Christina	MCF-HDR Université de Versailles Saint-Quentin en Yvelines – LMV	Rapporteur.e
Lapôtre, Vianney	MCF-HDR Université de Bretagne Sud – Lab-STICC	Rapporteur.e
Killian, Cedric	Pr. Université Jean Monnet St Etienne – LabHC	Examinateur
Hély, David	Pr. Grenoble-INP LCIS	Examinateur
Le Bouder, Hélène	MCF IMT ATLANTIQUE – Equipe IRISA OCIF	Examinatrice
Bossuet, Lilian	Pr. Université Jean Monnet St Etienne – LabHC	Directeur

Résumé

Les attaques par injection de fautes font partie des principales menaces contre les algorithmes de chiffrement embarqués sur des cibles matérielles. Ces attaques sont basées sur des injections de fautes, réalisées grâce à des menaces physiques, qui vont perturber le fonctionnement du chiffrement. Ces injections, couplées à des analyses sur les perturbations engendrées, peuvent permettre à un adversaire de récupérer des informations sur le système. Plusieurs solutions de défense contre ces attaques existent déjà, se basant principalement sur une redondance appliquée au chiffrement, afin d'empêcher l'adversaire de perturber le fonctionnement sans être détecté. Néanmoins, dans un contexte de cryptographie légère, ces contremesures sont coûteuses à mettre en place. Cette thèse va continuer l'étude de ces contremesures face aux injections de fautes, en se positionnant sur des contremesures appliquées au niveau de la primitive cryptographique, afin de rendre ces solutions généralisables à toutes les implémentations, avec pour trame de fond le surcoût engendré par ces contremesures. En premier lieu, une amélioration d'une contremesure basée sur le principe de respect du code est proposée. Cette contremesure, basée sur un code détecteur d'erreur, est généralisée avec un exemple pour montrer comment elle peut être appliquée à tout algorithme de chiffrement. Ensuite, une méthode de classification des boîtes de substitution (utilisée pour la couche non linéaire des algorithmes de chiffrement par blocs) est introduite. Cette méthode, basée sur des relations d'équivalence, va permettre à un concepteur de primitive cryptographique d'optimiser son choix de S-box, en maintenant de bonnes propriétés cryptographiques tout en réduisant le coût d'implémentation matérielle de la S-box. Enfin, une toute nouvelle contremesure dédiée à contrer les analyses basées sur les injections de fautes persistantes sur la S-box d'un chiffrement est présentée. Cette contremesure, concentrée sur les propriétés de permutation d'une boîte de substitution, permet de détecter 100% des fautes exploitables pour un attaquant cherchant à appliquer une analyse de fautes persistantes proposée dans la littérature. Une proposition d'implémentation physique est également donnée pour limiter le surcoût engendré par cette contremesure tout en maintenant un haut niveau de sécurité.

Abstract

Fault injection attacks are one of the main threats to encryption algorithms embedded in hardware targets. These attacks are based on fault injections, carried out using physical threats, which disrupt the operation of the encryption. These injections, coupled with analyses of the disruptions caused, can enable an adversary to recover information from the system. Several defence solutions against these attacks already exist, based mainly on redundancy applied to the encryption, to prevent the adversary from disrupting the operation without being detected. However, in the context of lightweight cryptography, these countermeasures are costly to implement. This thesis will continue the study of these countermeasures against fault injections, by taking a stand on countermeasures applied at the level of the cryptographic primitive, in order to make these solutions generalizable to all implementations, with a focus on the additional cost generated by these countermeasures. Firstly, an improvement to a countermeasure based on the principle of code abiding is proposed. This countermeasure, based on an error detection code, is generalised with an example to show how it can be applied to any encryption algorithm. Next, a substitution box classification method (used for the non-linear layer of block ciphers) is introduced. This method, based on equivalence relations, will enable a cryptographic primitive designer to optimise his choice of S-box, maintaining good cryptographic properties while reducing the hardware implementation cost of the S-box. Finally, a brand new countermeasure designed to counter analyses based on persistent fault injections on the S-box of an encryption is presented. This countermeasure, which focuses on the permutation properties of a substitution box, enables 100% detection of faults that could be exploited by an attacker seeking to apply a persistent fault analysis proposed in the literature. A physical implementation proposal is also given to limit the extra cost generated by this countermeasure while maintaining a high level of security.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

AES	Advanced Encryption Standard
ALM	Adaptive Logic Module
ALUT	Adaptive Look-Up Table
ANF	Algrebraic Normal Form
BALoo	Bijection Assert with Loops
CARDIS	Smart Card Research and Advanced Application Conference
CHES	Conference on Cryptographic Hardware and Embedded Systems
CPA	Correlation Power Analysis
DDT	Difference Distribution Table
DES	Data Encryption Standard
DFA	Differential Fault Analysis
DPA	Differential Power Analysis
FPGA	Field Programmable Gate Array
GF	Galois Field
IOLTS	International On-Line Testing Symposium
IoT	Internet of Things
LAT	Linear Approximation Table
LE	Logic Element
LED	Lightweight Encryption Device
MDPI	Multidisciplinary Digital Publishing Institute
NESSY	Nano-Electronics for Secure Systems
NIST	National Institute of Standards and Technology
PFA	Persistent Fault Analysis
PGCD	Plus Grand Commun Diviseur
RFID	Radio Frequency Identification
S-box	Substitution Box (ou Boîte de substitution)
SFA	Statistaical Fault Attack
SIFA	Statistical Inefective Fault Attack

Chapitre 1

Introduction

Le développement massif de l'informatique et des objets connectés, avec une hétérogénéité grandissante ainsi qu'une forte augmentation du nombre de données échangées entre des utilisateurs toujours plus nombreux, a amené avec lui plusieurs problématiques liées aux communications entre ces objets connectés. Cet ensemble d'objets connectés est appelé *Internet of Things* (IoT). La théorie des communications modélise les échanges à travers un canal. Cependant, les canaux de communication entre les objets sont, pour la plupart, non protégés. Ainsi, les données sensibles échangées peuvent être compromises, et accessibles à un adversaire qui se place au milieu de l'échange. Une mesure standardisée doit donc être appliquée pour sécuriser ces communications. Ceci est le sujet de la cryptographie.

Cette science souhaite assurer l'authenticité, l'intégrité, la non répudiation et la confidentialité des échanges. L'authenticité faisant référence à assurer la provenance des messages, la confidentialité permet de garder une conversation secrète, la non répudiation, par le biais d'une signature de message, souhaite prouver l'envoi du message par un correspondant et l'intégrité valide un partage qui ne modifie pas les messages originels.

1.1 Chiffrement symétrique

C'est le chiffrement des données qui assure la confidentialité des échanges, et c'est sur cet aspect de la cryptographie que ce manuscrit se concentre. Le chiffrement est l'art de l'écriture secrète : le message chiffré est visible par tout le monde, mais seules les personnes ayant connaissance du secret utilisé peuvent déchiffrer le message. Ainsi, même à travers un canal de communication non sécurisé, un adversaire au milieu de l'échange retrouve le message chiffré sans pouvoir le déchiffrer. Le processus de chiffrement prend en entrée un message et un secret appelé clé, utilise un algorithme permettant de sécuriser le message, et renvoie en sortie un message chiffré, que le destinataire doit déchiffrer, en étant par principe le seul à pouvoir le déchiffrer, à condition qu'il soit le seul à connaître la clé.

En théorie des chiffrements, le secret de la communication est placé dans une clé de chiffrement, sur laquelle repose toute la sécurité de l'échange. Le chiffrement suit donc les principes de Kerckhoffs [Pet83] : « l'adversaire connaît le système ». C'est-à-dire que tout le chiffrement, excepté la clé, est publiquement connu. De plus, le déchiffrement à l'aide d'une clé k' d'un texte chiffré à l'aide d'une clé k ne doit pas révéler d'information sur le texte en clair d'origine si $k \neq k'$. Ainsi, les adversaires souhaitant attaquer le chiffrement ont une parfaite connaissance de l'algorithme en lui-même et des opérations qui le composent, et cherchent donc seulement à deviner la clé utilisée. Ces principes ont permis de développer des chiffrements mathématiquement sûrs.

Deux différentes approches de chiffrement existent dans la littérature.

La première est le chiffrement symétrique : l'émetteur du message et son destinataire partagent la même clé secrète pour le chiffrement et le déchiffrement. Les algorithmes de chiffrement et de déchiffrement sont donc symétriques. Ce type de chiffrement assure la confidentialité ainsi que l'intégrité du chiffrement. En effet, seuls les détenteurs de cette clé secrète peuvent déchiffrer le message (confidentialité), en retrouvant le message d'origine si la communication ou le chiffrement n'ont pas été perturbés (intégrité). Cependant, le chiffrement symétrique ne permet pas de déterminer l'authenticité du message, car toutes les personnes possédant la clé enverront le même message chiffré pour un texte donné.

La seconde approche est le chiffrement asymétrique : chaque intervenant dans une communication possède une paire de clés, une clé publique et une clé privée. La clé publique est, par principe, diffusée et connue de tous quand la clé privée reste secrète. L'émetteur utilise sa clé privée et la clé publique du destinataire afin de chiffrer son message, et le destinataire utilise sa clé privée et la clé publique de l'émetteur pour déchiffrer ce message.

Les algorithmes de chiffrement symétriques sont les plus légers à mettre en place, tout en assurant une sécurité de la communication. De plus, la version matérielle d'un chiffrement symétrique est la plus efficace. Ce sont donc ces chiffrements qui sont le plus souvent embarqués sur des cibles matérielles et qui seront étudiés dans ce manuscrit, puisque dans cette thèse, nous nous intéressons particulièrement aux chiffrements embarqués pour l'IoT avec donc de fortes contraintes de surface et de consommation.

Classiquement, les chiffrements symétriques sont exploités de la façon suivante : un émetteur, appelé Alice, écrit un message en clair noté p (de l'anglais *plaintext*) pour un destinataire, appelé Bob. Alice va utiliser un algorithme de chiffrement ayant pour paramètres d'entrée le message p ainsi qu'une clé secrète k. Cet algorithme va envoyer à Bob un message chiffré noté c (de l'anglais *ciphertext*), qui va à son tour utiliser un algorithme de déchiffrement avec c et k en paramètres, permettant de retrouver p. Ce principe est illustré dans la Figure 1.1. Alice et Bob possèdent tous les deux la clé k qui va permettre de chiffrer p en c et de déchiffrer c en p. L'adversaire Eve, ne connaissant pas k et se trouvant au milieu de la communication, ne peut pas retrouver p à partir de la seule connaissance de c.

Le type de chiffrement symétrique le plus répandu est le chiffrement par blocs. Son utilisation nécessite le découpage des données en blocs de taille fixe. Chacune des fonctions constituant un algorithme de chiffrement par bloc est réversible, permettant ainsi de « remonter » l'algorithme lors du déchiffrement. Au fil du temps, le *National Institute of Standards and Technology* (NIST) a standardisé des algorithmes de chiffrement par blocs comme l'algorithme DES [Sta+77] en 1977, et plus récemment l'algorithme de chiffrement symétrique le plus répandu dans le monde : l'AES [DR02], créé par Joan Daemen et Vincent Rijmen. Ce chiffrement est appliqué sur des blocs de 128 bits et devient le standard de la cryptographie symétrique en 2001.

Avec le développement de l'IoT et les contraintes amenées par les environnements embarqués (faible consommation de courant, petite taille des circuits électroniques, nombre de composants physiques, ...), une nouvelle gamme d'algorithmes de chiffrements dits « légers » a vu le jour. En effet, ces nouveaux chiffrements sont conçus pour

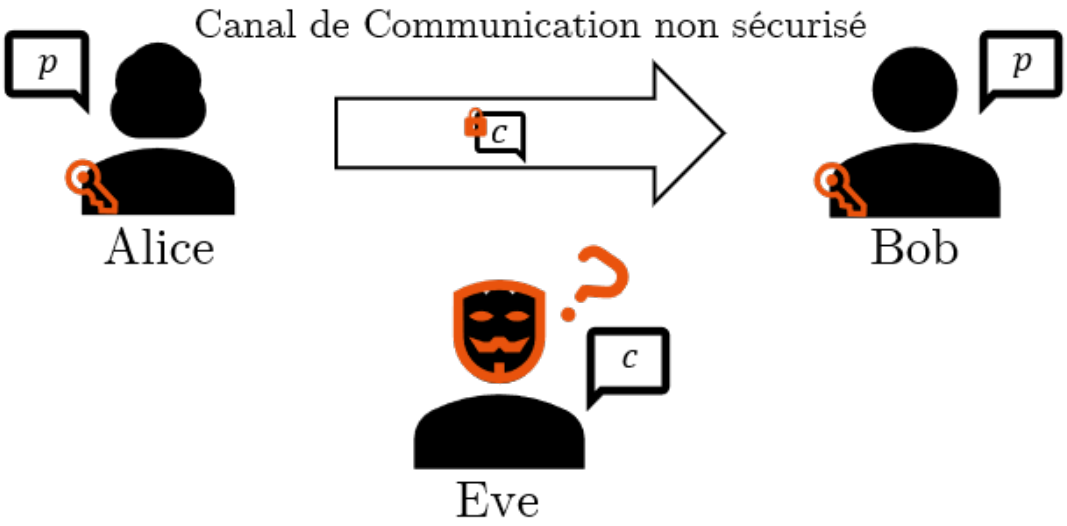

FIGURE 1.1 – Chiffrement symétrique utilisant la clé k pour transformer p et c afin que l'adversaire, sans la connaissance de la clé k, ne puisse pas retrouver p.

être utilisés lorsque les performances actuelles des standards du NIST ne répondent pas aux contraintes liées à l'environnement dans lequel les modules sont embarqués, comme le déploiement de tags RFID ou de réseaux de capteurs. Ces développements de nouveaux algorithmes légers ont résulté sur la standardisation d'un nouvel algorithme en février 2023 : la famille de chiffrement authentifié et de fonction de hachage Ascon [Dob+21]. Un chiffrement authentifié permet d'ajouter l'authentification de l'émetteur même avec un chiffrement symétrique. La fonction de hachage permet la signature du message, afin de vérifier l'intégrité de la transmission.

Malgré leurs propriétés cryptographiques qui rendent les chiffrements mathématiquement sûrs, des attaques qui permettent de retrouver la clé secrète existent. L'attaque la plus simple, mais également la moins efficace, est l'attaque dite par force brute. Le principe est de tester toutes les clés possibles jusqu'à trouver la bonne clé secrète. Néanmoins, avec des clés de taille n bits, les clés forment un ensemble composé de 2^n candidats possibles. Ainsi, l'augmentation des tailles de clé vise à rendre impossible (dans un temps exploitable) cette attaque par force brute. Le travail des chercheurs travaillant sur les attaques contre les implémentations cryptographiques est donc de trouver un moyen de réduire au maximum le nombre de candidats de clé. Plus précisément dans notre cas, c'est l'implantation des algorithmes sur des circuits électroniques qui les rend vulnérables à des attaques visant à identifier la clé. C'est le principe des attaques physiques, et en particulier des attaques par injection de fautes.

1.2 Attaques par injection de fautes

Un objectif de l'adversaire lorsqu'il attaque une communication, sur laquelle est appliqué un chiffrement symétrique, est de trouver la clé secrète afin de pouvoir déchiffrer les messages secrets ou chiffrer ses propres messages en se faisant passer pour un des participants. Ainsi, il va se placer au milieu de la communication pour tenter de trouver des moyens d'obtenir des informations sur la clé secrète. En reprenant la communication entre Alice et Bob, Eve va tenter de retrouver la clé secrète grâce

à une attaque sur le chiffrement mais comme nous l'avons déjà dit, la cryptanalyse mathématique est très difficile. Eve peut alors se déplacer du côté de Alice ou de Bob pour avoir accès au matériel qui exécute le chiffrement et l'attaquer physiquement.

En effet, que ce soit pour une implémentation matérielle ou bien une implémentation logicielle, ce sont en fin de compte des composants électroniques qui réalisent le chiffrement. Ainsi, cette électronique amène avec elle des canaux auxiliaires physiques par lesquels un attaquant peut interagir avec la cible matérielle et récupérer des informations sur le chiffrement, mais elle est aussi très sensible aux perturbations.

Les attaques par injection de fautes utilisent une source d'injection physique afin de perturber le fonctionnement de la cible matérielle, et ainsi observer l'impact de cette perturbation sur le message chiffré produit. Certaines analyses vont utiliser une comparaison entre deux chiffrements issus du même message clair avec et sans injection de faute. Dans la littérature, la plupart des analyses d'impact de fautes actives reposent sur ce principe d'analyse différentielle. Cependant, d'autres méthodes sont également possibles, comme les analyses reposant sur des injections de fautes persistantes, qui seront présentées dans le Chapitre 2. Les attaques par injection de fautes sont divisées en deux sous-groupes : les attaques *invasives* et *semi-invasives*.

Les attaques actives invasives affectent irrémédiablement le circuit (attaque chimiques sur la puce, découpe du circuit au laser, ...) afin de récupérer des informations. Au contraire, les attaques actives semi-invasives affectent le circuit sans le détériorer. Le but ici est d'injecter une perturbation qui va affecter le comportement de l'algorithme de chiffrement afin d'observer un impact sur le message chiffré en sortie. En choisissant précisément quelle injection réaliser et en analysant son impact, l'adversaire va pouvoir récupérer de l'information sur la clé secrète. Ces attaques semi-invasives sont réalisées à l'aide de sources physiques comme des injections laser ou des glitchs d'horloge. Ce principe est illustré sur la Figure 1.2. Dans cette figure, Eve s'est rapprochée de Alice afin d'avoir physiquement accès au dispositif de chiffrement dans l'objectif d'injecter une faute. C'est la méthode différentielle qui est présentée, en comparant le fonctionnement du dispositif avec et sans injection de fautes. Les connaissances couplées sur le message en clair à chiffrer, sur les différents messages chiffrés produits, ainsi que sur l'injection elle-même, permettent de retrouver des parties de la clé.

La première attaque par injection de faute a été présentée en 1997 par Biham et Shamir, avec une analyse différentielle (appelée *Differential Fault Analysis* [BS97]) appliquée sur le standard de l'époque, l'algorithme de chiffrement symétrique DES. Cette attaque permet de retrouver la clé secrète en analysant entre 50 et 200 paires de messages chiffrés. Une paire étant composée d'un message chiffré fauté et d'un message chiffré non fauté, tous deux issus du même message en clair.

Ces injections sont donc combinées à des analyses afin de retrouver de l'information sur le secret du chiffrement. Dépendamment des analyses choisies, l'adversaire peut avoir besoin de maîtriser les textes à chiffrer et ainsi utiliser les informations provenant de l'entrée et de la sortie du chiffrement (c'est le cas dans la DFA et globalement l'ensemble des études différentielles comme deux chiffrements du même message en clair doivent être réalisées), quand d'autres analyses ne nécessitent qu'une lecture du texte chiffré pour retrouver la clé secrète (comme dans l'analyse de fautes persistantes).

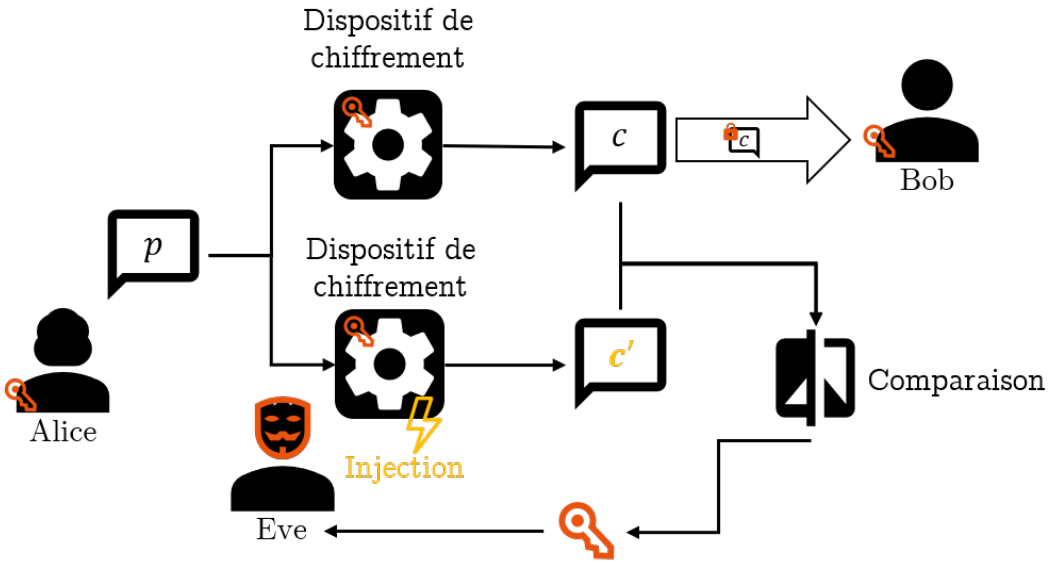

FIGURE 1.2 – Attaque par injection de faute, pour laquelle Eve se rapproche de Alice afin d'avoir physiquement accès au dispositif de chiffrement, couplée à une analyse différentielle entre un chiffré non fauté et un chiffré fauté, tous deux issus du même textes en clair.

1.3 Protection des dispositifs

Il est donc nécessaire de protéger les implémentations des algorithmes de chiffrement symétriques afin de les protéger contre les attaques par injection de fautes. Plusieurs solutions sont possibles. La première est la création d'un bouclier physique, empêchant que la tentative d'injection d'une faute n'impacte le déroulement du chiffrement. Néanmoins, cette contremesure a l'inconvénient de ne pas être généralisable à toutes les implémentations. En effet, chaque circuit électronique aura besoin de son propre bouclier dédié, qui dépendra des caractéristiques techniques et de la technologie de la cible.

Une autre solution, qui est très largement étudiée dans la littérature, est une redondance appliquée au chiffrement et couplée à un vote majoritaire entre les différents chiffrés produits. Cette solution, présentée plus en détail dans le Chapitre 2, est elle aussi dépendante de la cible matérielle choisie. En effet, chaque cible matérielle va stocker et traiter les données différemment, rendant une contremesure matérielle générale difficile à réaliser. C'est pourquoi nous avons préféré étudier des contremesures au niveau de l'algorithme de chiffrement en lui-même, en tentant de trouver des moyens de détecter des injections de fautes en constatant l'impact d'une faute sur la primitive cryptographique elle-même.

Cette approche est similaire à la proposition de masquage des données qui permet de se prémunir des attaques par canaux auxiliaires, en dissociant les fuites perçues par un attaquant et les données calculées. En effet, dans les deux cas, c'est l'algorithme de chiffrement lui-même qui est modifié, afin de contrer des attaques. Ainsi, quelle que soit la cible matérielle choisie, les preuves sur la sécurité des données restent valides. Le masquage ainsi que les protections au niveau de la primitive sont dont généralisables. Notre but est donc de trouver de nouvelles contremesures basées sur ce principe, ou d'améliorer des solutions existantes.

De plus, ces contremesures ont pour but d'être implémentées sur des chiffrements embarqués sur des cibles matérielles. Ainsi, pour respecter les contraintes liées à cette aspect d'implémentation, l'estimation des surcoûts engendrés par ces contremesures sera étudiée. En effet, l'application de différents niveaux de contremesure est un ensemble de compromis. Une contremesure plus efficace sera souvent plus coûteuse et inversement. Optimiser le surcoût permettra de garder un niveau de sécurité très élevé sans surcharger la cible matérielle.

1.4 Contributions

Cette thèse a pour objectif de trouver de nouvelles contremesures face aux attaques physiques par injection de fautes, ou d'améliorer des solutions déjà existantes pour les adapter à de nouvelles problématiques. Chaque solution apportée doit être appliquée au niveau du code afin de détecter une faute injectée, avec une optimisation souhaitée en terme de surcoûts d'implémentation. Les solutions doivent être généralisables à toutes les cibles matérielles et à tous les algorithmes de chiffrement visés.

La première contribution est une **généralisation de la notion de « *respect du code* » applicable à n'importe quel chiffrement par blocs**, déjà existant ou non. En effet, Simon *et al.* ont introduit cette notion en l'intégrant dès la conception à leur fonction de permutation Friet [Sim+20]. Cependant, c'est une contremesure orientée au niveau du bit et non au niveau du bloc. Ainsi, l'utilisation de cette solution sur un chiffrement déjà existant n'est pas triviale. Notre utilisation de cette notion nécessite une conversion de l'algorithme dans sa version *bitslice*, qui va permettre d'appliquer la contremesure au niveau des bits. De plus, une approche plus orientée vers le surcoût engendré par la contremesure permet d'obtenir une solution efficace en limitant son coût d'implémentation. Le Chapitre 3 présente cette généralisation et son application sur le chiffrement léger LED.

Lors de l'adaptation de la fonction `SubBytes` expliquée dans le Chapitre 3, une nouvelle problématique s'est posée : **comment classifier les S-boxes en fonction de leur coût d'implémentation ?** En effet, dans une optique d'optimisation des coûts matériels, le choix de la S-box utilisée pour la couche de substitution est important. Ainsi, le Chapitre 4 présente une relation d'équivalence permettant de classifier les permutations en fonction de leur coût. Cette classification permet de regrouper les permutations ayant les mêmes propriétés cryptographiques ainsi que les mêmes coûts d'implantation, pour aider le concepteur dans son choix. De plus, une méthode est introduite afin d'optimiser le choix d'une S-box en fonction de certains critères d'implémentation.

Enfin, le Chapitre 5 se concentre sur la **première contremesure dédiée à contrer l'analyse des fautes persistantes**. L'analyse présentée dans la littérature est basée sur l'exploitation d'une faute persistante injectée sur une S-box, qui induit une modification dans la distribution de probabilités d'apparition des différents octets possibles dans les textes chiffrés [Zha+18]. Cette solution utilise les propriétés de permutation d'une S-box afin de détecter une faute injectée. La contremesure est ensuite testée sur une implantation FPGA d'un chiffrement AES afin d'évaluer sa robustesse et son surcoût.

Chapitre 2

Notions Préliminaires

Ce manuscrit présente des contremesures appliqués à des chiffrements symétriques par blocs contre les attaques par injection de fautes. Ainsi, ce chapitre va introduire toutes les notions utilisées dans les contributions : les chiffrements symétriques sur lesquels seront appliquées les contremesures, les attaques par injection de fautes face auxquelles les contremesures sont proposées, des exemples de contremesures existantes dans la littérature ainsi que des critères pour estimer le surcoût d'implémentation apporté par les contremesures.

2.1 Chiffrements symétriques

Les chiffrements qui seront utilisés et étudiés dans ces travaux sont les chiffrements symétriques par blocs. Dans le Chapitre 1, le chiffrement symétrique par blocs le plus utilisé, l'AES [DR02], a été présenté. Cet algorithme sera détaillé par la suite. En plus de ce chiffrement, deux autres algorithmes seront également utilisés : l'algorithme léger LED [Guo+11] et la permutation intégrant un code de détection d'erreur FRIET [Sim+20].

2.1.1 Algorithme AES

AES, ou *Advanced Encryption Standard*, est le résultat d'un travail de standardisation des algorithmes de chiffrement symétriques, qui est devenu la référence de la cryptographie symétrique par blocs en 2001 [DR02].

L'algorithme est appliqué sur des blocs de 128 bits, en utilisant des clés de taille 128, 192 ou 256 bits. Le texte à chiffrer est divisé en 16 octets qui sont représentés sous une forme de matrice de 4×4 éléments, notée A. Cette matrice A, dont les éléments sont appelés A_i ($i \in [0, 15]$), est l'état du chiffrement sur lequel vont être appliquées les opérations.

$$A = \begin{pmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{pmatrix}$$

L'algorithme est composé de plusieurs tours, eux-mêmes divisés en quatre opérations : *SubBytes*, *ShiftRows*, *MixColumns* et *AddRoundKey*. En fonction de la taille de clé (128/192/256 bits), le nombre de tours varie ($N_r = 10/12/14$). Les spécifications de l'AES sont illustrées dans la Figure 2.1.

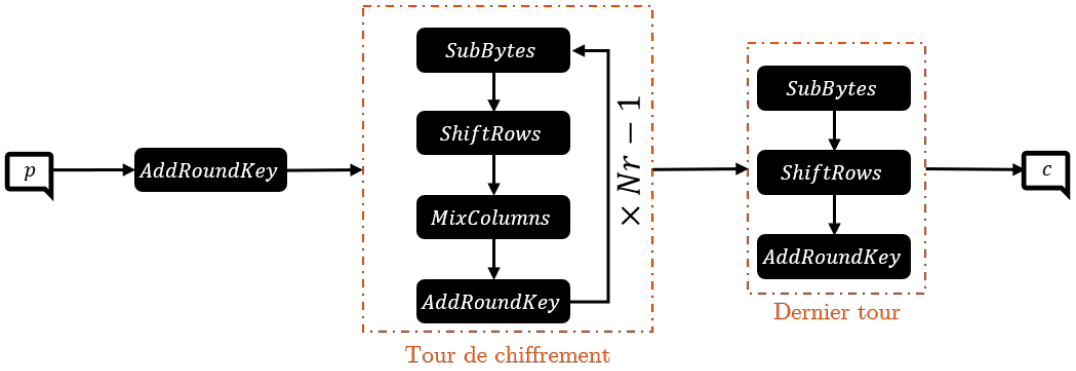

FIGURE 2.1 – Décomposition de l'algorithme AES.

AddRoundKey

Utilisation des sous-clés dérivées directement de la clé maître par un algorithme de cadencement de la clé. Somme binaire de la sous-clé secrète avec l'état actuel du chiffrement. La sous-clé est elle aussi représentée sur une matrice de 4×4 octets, notée K. L'opération réalisée est donc $A \oplus K$. Elle permet d'intégrer la sous-clé dans le message. C'est la seule fonction qui fait intervenir la clé. Le reste des opérations a pour but de mélanger les données pour rendre le déchiffrement impossible sans la connaissance de cette clé.

SubBytes

Application de permutation utilisant une boîte de substitution (ou S-box), notée S, possédant plusieurs bonne propriétés cryptographiques comme la non linéarité ou un degré algébrique précis.

La S-box est présentée sur le Tableau 2.1, en notation hexadécimale (la sortie correspondante à la valeur `0xa4` se trouve à la ligne a0 et à la colonne 04).

TABLE 2.1 – Boîte de substitution de l'AES.

	00	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0a**	**0b**	**0c**	**0d**	**0e**	**0f**
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Le rôle de la fonction de substitution, remplissant la couche non linéaire de l'algorithme, est d'apporter de la confusion dans le chiffrement. Les prochaines fonctions, faisant partie de la couche linéaire de l'algorithme, ont pour but d'amener de la diffusion dans les données.

ShiftRows

Rotation des lignes de la matrice d'état A (avec A_i l'octet i de l'état). La rotation est incrémentée à chaque ligne et illustrée sur la matrice suivante.

$$\begin{pmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_1 & A_5 & A_9 & A_{13} \\ A_2 & A_6 & A_{10} & A_{14} \\ A_3 & A_7 & A_{11} & A_{15} \end{pmatrix} \rightarrow \begin{pmatrix} A_0 & A_4 & A_8 & A_{12} \\ A_5 & A_9 & A_{13} & A_1 \\ A_{10} & A_{14} & A_2 & A_6 \\ A_{15} & A_3 & A_7 & A_{11} \end{pmatrix}$$

MixColumns

Couplée à *ShiftRows*, cette fonction a pour but de diffuser l'information. Ici, la multiplication binaire des colonnes sur le corps de Galois $GF(2^8)$ permet de faire intervenir plusieurs bits d'entrée dans chacun des bits de sortie. Le déchiffrement est donc encore plus compliqué sans connaissance de la clé. La nouvelle colonne de sortie est la multiplication (dans $GF(2^8)$) de la colonne d'entrée par une matrice. Notons C_i les coefficients de la colonne de sortie de la fonction et A_i ses coefficients d'entrée.

$$\begin{pmatrix} C_i \\ C_{i+4} \\ C_{i+8} \\ C_{i+12} \end{pmatrix} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \times \begin{pmatrix} A_i \\ A_{i+4} \\ A_{i+8} \\ A_{i+12} \end{pmatrix}$$

Dans le corps $GF(2^8)$:

- La multiplication par « 1 » est neutre.
- La multiplication par « 2 » est un décalage d'un cran vers la gauche. Si le résultat sort de $GF(2^8)$, alors il est réduit par le polynôme caractéristique $x^8 + x^4 + x^3 + x + 1$.
- La multiplication par « 3 » est une multiplication par « 2 » suivie d'une addition entre la valeur originale et sa multiplication par « 2 ».

KeySchedule

A la fin de chaque tour, la clé suit une expansion qui va permettre de dériver une clé plus longue à partir des 128 bits initiaux. Ainsi, chaque tour utilise une clé différente, chacune d'entre elles issue de la précédente. La fonction d'extension est elle aussi composée d'une partie linéaire, proche de la fonction *MixColumns*, et d'une partie non linéaire, proche de la fonction *SubBytes*. De plus, une constante est ajoutée empêchant les attaques par glissement. Les attaques par glissement, introduites en 1999 par Biryukov et Wagner [BW99], reposent sur le fait que les chiffrements (notamment les chiffrements par blocs) ayant un nombre fixé de tours identiques, n'utilisent que des clés dérivées à chaque tour. Cette similarité des tours, liée au fait que la dérivation peut être manipulée par l'adversaire, qui peut forcer toutes les clés à être identiques par exemple, rend l'algorithme de chiffrement plus faible. Les attaques par glissement utilisent cette faiblesse pour casser le chiffrement.

2.1.2 Algorithme LED

LED (sigle de *Light Encryption Device*), est un algorithme de chiffrement symétrique léger présenté en 2011 à la conférence CHES par Guo *et al* [Guo+11]. Son objectif est d'offrir l'empreinte matérielle la plus petite en comparaison des autres chiffrements par blocs, tout en répondant à trois buts : avoir une expansion de clé très légère (même inexistante), obtenir une bonne résistance face aux attaques à clé liée (en utilisant des preuves dérivées de celles réalisées sur l'AES) et enfin maintenir des performances logicielles raisonnables bien que l'algorithme soit optimisé pour une implémentation matérielle.

LED est un algorithme dont les spécifications sont calquées sur celles de l'AES. Cette section va permettre de détailler les opérations le composant. En effet, elles seront par la suite modifiées dans le Chapitre 3 pour s'adapter à un code détecteur d'erreurs.

LED chiffre des blocs de 64 bits en utilisant des clés entre 64 et 128 bits, tant que la longueur de cette clé est divisible par 4. Ainsi, il est possible de retrouver des implémentations LED avec des clés de longueur 96. Cependant, par soucis de simplification, nous allons ici nous concentrer sur des clés de 64 bits, notées k. La structure de LED est illustrée avec la Figure 2.2.

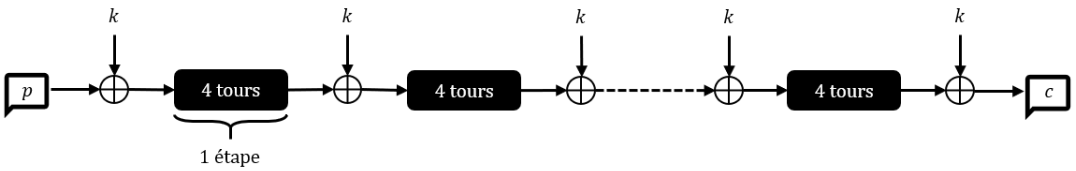

FIGURE 2.2 – Structure de l'algorithme LED.

Le bloc de 64 bits, en cours de chiffrement, est nommé l'état du chiffrement et est représenté par une matrice de 4×4 mots de 4 bits. Les éléments de cette matrice sont notés S_i. Cette représentation est illustrée sur la matrice suivante.

$$\begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ S_4 & S_5 & S_6 & S_7 \\ S_8 & S_9 & S_{10} & S_{11} \\ S_{12} & S_{13} & S_{14} & S_{15} \end{pmatrix}$$

Chaque étape comporte 4 tours composés de 4 fonctions : *AddConstant* , *SubCells*, *ShiftRows* et *MixColumnsSerial*. Entre chaque étape, la clé k est intégrée à l'état du chiffrement grâce à une somme binaire entre les deux matrices d'état et de clé. La décomposition d'un tour est montrée sur la Figure 2.3.

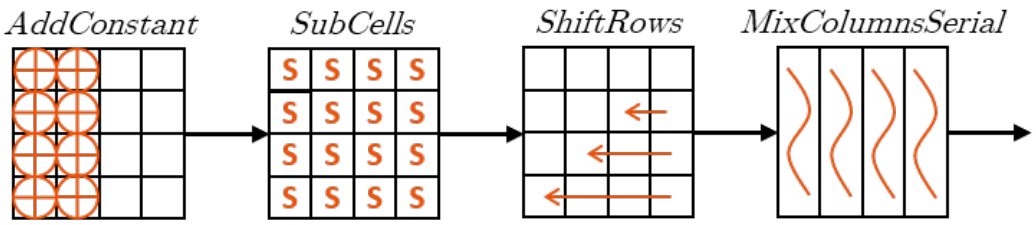

FIGURE 2.3 – Décomposition d'un tour de LED.

Comme elles seront utilisées et modifiées dans le Chapitre 3, ces fonctions seront introduites plus en détails dans la suite du chapitre.

AddConstant

Une constante va être ajoutée à l'état. À chaque tour, six bits dénotés rc_5, rc_4, rc_3, rc_2, rc_1, rc_0 sont décalés d'un cran vers la gauche, et rc_0 prend la valeur $rc_5 \oplus rc_4 \oplus 1$. Ils sont au préalable initialisés à 0 et sont modifiés avant leur utilisation, de sorte que le premier rc_0 utilisé soit égal à 1. De plus, la taille de la clé est aussi injectée dans l'état en notant ks_i le i^{eme} bit de la taille de clé. Ainsi, une constante RC est créée et additionnée dans $GF(2)$ (utilisant une somme binaire) à l'état. La matrice suivante montre la construction de la constante, l'opérateur $\|$ faisant référence à une concaténation.

$$RC = \begin{pmatrix} 0 \oplus (ks_7 \| ks_6 \| ks_5 \| ks_4) & (rc_5 \| rc_4 \| rc_3) & 0 & 0 \\ 1 \oplus (ks_7 \| ks_6 \| ks_5 \| ks_4) & (rc_2 \| rc_1 \| rc_0) & 0 & 0 \\ 2 \oplus (ks_3 \| ks_2 \| ks_1 \| ks_0) & (rc_5 \| rc_4 \| rc_3) & 0 & 0 \\ 3 \oplus (ks_3 \| ks_2 \| ks_1 \| ks_0) & (rc_2 \| rc_1 \| rc_0) & 0 & 0 \end{pmatrix}$$

SubCells

Chaque mot de 4 bits de l'état est substitué en utilisant la même S-box que l'algorithme de chiffrement `PRESENT` [Bog+07a], présentée dans le Tableau 2.2.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

TABLE 2.2 – `PRESENT` S-box.

ShiftRows

Chaque ligne l de l'état est décalée de l positions vers la gauche (avec $l = 0, 1, 2, 3$). Cette rotation est montrée sur la matrice suivante.

$$\begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ S_4 & S_5 & S_6 & S_7 \\ S_8 & S_9 & S_{10} & S_{11} \\ S_{12} & S_{13} & S_{14} & S_{15} \end{pmatrix} \rightarrow \begin{pmatrix} S_0 & S_1 & S_2 & S_3 \\ S_5 & S_6 & S_7 & S_4 \\ S_{10} & S_{11} & S_8 & S_9 \\ S_{15} & S_{12} & S_{13} & S_{14} \end{pmatrix}$$

MixColumnsSerial

Chaque colonne de l'état est remplacée par sa multiplication avec la matrice M suivante (en pratique, elle sera remplacée par le résultat de quatre multiplications par la matrice A).

$$M = \begin{pmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix} = A^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}^4$$

2.1.3 Permutation FRIET

En 2020, Simon *et al.* ont présenté un nouveau schéma de chiffrement authentifié basé sur une nouvelle permutation, nommée FRIET-P [Sim+20]. Le but de cette nouvelle permutation est d'intégrer de la résistance aux injections de fautes dès la conception de la primitive cryptographique. L'objectif est de choisir un code détecteur ou correcteur d'erreur C et d'adapter une permutation en une permutation plus large, qui respecte le code C. Ainsi, une faute injectée à n'importe quelle étape du chiffrement pourra être détectée.

Un tour de la permutation FRIET-P est donné dans la Figure 2.4. Ce tour est divisé en plusieurs opérations, nommées $\delta, \mu_1, \mu_2, \xi$ et τ. Elle prend en entrée les quatre mots de donnée à chiffrer (a, b, c et d), ainsi que deux mots d'une constante rc_i_c et rc_i_d. En sortie de la permutation, on retrouve les quatre mots de données a', b', c' et d' ainsi que les deux mots de la constante qui évolueront pour le prochain tour. L'opérateur Xor est illustré par la porte logique $\oplus$.

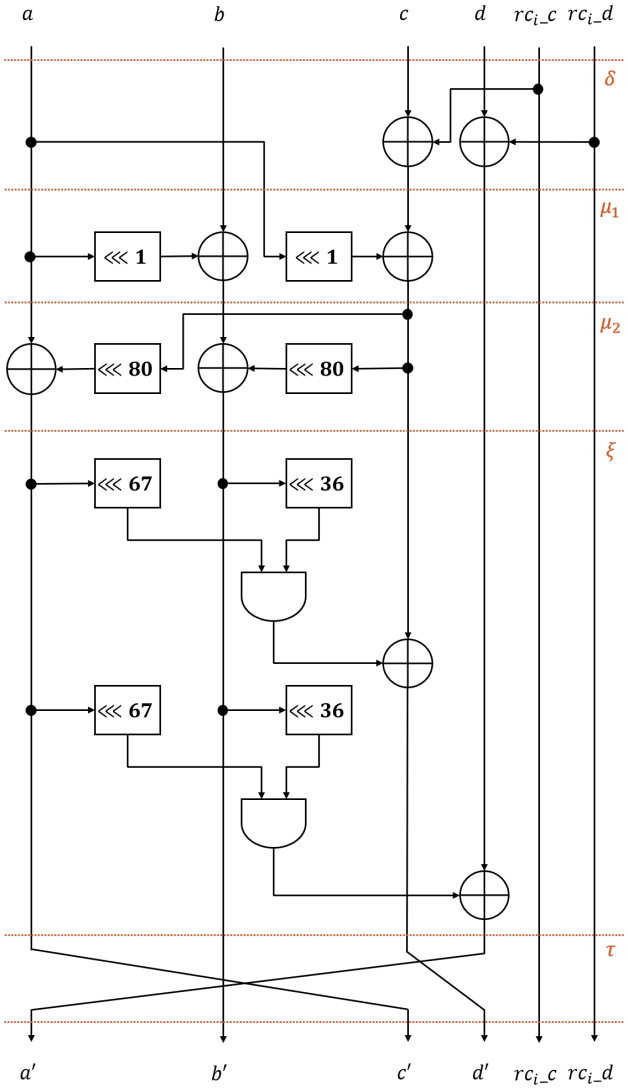

FIGURE 2.4 – Tour de FRIET-P.

Afin de détecter les injections de fautes, une vérification est faite sur la somme binaire des valeurs de sortie de l'opération. Le but ici est donc d'intégrer, dès la conception de ce nouvel algorithme, un moyen de contrôler l'intégrité des données. Un code détecteur est ainsi directement appliqué sur le chiffrement. Le désavantage de cette solution est qu'elle nécessite la création d'une nouvelle primitive cryptographique. Elle n'est donc pas applicable instantanément sur des algorithmes de chiffrement plus standards comme l'AES.

2.1.4 Version bitslice des chiffrements

Une deuxième représentation des chiffrements par blocs sera également intéressante et utilisée dans la suite de ce manuscrit. Cette approche est appelée « par tranches » ou en anglais (et dans la suite du manuscrit) *bitslice*. Ici, l'état originellement composé de plusieurs mots (16 mots de 8 bits pour l'AES et 16 mots de 4 bits pour LED) se retrouve sérialisé en une seule suite de bits (128 bits pour l'AES et 64 bits pour LED). En plus de cette sérialisation, chaque bit est stocké séparément dans un registre mémoire différent. Ainsi, les opérations sont réalisées sur les bits plutôt que sur les mots. Une parallélisation des chiffrements peut également être envisagée. En effet, chaque registre peut contenir un bit donné de plusieurs états chiffrés en parallèle. Comme les opérations sont vues sous leur version bit à bit, le chiffrement peut être appliqué à tout le registre et ainsi plusieurs textes en clair peuvent être chiffrés en parallèle en utilisant une même clé, ou plusieurs clés différentes. Cette version des chiffrements nécessite une adaptation des opérations afin qu'elles correspondent au bon stockage des données. La transformation des registres est illustrée sur la Figure 2.5.

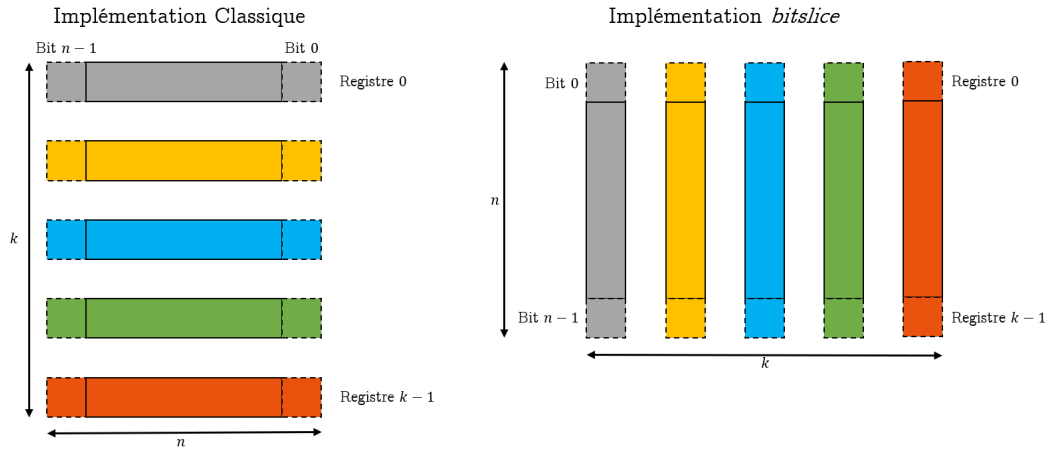

FIGURE 2.5 – Transformation *bitslice* de k états de n bits en parallèle.

Cette représentation offre même parmi les meilleures performances entre toutes les représentations disponibles. En effet, jusqu'à N chiffrements en parallèles sont possibles sur une machine ayant des registres de taille N. Le débit de chiffrement est donc bien supérieur à une implémentation classique. C'est ainsi que les meilleures performances ont été obtenues pour des implémentations DES et AES. Biham a ainsi proposé une implémentation bitslice de l'algorithme DES [Bih97] qui est trois fois plus performante que les implémentations classiques. Afin d'améliorer cette version en termes de nombre de portes logiques nécessaires à l'implémentation des 8 S-boxes composant la partie non-linéaire de DES (une moyenne de 100 est donnée par Biham), Kwan a présenté plusieurs versions de l'implémentation bitslice [Kwa00] qui utilisent différentes configurations de portes logiques : configuration « standard » (portes `AND`,

`OR`, `XOR` et `NOT`) résultant en une moyenne de 56 portes logiques et une configuration « non standard » (portes `NAND`, `NOR`, `NXOR`, `AND-NOT` et `OR-NOT`) qui donne en moyenne 51 portes logiques.

De même, Ribeiro *et al.* ont présenté une implémentation bitslice de l'AES [RSD06] qui était à ce moment la version la plus optimisée de cet algorithme. En effet, les résultats donnés montrent une meilleure performance, qui est principalement due à la parallélisation des chiffrements. En effet, un chiffrement unique est bien moins efficace dans sa version bitslice que dans sa version standard, mais la parallélisation de N chiffrements simultanément permet de compenser ce désavantage afin de devenir la version la plus performante.

De plus, la représentation bitslice d'un algorithme peut également amener de meilleures performances dans des implémentations sécurisées comme dans l'article de Salomon et Levi [SL22]. C'est toujours le principe de parallélisation des chiffrements qui donne ces améliorations.

2.2 Attaques par injection de fautes

Le Chapitre 1 a également présenté les différentes attaques physiques qui peuvent être réalisées sur une implémentation matérielle d'un algorithme de chiffrement. Les attaques qui seront abordées dans ce manuscrit sont les attaques actives semi-invasives : les attaques par injection de fautes. Le but de ce type d'attaque est de perturber le déroulement du chiffrement afin d'étudier l'impact d'une faute injectée dans le processus de cryptographie et ainsi pouvoir acquérir des informations sur la clé secrète. Plusieurs points vont être abordés dans cette section : quelles sont les différentes injections de fautes possibles, quel impact est engendré par les injections et comment étudier cet impact.

2.2.1 Injection d'une faute

L'injection de la faute est l'action de perturber le circuit afin de modifier le fonctionnement de l'algorithme, au niveau de la valeur de ses données ou des calculs à exécuter. Une injection est réalisée à partir d'une menace physique, comme un banc laser [SA03 ; Col+19 ; Men+20], un émetteur d'impulsions électromagnétiques [Pou+11 ; SH] ou un banc de rayons X [Anc+17 ; Mai+22]. L'impact de l'injection sur le chiffrement dépend du type d'injection, de la cible visée et de l'instant auquel la faute est injectée. En effet, un glitch d'horloge peut permettre de simplement sauter une instruction et ainsi changer le message chiffré produit. Les injections laser et électromagnétiques vont avoir pour objectif de modifier des données utilisées, ou de modifier les opérations à effectuer avec ces données [Col+21]. Cependant, en fonction de la technologie de la cible, une injection qui décharge le transistor contenant la valeur du bit visé va vider l'information du transistor et ainsi le replacer dans un état d'initialisation. Sur certaines cibles, l'état initial est la valeur 0, sur d'autres la valeur 1. Ainsi, injecter une faute sur un transistor va forcer à 0 ou à 1 ce bit. Trois modèles de fautes sur une donnée vont donc être différenciés : le *bitset*, le *bitreset* et le *bitflip*. Généralement, un attaquant est forcé par la technologie du dispositif cryptographique à se placer dans le modèle *bitset* ou le modèle *bitreset*. Cependant, pour une meilleure protection contre les injections de fautes, le modèle *bitflip* est également pris en compte. Le modèle *bitset* va inclure toutes les fautes qui ne forcent qu'à 1 la valeur, le modèle *bitreset* celles qui ne forcent qu'à 0 et le modèle *bitflip* les fautes qui inversent l'état d'un bit.

Plus formellement, avec x la donnée ciblée, f la faute injectée et x^f la donnée fautée, les trois modèles sont définis dans la Définition 1 ($\vee$ représente l'opérateur logique `Or`, $\wedge$ l'opérateur logique `And` et $\oplus$ l'opérateur logique `Xor`).

Définition 1 (Impacts d'injection.) *Les trois opérations appliquées aux données ciblées selon le modèle choisi sont les suivantes (avec f un booléen représentant la faute injectée).*

- *Modèle bitset (force les bits à 1) : $x^f = x \vee f$*
- *Modèle bitreset (force les bits à 0) : $x^f = x \wedge f$*
- *Modèle bitflip (0 devient 1 et 1 devient 0) : $x^f = x \oplus f$*

De plus les fautes sont également divisées en trois catégories : les fautes transitoires, les fautes persistantes et les fautes permanentes. Les fautes permanentes renvoient aux attaques actives invasives, faisant référence à des fautes irrémédiables, qui détériorent physiquement et indéfiniment le circuit de chiffrement. Ainsi, ces fautes ne seront pas étudiées dans cette thèse.

2.2.2 Fautes transitoires

Une faute transitoire impacte un circuit précis à un instant précis dans le but de créer une erreur dans le chiffrement à cet instant, pour la propager jusqu'au message chiffré. En effet, le principe de l'analyse exploitant ce type de fautes est de mener une étude différentielle entre les chiffrés produits avec et sans injection de fautes. Ainsi l'attaquant va comparer un message chiffré fauté et un message chiffré non fauté, tous deux issus du même message en clair, et intégrant de l'information sur la même clé. Les connaissances de l'attaquant sur son injection pourront mener à des fuites d'information sur la clé.

La première attaque de Biham *et al.* en 1997 était appliquée sur l'algorithme DES et basée sur une analyse différentielle [BS97] (appelée *Differential Fault Analysis* ou DFA), mais le standard ayant changé, la DFA a ensuite été portée sur l'AES [Gir05]. En effet, une faute aléatoire injectée entre l'avant-dernier *MixColumns* et le dernier *MixColumns* sur un octet défini et connu de l'attaquant permet de diffuser la faute sur 4 octets dans *MixColumns*. Ainsi, 4 fautes différentielles sont provoquées, ont interagi avec la dernière clé et peuvent être analysées. En émettant des hypothèses sur la faute initialement injectée, comme les 4 fautes provoquées proviennent de cette même injection, il est possible de réduire l'ensemble de clés possibles à seulement 128 candidats. En réitérant l'injection d'une faute au même endroit, il est possible de réduire de moitié ce nombre de candidats. De plus, en injectant d'autres fautes sur d'autres colonnes, il est possible de retrouver les autres octets de la clé grâce à environ 20 paires de chiffrés corrects et incorrects. Cette attaque peut également être généralisée à une injection aléatoire sur un emplacement aléatoire. L'injection est toujours faite au même moment, mais la valeur de la faute et l'octet impacté initialement sont inconnus. Dans ce cas, entre 40 et 50 paires sont nécessaires pour retrouver la clé. De même, en injectant cette fois-ci la faute entre l'antépénultième et le pénultième *MixColumns*, seulement 10 paires sont requises pour retrouver la clé.

Ces fautes sont celles qui sont les plus utilisées dans la littérature car les analyses issues de ces injections sont les plus performantes. En effet, nous avons vu qu'une simple DFA appliquée sur l'AES ne nécessite qu'une vingtaines d'injections pour retrouver la clé secrète. Cependant, ces fautes ont le désavantage de la précision. En effet, pour

que l'analyse soit efficace, il faut que la faute n'impacte qu'un octet bien précis à un instant très précis du chiffrement. Ainsi, un travail de synchronisation doit être réalisé en amont. En pratique, les performances maximales de la DFA sont même réalisées en indiquant depuis la cible le moment propice à l'injection de la faute. Ainsi, ce cas de figure ne peut pas être envisageable dans un cas réel d'utilisation. De plus, les attaques différentielles nécessitent un contrôle sur le texte en clair à chiffrer. En effet, afin de comparer le comportement du circuit avec et sans la faute injectée, il faut que les deux chiffrés étudiés soient issus du même texte en clair. Pour palier ces problèmes, un autre type de faute a été étudié : les fautes persistantes.

2.2.3 Fautes persistantes

Les fautes transitoires sont donc à la base des analyses différentielles, car la faute ne perturbe qu'un seul chiffrement. De plus, ces analyses nécessitent de réaliser l'injection au moment du calcul ciblé, ce qui est difficilement réalisable dans certains cas. Au contraire, les fautes persistantes sont injectées dans la mémoire non volatile, et sont donc présentes jusqu'au rafraîchissement de cette mémoire. La faute peut donc être injectée à n'importe quel moment du cycle de vie du circuit. L'injection est donc moins contraignante. L'analyse commence après l'injection. Ainsi, les analyses différentielles ne peuvent pas fonctionner avec ces injections, comme il n'est plus possible de retrouver des messages chiffrés corrects suite à l'injection de la faute (deux chiffrements du même texte en clair donneront deux chiffrés identiques).

Zhang *et al.* ont publié en 2018 une nouvelle analyse (nommée *Persistent Fault Analysis* ou *PFA*) [Zha+18] utilisant des injections de fautes persistantes affectant la valeur de données stockées en mémoire non-volatile d'un FPGA sur lequel était implémenté un AES. Le principe de cette attaque est d'injecter une faute f sur un octet (dont l'index est noté i) de la S-box de l'AES (notée S), et d'effectuer une étude statistique sur les octets des messages chiffrés. En effet, la distribution de probabilité des chiffrés c_j (j désignant l'octet j du texte chiffré), à travers une S-box S non fautée et en intégrant la clé k_j, est présentée sur le premier graphe (à gauche) de la Figure 2.6. On remarque bien que statistiquement, toutes les valeurs de S-box ont autant de probabilités d'apparaître. La distribution de probabilité est donc uniforme pour toutes les valeurs.

Au contraire, le deuxième graphe (à droite) de la Figure 2.6 montre une distribution de probabilités issue d'une S-box S^* dont un octet est fauté, avec donc une valeur apparaissant deux fois plus souvent notée c_j^{max} (valeur de S-box notée v^*) et une autre valeur n'apparaissant plus du tout noté c_j^{min} (valeur de S-box notée v).
L'attaquant va capturer plusieurs sorties c_j de chiffrements et tracer la distribution de probabilités d'apparition des différentes valeurs possibles de c_j. C_j désigne la variable aléatoire de la sortie du chiffrement. L'adversaire peut ainsi utiliser et combiner trois informations qu'il observe suite à la distribution de probabilités :

- $P_r(C_j = c_j) = 0 \Rightarrow c_j = c_j^{min} = v \oplus k_j$
- $P_r(C_j = c_j) = max_{c'_j}(P_r(C_j = c'_j)) \Rightarrow c_j = c_j^{max} = v^* \oplus k_j$
- $0 < P_r(C_j = c_j) < max_{c'_j}(P_r(C_j = c'_j)) \Rightarrow c_j \neq v \oplus k_j$ and $c_j \neq v^* \oplus k_j$

Comme l'analyse se base sur l'étude d'une distribution de probabilités erronée, si une S-box n'est pas fautée (ou si les fautes injectées ne modifient pas la distribution de

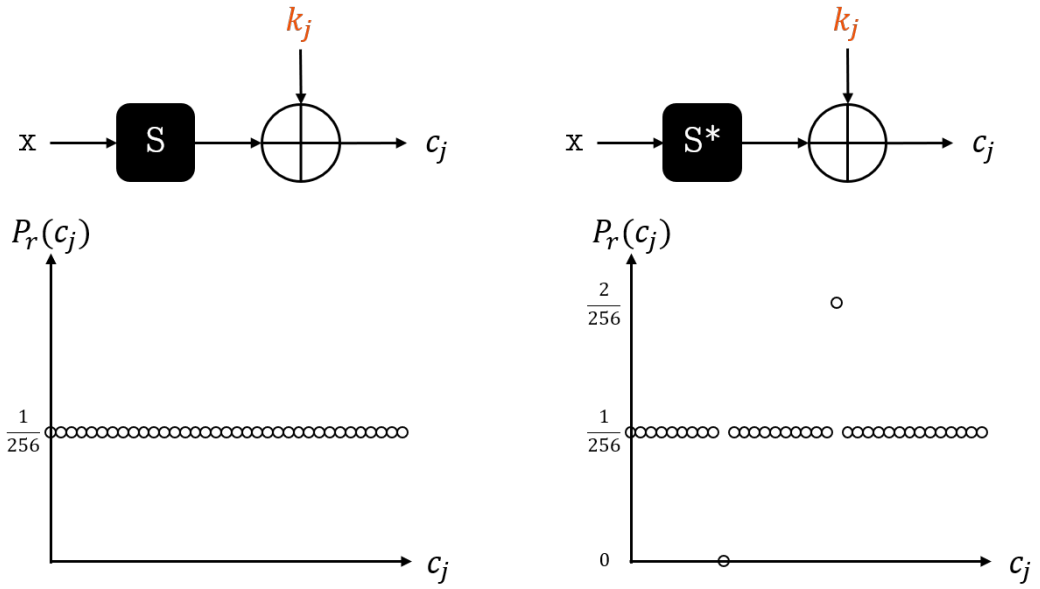

FIGURE 2.6 – Distributions de probabilités de c sans et avec injection.

probabilités), elle n'est pas exploitable par une PFA.

Dans la première analyse de Zhang *et al.*, l'adversaire sait où a été injectée la faute et quelle est la valeur de cette faute. Ainsi, il a connaissance des valeurs v et v^*. Après avoir éliminé tous les c_j qui apparaissent au moins une fois, il peut donc directement utiliser la relation sur les probabilités afin de déterminer l'octet de clé k_j :

$$P_r(C_j = c_j) = 0 \Rightarrow k_j = c_j \oplus v$$

Pour que la distribution soit précise, et donc pour pouvoir trouver avec certitude quel octet n'apparaît plus, environ 2 200 messages chiffrés sont nécessaires. Bien que cette analyse soit moins efficace qu'une analyse différentielle, l'avantage est qu'elle ne demande aucune maîtrise sur le texte en clair à chiffrer. De plus, l'injection peut être réalisée à tout moment alors qu'une faute transitoire nécessite une injection à un instant précis du chiffrement. Ainsi, l'injection est moins contraignante, et l'analyse, bien que moins efficace, est également plus simple à réaliser.

L'analyse est également possible lors d'une injection de multiple fautes. En effet, lorsque plusieurs fautes sont injectées, la distribution de probabilités n'est pas uniforme, et donc la PFA peut être utilisée. La Figure 2.7 montre ces probabilités, avec plusieurs valeurs apparaissant plus souvent et plusieurs valeurs n'apparaissant pas du tout. Notons que sur ce graphe, trois fautes ont été injectées, ainsi trois valeurs n'apparaissent plus. Cependant, même si ici trois valeurs apparaissent deux fois plus, il est également possible que les différentes fautes injectées mènent à une valeur qui apparaîtrait plus que deux fois (si plusieurs valeurs fautées sont égales). Dans tous les cas, l'analyse est toujours possible.

Zhang *et al.* ont présenté l'entropie de la clé selon le nombre de textes chiffrés étudiés et le nombre λ de fautes injectées (présentée sur la Figure 2.8). Cependant, on peut remarquer que plus le nombre de fautes est élevé, moins l'analyse est performante. En effet, plusieurs c_j peuvent être associés à c_j^{min} et à c_j^{max}, et donc les équations basées

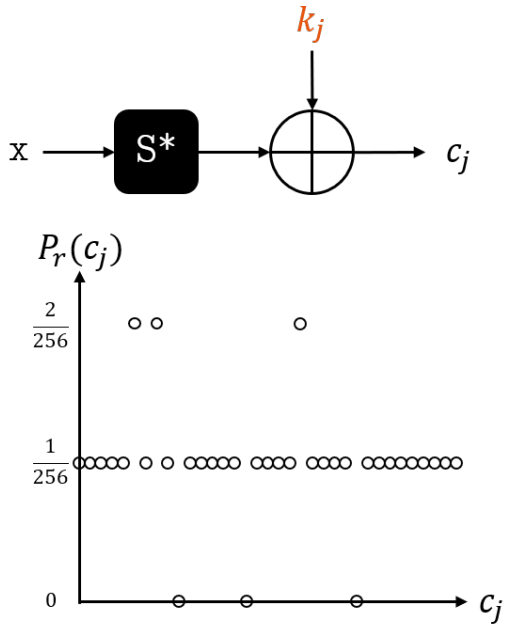

FIGURE 2.7 – Distribution de probabilités lors d'une injection multiple de fautes.

sur les probabilités ne peuvent pas être utilisées.

Plusieurs améliorations de l'analyse ont également vu le jour. En effet, Zhang *et al.* ont présenté plusieurs versions qui allègent les contraintes de l'attaquant [Zha+20]. La première version de l'analyse n'utilisait que c_j^{min} afin de retrouver k_j, et devait donc attendre que toutes les valeurs possible de sorties apparaissent, mais une relation existe entre c_j^{max} et c_j^{min} :

$$c_j^{max} = S^*[i] \oplus k_j = (S^*[i] \oplus S[i]) \oplus (S[i] \oplus k_j) = f \oplus c_j^{min}$$

Ainsi, c_j^{min} peut être retrouvé plus rapidement en utilisant cette relation et ainsi moins de textes chiffrés sont nécessaires pour retrouver k_j (environ 1 600). En effet, lorsqu'il ne reste que quelques candidats finaux pour c_j^{min}, les c_j^{max} reliés à ces candidats sont calculés avec la relation, et ainsi, avec la distribution de probabilité de ces c_j^{max}, la paire de candidat la plus probable regroupe un c_j^{min} de probabilité 0, et un c_j^{max} avec la plus grande probabilité. L'Exemple 1 illustre l'utilisation de cette estimation.

Exemple 1 (Analyses utilisant l'estimation de c_j^{min}) *Soit un ensemble de 1024 octets chiffrés dont le nombre d'apparition est le suivant :*

$$nb(c_j = x) = \begin{cases} 8 & x = 0 \\ 4 & x = 1, 2, \ldots, 253 \\ 0 & x = 254, 255 \end{cases}$$

Ici, deux valeurs sont encore candidates pour être c_j^{min}. En ayant connaissance de la faute injectée (dans notre exemple, $f = 255$), alors nous pouvons utiliser la relation précédente pour retrouver le c_j^{min} le plus probable :

- *Si $c_j^{min} = 254$, alors $c_j^{max} = f \oplus c_j^{min} = 255 \oplus 254 = 1$*
- *Si $c_j^{min} = 255$, alors $c_j^{max} = f \oplus c_j^{min} = 255 \oplus 255 = 0$*

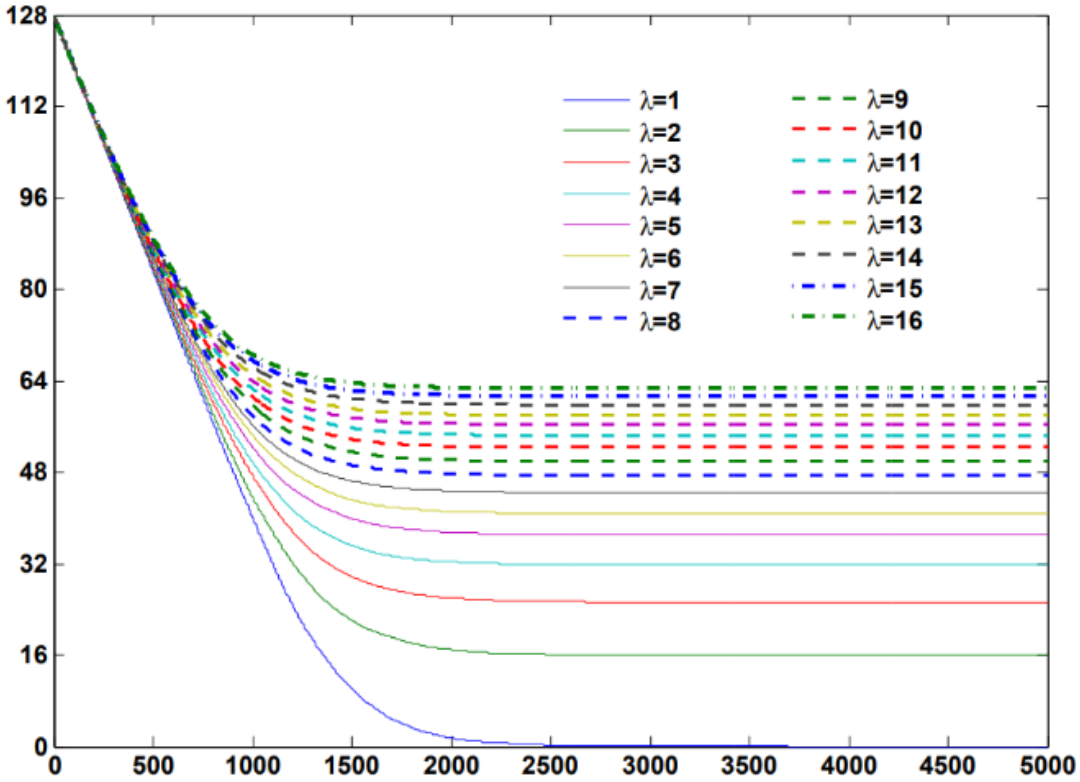

FIGURE 2.8 – Entropie de la clé en fonction du nombre λ de fautes injectées.

Vu notre distribution d'apparition, le candidat le plus probable est donc $c_j^{min} = 255$, *lié au* $c_j^{max} = 0$ *qui est aussi le candidat le plus probable.*

Dans un second temps, l'analyse peut aussi être effectuée sans avoir à connaître la valeur de la faute f, car cette valeur est retrouvable avec environ 600 messages chiffrés. En effet, en émettant des hypothèses sur cette valeur de faute, il est possible de retrouver quelle est la véritable valeur. De même, la localisation i de l'injection de la faute dans la S-box peut également être inconnue, l'analyse pourra se charger de la retrouver en même temps que le calcul de c_j^{min}, car environ 300 messages sont nécessaires pour retrouver i. Ainsi, comme en moyenne 1 600 chiffrés sont nécessaires pour retrouver la clé, les valeurs de f et de i ont largement assez de chiffrements pour être trouvées.

Plus récemment en janvier 2023, Cheng *et al.* ont utilisé un réseau de neurones à apprentissage profond pour appliquer l'analyse des fautes persistantes et ainsi encore faire évoluer cette analyse [Che+23].

2.2.4 Évaluation d'une attaque

Afin de déterminer les performances d'une attaque, ou de comparer plusieurs attaques entre elles, plusieurs métriques peuvent être appliquées. Dans les métriques qui vont être présentées, nous considérons que les hypothèses de clé sont ordonnées, de la plus probable à la moins probable, dans un vecteur de rang. Ce vecteur est donc composé de toutes les hypothèses possibles, avec k^* faisant référence à l'hypothèse juste, donc la sous-clé vraiment utilisée pour le chiffrement.

La première métrique sera le **taux de succès**, donné dans la Définition 2. Pour le calculer, on applique donc l'attaque de nombreuses fois, et on vérifie combien de fois la bonne hypothèse de clé est contenue dans les o premiers indices du vecteur de rang. L'objectif est donc d'avoir un taux de succès le plus élevé possible, pour un ordre donné. Plus cet ordre est bas, plus l'attaque est précise.

Définition 2 (Taux de succès d'ordre o.) *Le taux de succès d'ordre o d'une attaque est la probabilité que la clé réelle k^* soit contenue dans les o premières hypothèses du vecteur de rang.*

La seconde métrique est l'**entropie de rang**, donnée dans la Définition 3. Pour la calculer, on applique donc l'attaque de nombreuses fois, et on mesure à chaque fois la position de la bonne hypothèse dans le vecteur. L'objectif est donc d'avoir une entropie de rang la plus faible possible pour une attaque précise.

Définition 3 (Entropie de rang.) *L'entropie de rang d'une attaque est le rang moyen auquel se trouve la bonne hypothèse de clé k^* dans le vecteur de rang.*

Enfin, le dernier paramètre sur lequel une attaque va être évaluée est le nombre de chiffrements à effectuer pour atteindre un certain score. Par exemple, Zhang *et al.* proposent une analyse de fautes persistantes ayant une entropie de rang de 1 avec 2 000 chiffrements, quand cette entropie passe à 2^{40} pour 1 000 chiffrements.

Ces deux principales métriques vont permettre d'évaluer les attaques, ainsi que les contremesures, qui auront pour objectif de rendre les attaques moins efficaces, et donc de baisser leur score sur ces métriques.

2.3 Contremesures générales

Face aux attaques actives par injection de fautes, plusieurs contremesures peuvent être imaginées. Une première solution est d'ajouter un « bouclier » de protection physique autour de la carte. Ainsi, ce boucler permet d'isoler le chiffrement de toute interaction avec l'extérieur. Bien que très efficace, ce bouclier peut néanmoins être détruit par une attaque chimique et/ou mécanique et donc rendre à nouveau l'implémentation vulnérable aux injections de fautes. Une fois que le bouclier est détruit, toutes les attaques existantes sont à nouveau possibles. De plus, chaque bouclier doit être fabriqué sur mesure en fonction de la taille de la carte, des contraintes physiques liée à l'environnement d'utilisation du chiffrement (un concepteur de chiffrement léger et embarqué ne peut pas se permettre de rajouter une énorme couche de protection autour d'elle). En revanche, dans ce manuscrit, nous souhaitons plutôt intégrer une contremesure appliquée au niveau algorithmique de la primitive cryptographique plutôt que sur la carte physique en elle-même. Ainsi, quelle que soit la cible embarquée choisie, la contremesure sera tout aussi efficace.

Dans la plupart des analyses basées sur une injection de faute, un certain nombre de chiffrés sont nécessaires pour retrouver une clé. Ces textes doivent être chiffrés en utilisant la même clé, et c'est la combinaison des informations récupérées à chaque chiffrement qui permet de retrouver la clé complète. Ainsi, une contremesure souvent utilisée est de changer la clé au bout d'un nombre de chiffrement prédéfini. Plus la clé est changée souvent, moins l'adversaire aura d'information à sa disposition pour retrouver cette clé. Cependant, comme le chiffrement est utilisé lors d'une communication entre plusieurs entités, la nouvelle clé doit à chaque fois être connue de tous,

donc un échange préalable est nécessaire, ce qui implique une complexité accrue de la communication.

Afin de se protéger contre une tentative d'analyse par injection de faute, un moyen efficace est de détecter quand une faute est injectée. Lorsque la détection a lieu, l'algorithme de chiffrement peut ensuite dé-corréler le message chiffré de sortie et la clé de chiffrement. Ainsi, l'adversaire ne pourra pas récupérer d'information directe sur la clé depuis le chiffré. Pour supprimer le lien entre la clé et la sortie, trois principales solutions existent :

— n'envoyer aucune sortie,
— envoyer toujours la même sortie prédéfinie,
— envoyer une sortie aléatoire.

Dans les trois cas, la clé n'intervient plus dans la construction du message de sortie. Ainsi, l'algorithme est protégé contre les analyses directes sur l'injection de la faute. Quelques méthodes de détection sont présentées dans la Section 2.3.1. Cependant, de nouvelles techniques d'analyse ont émergé pour palier le problème de la détection. Ces techniques sont basées sur une étude statistique de l'efficacité des fautes. En effet, si l'adversaire ne reçoit aucune sortie, ou une sortie fixe avec plusieurs entrées différentes, alors il peut considérer que son injection a bien eu un impact. En analysant les cas dans lesquels l'injection a un impact et les cas dans lesquels elle n'en a pas, il peut trouver de l'information sur la clé. C'est le principe des *Statistical Fault Attacks* (*SFA*) [Fuh+13], des *Inefective Fault Analysis* [Cla07] et des *Statistical Inefective Fault Attacks* (*SIFA*) [Dob+18b ; Dob+18a].

Les *SFA*, présentées originellement par Fuhr *et al.* [Fuh+13], ont permis de retrouver la clé d'une implémentation AES. L'attaquant n'a besoin que de savoir si l'injection réalisée a eu un impact (dans ce cas sur l'uniformité de la distribution des valeurs dans le tour 9). L'adversaire utilise un ensemble de chiffrés fautés, qu'il va déchiffrer (avec une hypothèse sur 32 bits de la clé) afin de retrouver cette non-uniformité dans la distribution. En fonction du modèle d'attaque choisi (connaissances de l'attaquant sur la distribution des valeurs au tour 9), les 32 bits de la clé peuvent être retrouvés grâce à un ensemble de messages contenant entre 6 chiffrés (pour l'adversaire ayant le plus de connaissances) et 80 chiffrés (pour l'adversaire ayant le moins de connaissances). Pour cette analyse, les chiffrés doivent être connus, même s'ils sont fautés. En revanche, dans le cas des *SIFA* [Dob+18b ; Dob+18a], le but est justement de retrouver la clé en ayant seulement accès aux chiffrés corrects. Cette fois-ci, l'attaquant sachant que son injection a induit une perturbation dans la distribution des valeurs au tour 9, les hypothèses de clé menant à une distribution biaisée depuis une sortie correcte sont dont réfutées.

Les deux contremesures qui seront présentées dans ce manuscrit auront pour objectif de détecter l'impact d'une injection de faute dans le chiffrement. Les décisions prises après la détection sont hors contexte.

2.3.1 Redondance

La solution classiquement utilisée pour détecter les injections de fautes est la redondance. En effet, une redondance appliquée à un chiffrement avec un vote final permet souvent de détecter si une faute a été injectée ou non. Une redondance est le fait de

chercher à ajouter une duplication d'une donnée afin de vérifier si une donnée a été impactée par une faute quand sa copie est intacte. Ces redondances peuvent être divisées en trois catégories : la redondance spatiale (Définition 4), la redondance temporelle (Définition 5) et la redondance d'information (Définition 6).

Définition 4 (Redondance spatiale) *Solution basée sur le principe que le chiffrement est dupliqué dans l'espace : circuit physique dupliqué, données en mémoire dupliquées. Ainsi, plusieurs chiffrements sont appliqués en parallèle, et les résultats sont comparés. Si tous les résultats sont égaux, la contremesure détermine qu'aucune faute n'a été injectée, et si les résultats diffèrent, une erreur est détectée.*

Cette solution est illustrée avec la Figure 2.9, où un même texte en clair est chiffré sur deux implémentations en parallèle, dont les résultats sont comparés. Ce genre de solution peut être contournée en injectant une faute sur chacune des implémentations dupliquées. Ainsi, la même erreur est appliquée sur tous les états en parallèle, et tous les textes chiffrés seront identiques.

Néanmoins, injecter plusieurs fautes sur un même circuit, mais à plusieurs endroits différents nécessite plusieurs sources qui injectent en parallèle. Ainsi, cela rajoute de fortes contraintes pour l'adversaire. Cependant, de telles attaques ont déjà prouvé leur efficacité par le passé, avec des injections multiples de fautes [Col+21]. De plus, il est possible d'attaquer le système de vote en parallèle, qui est souvent stocké sur un seul bit d'information, et qui dont peut être attaqué au même titre que l'algorithme de chiffrement. Ainsi, il est possible de tromper le système de vote afin de considérer un chiffré fauté comme correct.

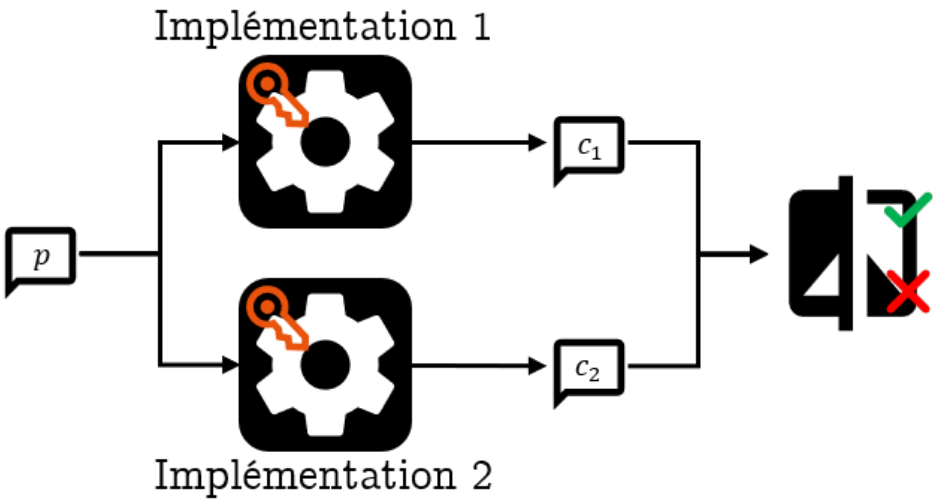

FIGURE 2.9 – Redondance spatiale appliquée à un chiffrement.

Définition 5 (Redondance temporelle) *La redondance temporelle appliquée à un algorithme de chiffrement va chiffrer le même texte en clair plusieurs fois au cours du temps, en utilisant le même circuit physique de chiffrement. La comparaison des textes chiffrés résultants permet d'établir sur la présence d'une erreur. Une erreur dans le chiffrement appliquant directement une modification du chiffré en sortie, cette redondance permet de détecter une faute injectée.*

Cette solution est illustrée avec la Figure 2.10, où un même texte en clair est chiffré deux fois par la même implémentation et les résultats sont comparés. Le premier chiffrement est réalisé à l'instant t_1 et le second à l'instant t_2. De même que pour la redondance spatiale, la faute peut également être injectée à chaque itération du chiffrement, et ainsi tous les textes chiffrés sont fautés, et l'erreur n'est pas détectée [End+14]. Ces injections nécessitent une synchronisation parfaite pour affecter tous les chiffrements successifs de la même manière, et ainsi obtenir le même message chiffré. De plus, comme pour la redondance spatiale, il est possible de fauter le bit stockant le vote majoritaire, et ainsi tromper la contremesure.

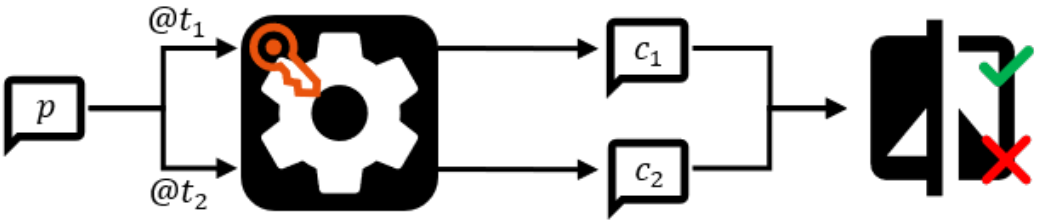

FIGURE 2.10 – Redondance temporelle appliquée à un chiffrement.

Ces deux redondances spatiale et temporelle peuvent également être appliquées à un niveau plus bas du chiffrement, en dupliquant (une ou plusieurs fois) les instructions réalisées directement les unes à la suite des autres [Bar+10]. Dans le cadre de la redondance spatiale, il est plus difficile pour l'adversaire d'injecter plusieurs fautes sur un espace très rapproché, même avec un laser multi-spots. De même dans le cadre d'une redondance temporelle, injecter deux fautes sur un délai très court est beaucoup plus compliqué. Néanmoins dans les deux cas, une modification complète de l'algorithme ou de son implémentation doit être effectuée. En effet, toutes les instructions doivent être dupliquées soit physiquement, en plaçant les mêmes composants sur un espace très restreint, soit de façon algorithmique en intégrant des nouvelles instructions copies entre les instructions du chiffrement. La contremesure est donc plus compliquée à mettre en place qu'une simple redondance.

Définition 6 (Redondance d'information) *L'objectif est de rajouter de l'information de contrôle aux données manipulées afin de vérifier la présence d'une erreur. Le texte en clair n'est chiffré qu'une seule fois, les données redondantes permettant de révéler si une faute a été injectée. L'impact de la faute doit induire une perturbation entre les données redondantes réelles et théoriques.*

Cette solution est illustrée avec la Figure 2.11, où une information supplémentaire (ici en violet) est ajoutée au texte en clair. Cette information redondante est transformée par le chiffrement (devenue rose) et vérifiée pour supposer de l'injection. L'information redondante peut soit être calculée en amont du chiffrement, et ainsi être comparée avec les données redondantes issues de l'état actuel du chiffrement, ou accompagner le message à chiffrer en adaptant l'algorithme de chiffrement. En fonction de l'information redondante choisie, l'attaquant peut tenter de contourner la contremesure en injectant une erreur précise à un endroit défini pour tromper le contrôle. Par exemple, il peut injecter une faute dans l'information calculée en amont afin qu'elle corresponde à un état également fauté. De même, il peut affecter l'état et son information redondante afin que plusieurs injections se compensent et ne soient pas détectées. Comme c'est cette méthode de redondance qui va être utilisée dans les deux contremesures de ce manuscrit, nous allons détailler quelques exemples dans la Section 2.3.2.

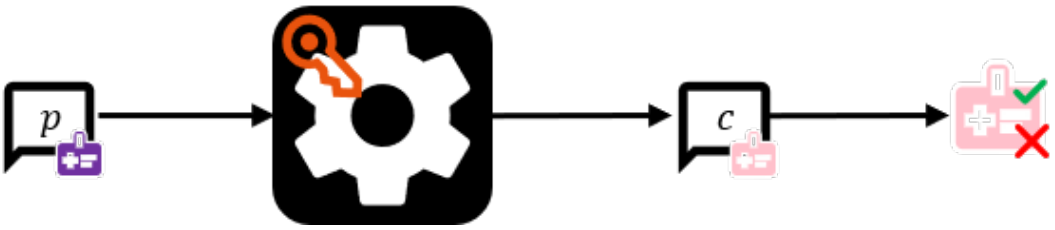

FIGURE 2.11 – Redondance d'information appliquée à un chiffrement.

2.3.2 Redondance d'information

Plusieurs informations peuvent faire office de redondance lors d'un chiffrement. La première solution est d'ajouter un ou plusieurs bits de parité sur les mots chiffrés.

Bit de Parité

La Définition 7 présente le calcul du bit de parité, en mode pair, dans laquelle l'opérateur || fait référence à une concaténation.

Définition 7 (Bit de parité, mode *pair*) *Bit ajouté à un mot qui détermine si le nombre de bits à 1 du mot est pair (bit de parité à 0) ou impair (bit de parité à 1). Le mode impair change juste la valeur du bit de parité* $1 \leftrightarrow 0$.
Plus formellement, soit un mot x *sur* n *bits* $x = x_{n-1}||x_{n-2}||\dots||x_0$, *alors le bit de parité* $x_p = x_{n-1} \oplus x_{n-2} \oplus \cdots \oplus x_0$.
Le mot complet intégrant le bit de parité, noté x', *prend en compte tous les bits ainsi que le bit de parité, et doit vérifier la propriété suivant :* $x_{n-1} \oplus x_{n-2} \oplus \cdots \oplus x_0 \oplus x_p = 0$

L'Exemple 2 illustre le calcul du bit de parité dans le mode pair, et montre comment une faute injectée peut être détectée.

Exemple 2 (Bit de parité) $x = 10101011$
- $x_p = 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 0 \oplus 1 \oplus 1 = 1$
- *Mot complet avec bit de parité :* $x' = 101010111$. *La somme binaire fait bien 0.*
- *Mot complet fauté :* $x'^* = 101000111$. *La somme binaire fait 1 donc erreur détectée.*

Ce bit de parité a été utilisé en temps que contremesure face à des injectons de fautes dans [Ber+02] sur une implémentation AES. Dans ce papier, les auteurs ont créé une matrice qui regroupe tous les bits de parités des différents mots composant l'état du chiffrement. Les fonctions composant le chiffrement AES sont modifiées pour s'adapter à cette matrice de parité. Par exemple, la fonction *SubBytes* n'utilise plus une S-box de 256 valeurs sur 8 bits mais une S-box de 512 valeurs sur 9 bits. Chaque entrée sur 8 bits, à laquelle est ajoutée le bit de parité, est associée à une sortie sur 9 bits qui correspond à la sortie sur 8 bits originale à laquelle est ajoutée le bon bit de parité. Ainsi, la moitié de la S-box correspond aux « bonnes valeurs », et l'autre moitié est remplie de la valeur 000000001. La moitié est donc fausse et remplie de la même valeur. Cela permet de différencier les valeurs non fautées et les valeurs fautées. Les auteurs ont décidé de ne pas garder les propriétés de bijectivité de la représentation avec un bit de plus de cette S-box, en renvoyant toutes les valeurs fautées à la même valeur.

De même, les autres fonctions sont adaptées à la matrice de parité afin de tenir compte de cette propriété.

Redondance Intra-Instruction

Une autre information ajoutée peut être des valeurs de référence, dont le chiffré est connu, et ainsi vérifier l'intégrité du message en vérifiant la valeur finale de référence. C'est ce principe qui est utilisé par Lac *et al.* [Lac+17]. Afin de chiffrer un mot de 8 bits, les auteurs ont décidé de dupliquer ce mot, ainsi qu'une valeur de référence et sa duplication. Ainsi, chaque mot de 8 bits est intégré à un bloc de 32 bits, qui est chiffré et en fin de chiffrement, les duplicata sont vérifiés, ainsi que la valeur de référence. Une autre manière d'utiliser cette redondance interne est d'avoir trois copies de 8 bits de la donnée à chiffrer, ainsi que deux blocs de référence de 4 bits chacun. Ces deux solutions sont illustrées sur la Figure 2.12. La deuxième solution peut même amener à une correction d'erreur. En effet, avec les 3 copies, si deux copies ont une valeur et la troisième une autre valeur, alors la contremesure peut indiquer qu'une faute a été injectée et que la valeur non fautée est celle qui est en double. C'est le principe du vote majoritaire entre plusieurs sorties. La valeur présente sur le plus de sorties est considérée comme correcte. Ce processus peut être appliqué à un plus grand nombre de copies, face à k injections : au minimum $k + 1$ copies pour la détection, et au minimum $2k + 1$ copies pour la correction.

FIGURE 2.12 – Construction du bloc de 32 bits dans le cadre de la redondance interne [Lac+17].

Pour contourner cette contremesure, un attaquant doit injecter la même faute sur les différents duplicata de la donnée, sans intervenir sur les blocs de référence. Ainsi, pour encore plus brouiller les pistes, les auteurs ont également ajouté un mélange aléatoire des différentes parties du mot pour éviter ce cas d'injection.

L'avantage de cette méthode est la sécurité ajoutée au chiffrement, rendant l'injection quasi toujours détectable. Cependant, en contrepartie, la taille des données est multipliée par 4 ici. En effet, chaque mot de 8 bits doit être intégré à un bloc de 32 bits, et c'est ce bloc qui est chiffré. De plus, les blocs de référence sont ensuite comparés à une valeur pré-établie qui doit également être stockée. Un adversaire peut donc chercher à injecter des fautes sur le bloc de référence ainsi que sur la valeur finale à comparer.

Une autre référence de littérature utilisant de la redondance intra-instruction est donnée par Patrick *et al.* [Pat+17]. Leur méthode est proche de celle proposée par Lac *et al.*, mais cette fois-ci en utilisant une version bitslice de leur chiffrement. Dans ce cas, chaque registre est composé de 15 bits de données (en version bitslice, ces bits sont ceux du même indice de 15 états en parallèle), de 15 bits de redondance, liés aux bits de données, et de deux bits de référence. Cette redondance intra-instruction permet de détecter des fautes injectées sur les registres grâce aux bits de redondance, ainsi que des sauts d'instructions, avec les états de référence, dont le chiffré est connu préalablement.

2.3.3 Avantages et inconvénients des redondances

L'avantage des deux premières redondances (temporelle et spatiale) est que les primitives cryptographiques n'ont pas à être modifiées pour tenir compte de la redondance (dans le cas d'une redondance classique et non d'une redondance interne). En effet, les textes en clair sont simplement chiffrés plusieurs fois (en utilisant le même circuit à

plusieurs instants, ou en dupliquant le circuit). Cependant, la redondance spatiale induit un fort surcoût en termes de surface, car le circuit et tous ses composants doivent être implémentés plusieurs fois (une fois par redondance).

La redondance temporelle induit un fort surcoût pour la rapidité de l'implémentation. En effet, l'utilisateur doit attendre que toutes les itérations de chiffrement soient effectuées ainsi que la comparaison entre les chiffrés résultants pour ensuite obtenir son message chiffré.

Enfin, l'avantage de la redondance d'information est que l'information est ajoutée au niveau du code, et qu'elle peut donc être appliquée à toutes les implémentations, quelle que soit la cible matérielle ou logicielle. C'est pourquoi nous avons décidé de développer des contremesures utilisant cette redondance d'information. Néanmoins, les surcoûts engendrés sont dépendants de l'information redondante choisie, ainsi il est plus difficile de les estimer avec une implémentation de redondance d'information. Notre rôle est donc d'optimiser les contremesures afin de réduire le surcoût matériel et la latence induits.

2.4 Coût d'implémentation d'un chiffrement

Un chiffrement embarqué sur une cible matérielle amène avec lui différents coûts d'implémentation. Nous allons différencier les coûts logiciels et matériels. Le but de cette estimation est de déterminer le surcoût engendré par un ajout de contremesure en comparaison à une implémentation simple du chiffrement.

2.4.1 Coûts d'implémentation logiciels

Dans un premier temps, les coûts logiciels sont amenés par une implémentation logicielle du chiffrement. Ainsi, plusieurs valeurs sont mesurées afin d'estimer ces coûts.

Le premier indicateur est le nombre d'appels mémoire réalisés par une exécution du chiffrement. Les performances de l'algorithme sont liées à ces appels mémoire : plus le nombre d'appels est important, moins l'algorithme est performant.

Le second indicateur est le temps d'exécution du programme. Cette mesure permet de déterminer les performances de l'algorithme. Cependant, le temps d'exécution est lié à plusieurs facteurs. Le premier est évidemment le matériel sur lequel est implémenté le chiffrement. Bien que nous soyons dans le cadre d'une implémentation logicielle, les performances du chiffrement vont dépendre des caractéristiques du processeur utilisé, de la technologie mémoire en place, etc... Ainsi, pour être le plus objectif possible, les différents temps d'exécution seront donnés en comparaison les uns des autres. Sous forme de ratio, toutes les mesures ont la même base, et la comparaison est donc plus pertinente.

2.4.2 Coûts d'implémentation matériels

En parallèle de l'évaluation d'une implémentation logicielle, le coût d'une implémentation matérielle peut aussi être mesuré. En effet, dans certaines conditions d'environnement précis, un chiffrement intégré à une cible matérielle nécessite encore plus d'optimisation pour réduire son impact.

Pour ces travaux, les indicateurs sont mesurés lors d'une implémentation sur FPGA d'un chiffrement. Un FPGA, ou *Field Programmable Gate Array*, est un circuit standard dont les éléments logiques peuvent être reconfigurés afin de pouvoir intégrer plusieurs applications différentes au cours de la vie du circuit. Bien que moins performantes que sur des circuits dédiés, les implémentations faites sur FPGA ont l'avantage de la souplesse et de l'universalité de l'implémentation. En effet, en théorie, tous les FPGA disponibles sur le marché peuvent intégrer la même application. La synthèse permet d'assembler les éléments logiques du FPGA afin de créer les fonctions composant l'application. Ainsi, le développement initial est bien moins coûteux que pour un ASIC, c'est seulement la production de nombreux circuits physiques qui va jouer en la faveur des circuits dédiés. Dans notre cas, le but est simplement d'évaluer les surcoûts engendrés par l'ajout de contremesures, ainsi, nous n'allons pas fabriquer des centaines de milliers de circuits. Ainsi, la solution du FPGA est la plus performante, la plus souple et la moins coûteuse.

La première mesure est le nombre d'éléments logiques nécessaires à l'implémentation du chiffrement. En effet, sur du matériel, les fonctions sont implémentées grâce à de la logique, combinatoire et séquentielle. En prenant en compte le fait que nos implémentation matérielles seront uniquement réalisées sur FPGA, ce sera le nombre de LUT (de l'anglais *Look-up Tables*) qui sera compté, et plus particulièrement le nombre d'entrées/sorties de ces tables.

Le second indicateur qui sera utilisé pour évaluer le coût d'une implémentation matérielle est la taille mémoire nécessaire au stockage des informations liées au chiffrement. En effet, plus une implémentation a besoin d'espace mémoire, plus elle va prendre de place dans l'espace total disponible sur une carte. Ainsi, les autres applications embarquées auront moins de place mémoire pour elles. Dans le même ordre d'idée, le nombre de registres utilisés va aussi être calculé.

Ces trois mesures vont permettre d'estimer le coût en terme de taille de l'implémentation, que ce soit la taille mémoire ou la taille physique du circuit embarqué. En plus de ces mesures de tailles, deux mesures vont être également estimées : la consommation de puissance et la fréquence maximale de fonctionnement. La première va permettre de déterminer si le chiffrement, ou ses contremesures, auront un fort impact sur la consommation d'énergie du circuit. Le second, surtout évalué en comparaison d'une implémentation simple et de l'ajout d'une ou plusieurs contremesures, va permettre de déterminer la complexité et le ralentissement ajoutés à l'implémentation en raison des défenses ajoutées.

Pour donner un exemple, nous avons implémenté un AES intégrant 16 S-boxes en parallèle sur deux FPGA de chez INTEL différents afin de comparer les coûts matériels. Cette version du chiffrement, bien que plus lourde dû à ces 16 S-boxes, permet de chiffrer un bloc de 128 bits de texte en clair en seulement 11 cycles d'horloge. La première implémentation s'est faite sur un *Cyclone V 5CGXFC9E7F35C8* et la seconde sur un *Cyclone 10 LP 10CL120ZF780I8G*. Le logiciel Quartus II a été utilisé pour la synthèse, avec les options « meilleurs efforts pour la performance et la taille physique » sélectionnées. Les résultats des coûts d'implémentation sont présentés sur le Tableau 2.3. Dans le tableau, *ALM* signifie *Adaptive Logic Module*, qui sont des éléments logiques de base créés pour maximiser les performances et l'usage des ressources. Chaque ALM est composé de 8 entrées, d'une Adaptive Look-Up Table (*ALUT*) à 8 entrées, de 4 registres ainsi que de 8 sorties. La table est nommée ALUT car toutes

les entrées/sorties ne sont pas forcément utilisées lors d'une implémentation. En effet, dans certains cas, seules la moitié des entrées seront utilisées. *LE* signifie simplement *Élément Logique*, qui est la plus petite unité de logique des architectures Intel Cyclone 10. Un LE est composé de 4 entrées, d'une LUT à 4 entrées, d'un registre ainsi que de 4 sorties.

TABLE 2.3 – Coût d'implémentation d'un AES composé de 16 S-boxes sur deux FPGA INTEL.

Performances sur le Cyclone V 5CGXFC9E7F35C8

Ressources FPGA				
Logique (ALM)	Registres	Bits de mémoire	Puissance (mW)	F_{max} (MHz)
339	13	32 768	566,32	94

Performances sur le Cyclone 10 LP 10CL120ZF780I8G

Ressources FPGA				
Logique (LE)	Registres	Bits de mémoire	Puissance (mW)	F_{max} (MHz)
649	13	32 768	233,23	119

On peut voir que les deux implémentations ont chacune des avantages et des inconvénients différents. En termes de ressources logiques, cette implémentation de l'AES nécessite le même nombre d'éléments dans chacun des FPGA. En effet, comme chaque ALM peut intégrer 8 entrées/sorties et chaque LE seulement 4, et que chaque ALUT des ALM peut ne pas être utilisée à 100%, nous avons exactement le même nombre d'entrées/sorties pour les deux implémentations. De même pour le nombre de registres et de bits mémoire utilisés qui sont identiques entre le Cyclone V et le Cyclone 10. Ce sont les valeurs de consommation et de fréquence maximale qui différencient les FPGA. Notons tout de même que les valeurs de consommation et de fréquence ne sont que des estimations données par le logiciel Quartus II. Cependant, comme les deux implémentations sont évaluées avec le même outil, même si les valeurs sont seulement approchées, les comparaisons entre les résultats peuvent quand même être intéressantes. Ainsi, l'AES sur Cyclone V sera moins rapide et consommera plus que l'AES sur Cyclone 10.

Chapitre 3

Respect du code généralisé

Ce chapitre présente une contribution qui porte sur une contremesure face aux attaques par injection de fautes basée sur l'application d'une redondance d'information applicable à n'importe quel chiffrement par blocs. La redondance d'information choisie est un code détecteur d'erreur : le bit de parité. L'objectif est d'intégrer la contremesure directement sur la primitive, en chiffrant les blocs de données, mais également en appliquant les calculs sur les bits de parité associés aux mots du bloc. L'algorithme doit ainsi être adapté afin de pouvoir prendre en compte cette information redondante. Simon *et al.* en 2020 ont utilisé cette approche sur une nouvelle permutation [Sim+20], présentée dans le Section 2.1.3. Cependant, leur solution est dédiée à cette permutation et peut donc difficilement être applicable sur des algorithmes de chiffrement de la littérature. Ainsi, l'objectif ici est de présenter une méthode permettant d'utiliser la notion de « respect du code »sur une contremesure plus générale.

Dans ce chapitre, nous allons mettre en lumière une injection particulière sur la permutation `FRIET-P` dont l'impact n'est pas détecté par la contremesure proposée par Simon *et al.*, expliquer quelles sont les conditions de cette injection et comment la contrer. Ensuite, nous allons appliquer notre méthode de « respect du code »avec un exemple sur le chiffrement léger LED. Les transformations des différentes fonctions de la primitive seront détaillées, ainsi qu'une évaluation du surcoût engendré par la contremesure. Le but étant de trouver une contremesure permettant de se protéger contre les injections de fautes sur un registre (injection simple ou multiple).

3.1 Erreurs compensées

Dans cette section, le but est de mettre en évidence un scénario d'injection dont l'impact ne serait pas détecté par une redondance d'information.

3.1.1 Exemple sur la fonction μ_2

Nous allons montrer dans un premier temps un exemple d'injection non détectée sur la permutations `FRIET-P` par une redondance d'information basée sur la solution choisie par Simon *et al.*, c'est à dire la somme binaire de toutes les sorties. Ce scénario a été imaginé sur une injection durant l'opération μ_2. Par souci de simplicité, nous allons nommer les variables d'entrée de la fonction a, b, c et d, et les variables de sortie a', b', c' et d'. La Figure 3.1 est un rappel de cette opération, avec la mise en valeur de l'injection de la faute f et de sa valeur. Comme la permutation `FRIET-P` manipule des données de 128 bits, la faute injectée ici est $f = 2^{128} - 1 =$ `0xFF..FF`. La faute est injectée sur le mot c, avant son intégration dans la somme binaire avec b. Ainsi, avec une telle injection, b' et c' sont impactées.

Les valeurs des données en sorties suivent les équations suivantes :

- $a' = a \oplus (c \lll 80)$
- $b' = b \oplus ((c \oplus f) \lll 80 = b \oplus (c \lll 80) \oplus (f \lll 80) = b \oplus (c \lll 80) \oplus f$
- $c' = c \oplus f$
- $d' = d$

Ainsi, la fonction de vérification, qui effectue la somme binaire des quatre sorties a', b', c' et d', ne va pas détecter la faute car son impact ($\oplus f$) va se compenser entre b' et c'.

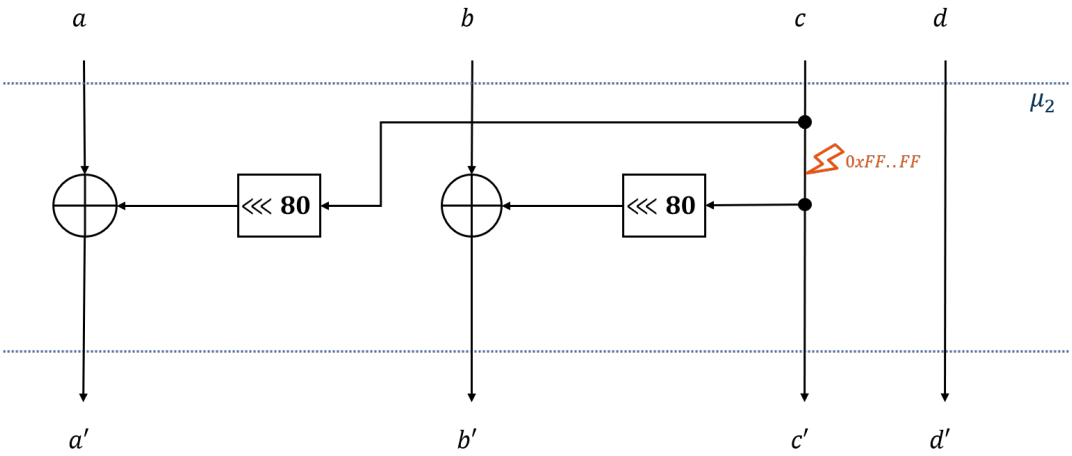

FIGURE 3.1 – Opération μ_2 de la permutation FRIET-P.

3.1.2 Généralisation de l'injection

Dans l'exemple précédent, nous avons montré que la faute dont la valeur est `0xFF..FF` n'est pas détectée si elle est injectée au bon instant de l'opération, car elle restait inchangée suite au décalage de 80 bits présent dans l'opération μ_2. Cependant, ce n'est pas la seule valeur de faute qui reste non détectée suite à un tel décalage. En effet, toutes les valeurs qui restent inchangées avec un décalage de 80 bits ne seront pas détectées. Le but est donc de déterminer quelles sont ces valeurs.

L'Algorithme 1 va permettre de trouver ces valeurs. La première étape est de trouver la longueur du cycle lié au décalage de 80 bits. La longueur du cycle est définie par le nombre de décalages à effectuer pour retomber sur la valeur initiale. Par exemple, l'algorithme prend en valeur initiale 1, et compte le nombre de décalages de 80 bits nécessaires pour retomber sur la valeur 1. La longueur du cycle trouvé est de 8.

Le nombre total de cycles est le PGCD entre la taille des données (ici 128) et la valeur du décalage. On peut aussi calculer le nombre de cycles total avec le quotient de la longueur de la donnée sur la taille du cycle. Le nombre de cycles est donc de $\texttt{PGCD}(128, 80) = \frac{128}{8} = 16$. Ce nombre de cycles permet de déterminer la longueur des mots à décaler sans les modifier. Ici, toute entrée du décalage composée de mots identiques de 16 bits reste inchangée. Toute faute composée de 8 mots identiques de 16 bits ne sera pas détectée.

Algorithm 1 Longueur d'un cycle de décalage.

Require: Taille du décalage en bits (ici 80)
Ensure: Longueur d'un cycle
  $i \leftarrow 1$
  $t_{cycle} \leftarrow 1$
  while $i \lll 80 \neq 1$ **do**
    $i \leftarrow i \lll 80$
    $t_{cycle} \leftarrow t_{cycle} + 1$
  return t_{cycle}

Comme les fautes non détectées sont la concaténation de 8 mots identiques de 16 bits, nous pouvons en dénombrer 2^{16}. Cependant, dans ce panel de fautes, la valeur `0x00..00` ne peut pas être prise en compte car elle ne correspond pas réellement à une valeur possible de faute. Ainsi, nous avons $2^{16} - 1$ fautes non détectées. En injectant des fautes aléatoires, même au bon endroit du chiffrement, la probabilité d'apparition d'une faute non détectée est donc d'environ 2^{-112}. Néanmoins, dans un modèle de faute plus précis que les fautes aléatoires, l'adversaire peut choisir d'injecter une erreur non détectable à chaque fois. Ainsi, une contremesure doit être apportée pour palier cette non-détection.

3.1.3 Solutions apportées

Une première solution pourrait être de réduire la taille du décalage pour limiter le nombre de fautes non détectées. Tout décalage d pour lequel $\texttt{PGCD}(d, 128) = 1$ permet de réduire au maximum le nombre de fautes non détectées. Pour de tels décalages, seules toutes les valeurs composées de la concaténation de 128 mots identiques de 1 bit restent non détectées. Cela représente uniquement la valeur `0xFF.FF` (la valeur `0x00..00` n'est toujours pas considérée comme une faute). Cependant, il reste quand même une faute non détectée, et la solution induit une modification dans la primitive de base.

Une seconde solution doit être apportée afin de détecter l'ensemble des fautes sans avoir à modifier la primitive. La méthode trouvée est d'effectuer une copie de toutes les valeurs intermédiaires utilisées plusieurs fois dans une opération et de vérifier si toutes les valeurs copiées sont égales avant de les utiliser. Une copie sera créée pour chaque utilisation de la valeur. Ainsi, si lors d'une fonction, une valeur est utilisée trois fois, alors trois copies seront créées. Ainsi, avec ce principe, une faute injectée durant une opération n'affecte qu'une seule copie de la variable, et ainsi elle est détectée sans pouvoir se compenser. Les Algorithmes 2 et 3 présentent la différence entre l'opération classique et l'opération protégée. Le surcoût emmené par cette solution est lié au nombre de copies créées et à la vérification de l'égalité de ces copies. Pour représenter la comparaison entre les différentes copies, une valeur booléenne `flag` sera utilisée, en considérant cette valeur comme globale et accessible à tout instant par la primitive. En effet, le designer de l'algorithme peut choisir à quel moment il effectue les tests de vérification de ce bit. Le but ici est de détecter à tout moment une faute injectée dans l'algorithme, et non de déterminer quelles sont les actions à planifier en cas de détection. La solution la plus simple dans ce cas est une valeur booléenne représentant la présence ou non d'une erreur.

Algorithm 2 Opération μ_2 classique.

Require: Quatre mots d'entrée a, b, c et d
Ensure: Quatre mots de sortie a', b', c' et d'
$a' \leftarrow a \oplus (c \lll 80)$
$b' \leftarrow b \oplus (c \lll 80)$
$c' \leftarrow c$
$d' \leftarrow d$
return (a', b', c', d')

Algorithm 3 Opération μ_2 protégée.

Require: Quatre mots d'entrée a, b, c et d
Ensure: Quatre mots de sortie a', b', c' et d'
$c_0 \leftarrow c$
$c_1 \leftarrow c$
$c_2 \leftarrow c$
$\texttt{flag} \leftarrow \texttt{flag}$ & $(\texttt{c}_0 == \texttt{c}_1)$ & $(\texttt{c}_0 == \texttt{c}_2)$
$a' \leftarrow a \oplus (c_0 \lll 80)$
$b' \leftarrow b \oplus (c_1 \lll 80)$
$c' \leftarrow c_2$
$d' \leftarrow d$
return (a', b', c', d')

3.2 Respect du code appliqué sur le chiffrement LED

Les différentes opérations appliquées à l'état du chiffrement doivent observer la notion de « respect du code ». En effet, un code détecteur d'erreur permet de séparer les valeurs manipulées en deux catégories : celles qui sont dans le code (appelées « bonnes valeurs ») vérifiant les propriétés du code, et celles qui sont hors du code ne vérifiant pas ses propriétés.

Dans notre cas, le code détecteur d'erreur choisi est le bit de parité, avec une parité paire pour les mots dans le code, et impaire pour les mots hors du code. Plus formellement, nous allons nous référer au code $\mathcal{C}$ proposé dans la Définition 8.

Définition 8 (Code détecteur d'erreur $\mathcal{C}$ de paramètres n, k, d) *Un code binaire $\mathcal{C}$ de dimension k et de longueur n est un sous-espace vectoriel de $\mathbb{F}_2^n$ de dimension k, avec une distance minimale de d entre les mots du code.*

Le but de notre contribution est de présenter une méthode applicable à tout type de chiffrement par blocs, en donnant un exemple sur un chiffrement déjà existant, le chiffrement LED. Étant appliqué à un état de 64 bits, découpé en 16 mots de 4 bits, les paramètres choisis seront :

- $k = 4$ pour des mots de longueur 4.
- $n = 5$ avec l'ajout d'un bit de parité aux quatre bits de données.
- $d = 2$ pour la distance de Hamming minimale entre deux mots du code qui permettra de détecter une injection de faute.

Ce code pourra être concaténé pour atteindre les 16 mots de 4 bits. Ainsi, le code global [80,64,2] n'est qu'une concaténation de 16 codes [5,4,2].

Chaque fonction utilisée doit respecter cette séparation. Une entrée dans le code doit toujours être reliée à une sortie dans le code, et une entrée hors du code doit toujours

être reliée à une sortie hors du code. Ainsi, une faute injectée va faire passer l'état hors du code et les fonctions respectant le code vont permettre de garder l'état hors du code jusqu'à la détection de l'injection. Plus formellement, une fonction respectant le code vérifie la Définition 9.

Définition 9 (Fonction respectant le code $\mathcal{C}$) *Une fonction f respecte le code $\mathcal{C}$ si et seulement si :*

— $\forall x \in \mathcal{C}, f(x) \in \mathcal{C}$

— $\forall x \notin \mathcal{C}, f(x) \notin \mathcal{C}$

Ainsi, la suite de cette section va présenter les modifications appliquées au chiffrement LED, au niveau du format de son état à chiffrer et sur les différentes opérations composant le chiffrement, qui doivent maintenant prendre en compte l'ajout du bit de parité sur chaque mot.

3.2.1 Modification de l'état

Notons S l'état sur 64 bits d'un chiffrement LED non protégé. En appliquant le code détecteur d'erreur choisi, un bit de parité doit être ajouté à chaque mot de 4 bits contenu dans l'état. En dénotant S_i le $i^{\text{ème}}$ bit de l'état S, nous avons donc :

$$S_{64+i} = S_{4\times i} \oplus S_{4\times i+1} \oplus S_{4\times i+2} \oplus S_{4\times i+3}$$

Avec cette modification, nous avons désormais un état de 80 bits, composé de 64 bits de données et de 16 bits de parité. La Figure 3.2 montre ce nouvel état sur 80 bits, avec quelques exemples de mots de 4 bits et de leur bit de parité associé.

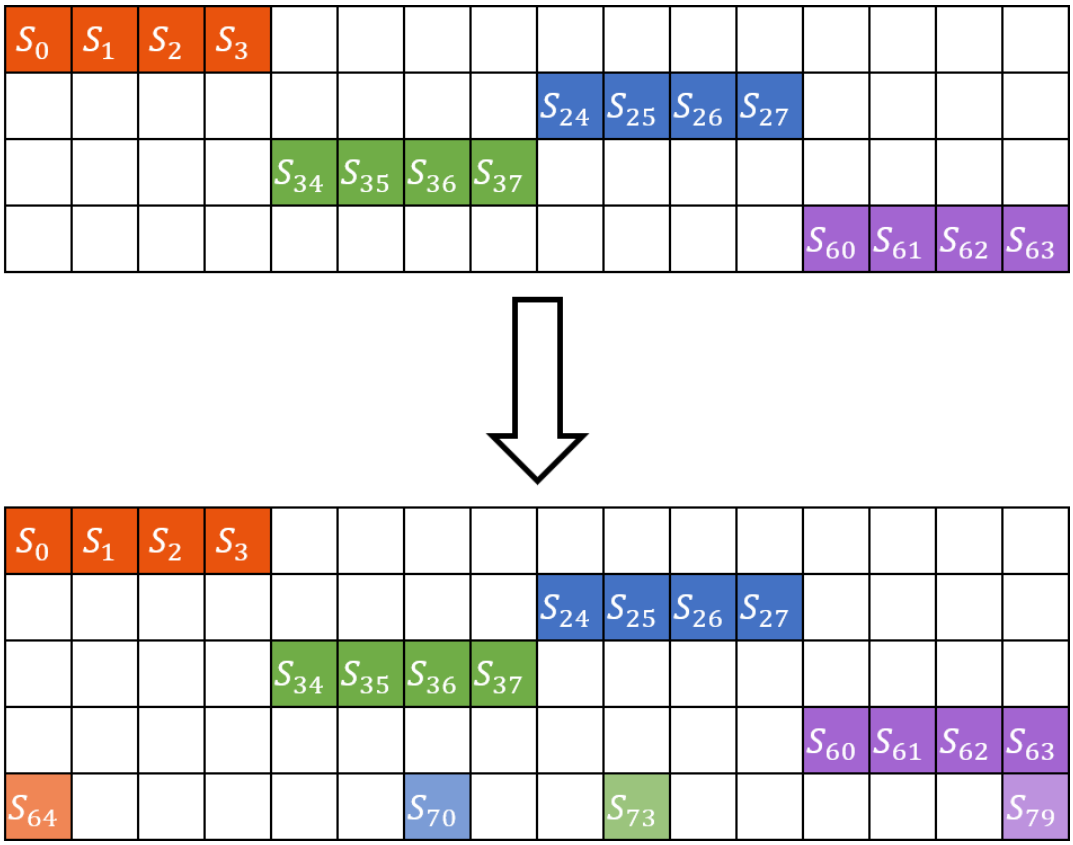

FIGURE 3.2 – Ajout des bits de parité à l'état.

En plus de l'ajout des bits de parité, le chiffrement va également être modifié pour respecter sa version bitslice. En effet, cette transformation va permettre de voir les registres non plus comme des états complets mais comme des bits indépendants. De plus, comme 80 registres seront nécessaires pour les 80 bits de l'état, les registres peuvent être complétés par d'autres données à chiffrer. Ainsi, sur une implémentation dont les registres sont de taille l, la version protégée du chiffrement pourra chiffrer jusqu'à l textes en clair en parallèle et en pouvant utiliser l clés différentes. En plus de cette parallélisation, la version bitslice nous assure une protection contre les injections non plus sur un bit ou un mot d'un registre mais bien sur le registre entier. En effet, même si tous les bits d'un registre sont fautés, cela ne correspond qu'à un seul bit de chaque état à chiffrer. Ainsi, ces erreurs engendrent une disparité entre le bit de parité existant et celui recalculé. N'importe quelle faute sur un registre du dispositif cryptographique sera détectée.

La Figure 3.3 montre cette transformation, en partant du principe que l textes doivent être chiffrés. Dans cette représentation, S_i^j correspond au bit i de l'état j. Ainsi, pour avoir un état complet dans chaque registre dans une version classique, nous avons besoin de l registres de taille 80 minimum, quand dans la version bitslice, il est nécessaire d'avoir 80 registres de taille l minimum.

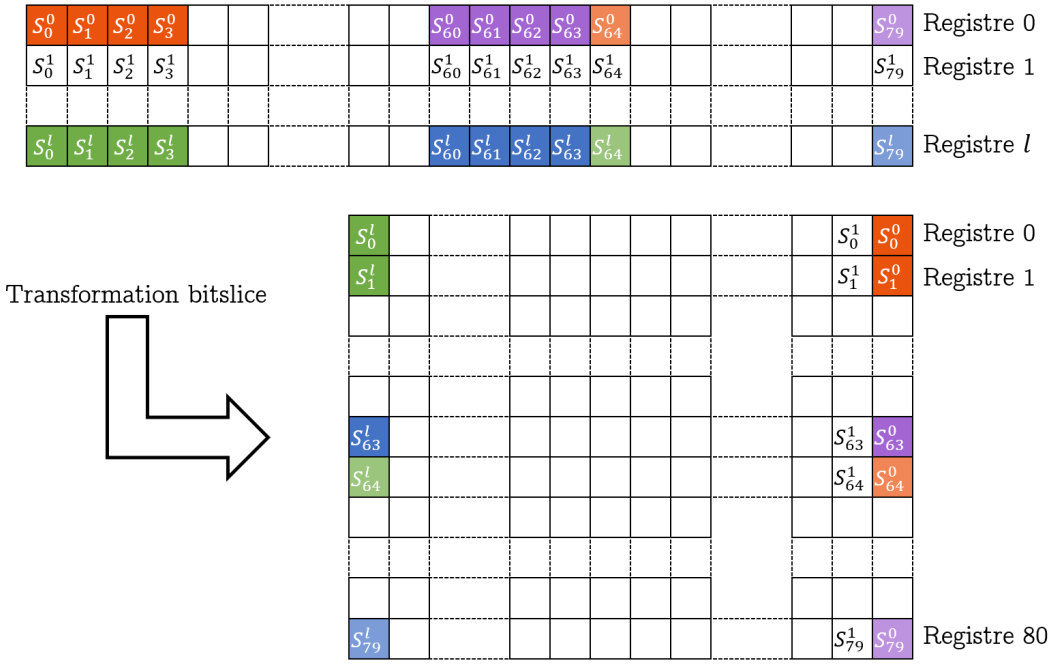

FIGURE 3.3 – Transformation bitslice des différents états.

L'Algorithme 4 présente l'ajout des bits de parité à l'état, et l'Algorithme 5 montre la transformation de l'état dans sa version bitslice. Ces deux transformations seront appliquées l'une après l'autre en début de chiffrement, afin de pouvoir correctement utiliser les futures opérations.

La fonction permettant la vérification de l'injection d'une erreur est très similaire à celle présentée dans l'Algorithme 4. La seule différence est que la valeur contenue dans S_{64+i} est comparée à $S_{4\times i} \oplus S_{4\times i+1} \oplus S_{4\times i+2} \oplus S_{4\times i+3}$ plutôt que d'être assignée à

cette même somme. Si le calcul du bit de parité (la somme binaire) est différente du bit de parité stocké, alors la contremesure permet de démontrer une injection de faute.

Algorithm 4 Ajout des bits de parité à l'état S.

Require: État S
Ensure: État S avec les bits de parité
  for i de 0 à 15 **do**
    $S_{64+i} \leftarrow S_{4\times i} \oplus S_{4\times i+1} \oplus S_{4\times i+2} \oplus S_{4\times i+3}$
  return S

Algorithm 5 Transformation des états S^j dans la version bitslice.

Require: l états S^j
Ensure: État S avec en version bitslice
  for i de 0 à 79 **do**
    for j de 0 à l **do**
      $S_i^j \leftarrow S_j^i$
  return S

3.2.2 *AddRoundKey*

La fonction *AddRoundKey* permet d'intégrer la clé secrète à l'état du chiffrement. Comme vu précédemment, l'état n'est plus considéré comme 16 mots de 4 bits mais plutôt comme 80 registres de 1 bit chacun. De même, la clé est également transformée pour correspondre à l'état. Ainsi, la fonction doit être adaptée pour coller à ces nouvelles formes. L'intégration de la clé se fait par une opération de somme binaire. Ainsi, comme cette opération est bit-à-bit, appliquer le XOR sur tout le registre en parallèle ne mélange pas les états entre eux. Les chiffrements en parallèle sont donc toujours possibles. De plus, les clés utilisées pour les chiffrements peuvent être identiques (une même clé qui est dupliquée sur toute la largeur du registre) ou différentes (chaque clé est stockée avec le bon décalage correspondant à son état à chiffrer). Ces deux cas peuvent correspondre à l'envoi de plusieurs messages en parallèle au même destinataire (donc avec la même clé) ou l'envoie d'un (ou plusieurs messages) à différents destinataires (donc avec une clé différente par destinataire). En considérant que chaque chiffrement utilise une clé différente, l'Algorithme 6 présente la transformation des clés pour les stocker dans le bon état.

Algorithm 6 Transformation des clés.

Require: Ensemble de clés k (k_j faisant référence à la clé j).
Ensure: Clés K dans leur version bitslice (K_i faisant référence au registre i).
  for i de 0 à 79 **do**
    for j de 0 à l **do**
      $K_i \leftarrow K_i \vee ((k_j \wedge i) \lll j)$
  return K

Même si les clés sont différentes, la méthode de duplication d'une valeur de 64 bits en 80 registres de l bits sera utilisée dans la fonction *AddConstant*, pour appliquer la constante à tous les états. L'Algorithme 7 montre cette duplication.

Algorithm 7 Duplication d'une donnée.

Require: Une donnée d sur 64 bits (d_i étant le bit i de la donnée).
Ensure: 80 registres de données D sur l bits (i étant l'indice du registre).
  $d = \texttt{Parity}(d)$ ▷ Algorithme 4
  for i de 0 à 79 **do**
    $D_i \leftarrow d_i \times 0b11\ldots11$ ▷ On retrouve l fois 1 pour une largeur de l bits
  return D

Ainsi, que les clés soient identiques ou différentes, les Algorithmes 6 et 7 renvoient 80 registres de l bits contenant toutes les clés. L'étape suivante est d'intégrer ces clés aux états chiffrés en parallèle, comme présenté dans l'Algorithme 8. On voit bien que le seul indice utilisé est celui des registres, comme les chiffrements en parallèle n'interfèrent pas les uns avec les autres. De plus, si une faute est injectée dans un des registres, comme le modèle le permet, alors cette faute sera propagée en sortie de la fonction. Par conséquent, la nouvelle fonction `AddRoundKey` respecte bien le code détecteur.

Algorithm 8 Fonction `AddRoundKey`.

Require: États S et clés K.
Ensure: États S avec clés intégrées.
  for i de 0 à 79 **do**
    $S_i \leftarrow S_i \oplus K_i$
  return S

3.2.3 *AddConstant*

La constante présentée dans le Chapitre 2 et rappelée par la Matrice suivante est calculée, dupliquée et ajoutée aux états (voir Algorithme 9).

$$RC = \begin{pmatrix} 0 \oplus (ks_7 \| ks_6 \| ks_5 \| ks_4) & (rc_5 \| rc_4 \| rc_3) & 0 & 0 \\ 1 \oplus (ks_7 \| ks_6 \| ks_5 \| ks_4) & (rc_2 \| rc_1 \| rc_0) & 0 & 0 \\ 2 \oplus (ks_3 \| ks_2 \| ks_1 \| ks_0) & (rc_5 \| rc_4 \| rc_3) & 0 & 0 \\ 3 \oplus (ks_3 \| ks_2 \| ks_1 \| ks_0) & (rc_2 \| rc_1 \| rc_0) & 0 & 0 \end{pmatrix}$$

Algorithm 9 Fonction `AddConstant`.

Require: États S
Ensure: États S avec constante intégrée.
  $RC \leftarrow \texttt{Duplicate}(\texttt{Parity}(RC))$ ▷ Algorithme 4 et 7
  for i de 0 à 79 **do**
    $S_i \leftarrow S_i \oplus RC_i$
  return S

3.2.4 *ShiftRows*

La fonction de rotation présentée précédemment est très peu modifiée. En effet, ce sont cette fois-ci les registres qui sont décalés plutôt que les bits dans le registre, mais le principe reste le même. De plus, les registres de parité sont décalés de la même manière que les mots dont ils sont issus. Ainsi, le registre S_{64+i} est décalé de

$\frac{i}{4}$ registres. La Figure 3.4 montre cette rotation. Par soucis de compréhension et de présentation, l'état est montré dans sa version non bitslice.

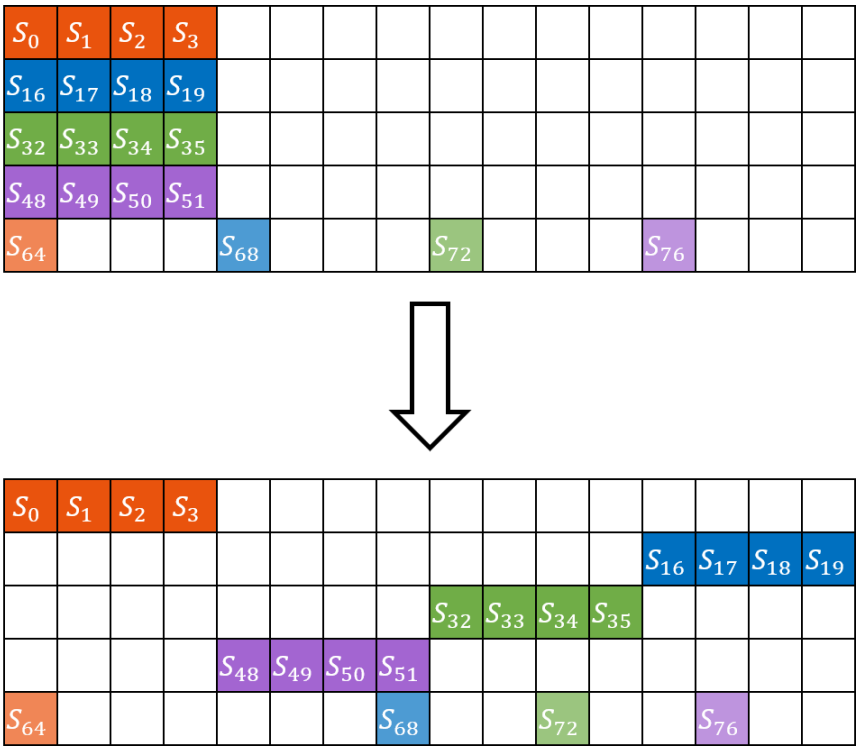

FIGURE 3.4 – Fonction *ShiftRows* adaptée.

3.2.5 *SubCells*

La fonction *SubCells* amène la partie non linéaire de l'algorithme et elle contribue à la confusion du chiffrement. Ici, la S-box, originellement sur 4 bits, doit être adaptée aux nouveaux mots de 5 bits (4 bits de donnée + 1 bit de parité). Ainsi, nous devons étendre la S-box de `PRESENT` (redonnée sur le Tableau 3.1) sur 5 bits.

TABLE 3.1 – S-box `PRESENT`.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Nous avons fait le choix de placer le bit de parité comme le bit de poids faible du mot. Ainsi, la moitié de la représentation sur 5 bits de la S-box est déjà remplie : on ajoute le bit de parité aux entrées sur 4 bits et à leurs sorties correspondantes. Ainsi, nous avons une S-box sur 5 bits, dérivée de la S-box du chiffrement `PRESENT` à compléter. Cette première S-box est présentée sur le Tableau 3.2.

TABLE 3.2 – S-box `PRESENT` à remplir.

x	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
$S[x]$	18	...	...	0A	...	0C	17	...	...	12	00	...	14	...	...	1B
x	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S[x]$	...	06	1D	...	1E	...	...	11	09	...	...	0F	...	03	05	...

Plusieurs solutions s'offrent alors pour remplir cette S-box. Le but de cette contre-mesure est de détecter une injection de faute sur un registre, tout en réduisant au maximum le surcoût engendré. La première, déjà utilisée dans [Ber+02], est de remplacer toutes les valeurs à remplir par la même valeur, en dehors du code. Ainsi, dans leur implémentation, ils ont remplacé toutes les valeurs manquantes par la valeur « 1 ». « 1 » ayant une parité impaire, elle est considérée hors du code. Cependant, l'objectif est de créer une version matérielle protégée d'un algorithme. Ainsi, nous voulons limiter les coûts matériels de l'implémentation.

Une autre solution est d'effectuer le calcul du bit de parité de l'entrée de la S-box, et de le comparer avec une estimation de ce bit de parité depuis la sortie de la S-box. Cette solution permet de ne pas avoir à stocker de bit de parité directement dans la S-box. C'est la solution proposée par Di Natale *et al.* [NFR07]. Cette solution est efficace, bien qu'assez coûteuse en raison de la logique ajoutée pour la prédiction et la comparaison.

Notre idée est donc de garder le principe de l'ajout du bit de parité directement dans la S-box, mais en limitant le coût d'implémentation. Afin de comparer les différents candidats entre eux, une fonction de score est introduite. Pour notre part, le score d'une S-box est le nombre de portes logiques nécessaires à son implémentation. Ainsi, nous allons transformer les S-boxes dans leur forme ANF (voir Définition 10). Cette forme va permettre de compter exactement combien de portes logiques `AND` et `XOR` seront utilisées lors de l'implémentation matérielle. De plus, la forme ANF d'une fonction booléenne est appliquée au bit, et non au mot. Ainsi, chaque registre de sortie a sa propre opération, et notre version bitslice est toujours possible.

Définition 10 (Algebraic Normal Form (ANF)) *En algèbre booléenne, la forme ANF d'une formule consiste à l'écrire sous une forme qui ne contient que des variables, des termes purement vrais (1) et des opérations* `AND` *(*$\wedge$*) et* `XOR` *(*$\oplus$*).*

Un exemple de forme ANF appliquée à la S-box `PRESENT` sur 4 bits est proposé dans l'Exemple 3.

Exemple 3 (Algebraic Normal Form (ANF)) *En notant* $x = x_3||x_2||x_1||x_0$ *l'entrée de la S-box (*$||$ *étant l'opérateur de concaténation) et* $y = y_3||y_2||y_1||y_0$ *la sortie de la S-box,* y *suit les équations suivantes (*$\oplus$ *est l'opérateur* `XOR` *et* $\wedge$ *l'opérateur* `AND`*) :*

- $y_0 = x_0 \oplus x_1 \wedge x_2 \oplus x_2 \oplus x_3$
- $y_1 = x_0 \wedge x_1 \wedge x_2 \oplus x_0 \wedge x_1 \wedge x_3 \oplus x_0 \wedge x_2 \wedge x_3 \oplus x_1 \wedge x_3 \oplus x_1 \oplus x_2 \wedge x_3 \oplus x_3$
- $y_2 = x_0 \wedge x_1 \wedge x_3 \oplus x_0 \wedge x_1 \oplus x_0 \wedge x_2 \wedge x_3 \oplus x_0 \wedge x_3 \oplus x_1 \wedge x_3 \oplus x_2 \oplus x_3 \oplus 1$
- $y_3 = x_0 \wedge x_1 \wedge x_2 \oplus x_0 \wedge x_1 \wedge x_3 \oplus x_0 \wedge x_2 \wedge x_3 \oplus x_0 \oplus x_1 \wedge x_2 \oplus x_1 \oplus x_3 \oplus 1$

La fonction `score` permettant de donner un score (nombre de portes logiques nécessaires à l'implémentation) à une S-box est présentée dans l'Algorithme 10. Appliqué à la S-box `PRESENT`, cet algorithme renvoie une valeur de 46 portes logiques.

Algorithm 10 Score d'une S-box.

Require: S-box S
Ensure: Nombre d'opérations logiques dans la forme ANF
  $f \leftarrow \texttt{ANF}(S)$
  return $f.\texttt{count}(\wedge) + f.\texttt{count}(\oplus)$

Liste Exhaustive

Ainsi, pour déterminer quelle est la meilleure S-box sur 5 bits à choisir, il faudrait calculer le score de tous les candidats, et choisir un candidat parmi ceux qui ont le score le plus bas. Cependant, avec 16 cases à remplir et 16 options pour chaque case, nous nous retrouvons avec 16^{16} candidats. Il est donc impossible de tout tester par calculs dans un temps raisonnable. Premièrement, le code de parité utilisé est le code $\mathcal{C} = [5, 4, 2]$, qui est concaténé pour retrouver un nouveau code au niveau de l'état : $\mathcal{C}' = [80, 64, 2]$. La concaténation des 16 codes $\mathcal{C}$ impose que les candidats pour la S-box représentée sur 5 bits doivent être des permutations. Ainsi chaque sortie est reliée à une et une seule entrée. Cela donne 16 ! candidats à évaluer. Ce nombre reste toujours trop élevé pour tous les estimer. Le Chapitre 4 présentera une méthode pour diminuer ce nombre de candidats, mais l'Algorithme 11 présente quand même le pseudo code qui permettrait de choisir une des meilleures S-boxes selon le critère du nombre de portes logiques.

Algorithm 11 Sélection de la S-box avec le plus petit nombre de portes logiques.

Require: Liste de toutes les permutations sur 5 bits dérivées de la S-box sur 4 bits (`Permutations`).
Ensure: Permutation avec le score le plus bas ainsi que son score.

```
low_score = 1000
for S ∈ Permutations do
    s ← score(S)                                  ▷ Algorithme 10
    if s < low_score then
        low_score ← s
        low_sbox ← S
return low_sbox, low_score
```

En exécutant ce code sur une partie non totale des candidats ayant de bonnes propriétés cryptographiques, le choix se porte temporairement sur une S-box nécessitant 94 portes logiques (quand la S-box la plus chère en requérait 124).

Construction

Afin d'éviter d'évaluer les 16 ! candidats, une nouvelle approche est introduite. Le but ici est de construire la S-box sur 5 bits depuis celle sur 4 bits. Dans la liste des 32 mots sur 5 bits, chaque mot du code est à un bit d'un mot hors du code. En choisissant le bit de poids faible comme séparation entre les mots du code et ceux hors du code, nous avons par exemple le mot 18 (dans le code) qui est à un bit de poids faible du mot 19 (hors du code). De plus, chaque entrée dans le code est substituée par une sortie dans le code, et chaque entrée hors du code doit être substituée par une sortie hors du code.

La méthode de construction suit donc la règle suivante : *une entrée hors du code est substituée par la sortie hors du code séparée d'un bit de poids faible de la sortie dans le code issue de l'entrée dans le code séparée d'un bit de poids faible de l'entrée hors du code originale*. L'Algorithme 12 permet de clarifier cette règle en montrant comment sont construites les sorties vides (. . .).

Grâce à cette méthode, nous retrouvons la S-box qui sera utilisée par la suite, présentée dans le Tableau 3.3, ne nécessitant que 62 portes logiques afin d'être implémentée.

Algorithm 12 Méthode de construction de la représentation sur 5-bit respectant le code de la S-box originale sur 4 bits.

Require: S-box S semi remplie.
Ensure: S-box S remplie entièrement.
  for i de 0 à 31 **do**
    if $S[i] = \ldots$ **then**
      $S[i] \leftarrow S[i \oplus 1] \oplus 1$
  return S

TABLE 3.3 – `PRESENT` S-box.

x	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
$S[x]$	18	19	0B	0A	0D	0C	17	16	13	12	00	01	14	15	1A	1B
x	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
$S[x]$	07	06	1D	1C	1E	1F	10	11	09	08	0E	0F	02	03	05	04

Cette S-box ne sera pas implémentée sous forme de tableau, mais plutôt sous sa forme ANF, avec chaque bit de sortie correspondant à une équation faisant intervenir les bits d'entrée. Ces équations sont données par la suite, avec x_i faisant référence au i^e bit de l'entrée et y_i le i^e bit de la sortie (le bit de poids faible étant à l'indice 0). La forme ANF ne comportant que des opérations `AND` et `XOR`, les opérateurs utilisés sont liés à ces opérations (respectivement $\wedge$ et $\oplus$).

- $y_0 = x_0 \oplus x_1 \wedge x_2 \wedge x_4 \oplus x_1 \wedge x_2 \oplus x_1 \wedge x_3 \wedge x_4 \oplus x_1 \wedge x_4 \oplus x_1 \oplus x_2 \oplus x_3 \wedge x_4 \oplus x_3 \oplus x_4$
- $y_1 = x_1 \oplus x_2 \wedge x_3 \oplus x_3 \oplus x_4$
- $y_2 = x_1 \wedge x_2 \wedge x_3 \oplus x_1 \wedge x_2 \wedge x_4 \oplus x_1 \wedge x_3 \wedge x_4 \oplus x_2 \wedge x_4 \oplus x_2 \oplus x_3 \wedge x_4 \oplus x4$
- $y_3 = x_1 \wedge x_2 \wedge x_4 \oplus x_1 \wedge x_2 \oplus x_1 \wedge x_3 \wedge x_4 \oplus x_1 \wedge x_4 \oplus x_2 \wedge x_4 \oplus x_3 \oplus x_4 \oplus 1$
- $y_4 = x_1 \wedge x_2 \wedge x_3 \oplus x_1 \wedge x_2 \wedge x_4 \oplus x_1 \wedge x_3 \wedge x_4 \oplus x_1 \oplus x_2 \wedge x_3 \oplus x_2 \oplus x_4 \oplus 1$

La section précédente a montré le problème de l'utilisation multiple de certaines variables. Ainsi, lors de l'implémentation, les variables qui seront utilisées plusieurs fois seront copiées et comparées avant d'être utilisées. La variable `flag` est toujours utilisée pour déterminer la présence d'une faute sur les copies. L'Algorithme 13 présente la fonction.

Si une faute est présente avant la fonction, la séparation du code permet de garder cette erreur dans l'état, et si une faute est injectée durant le déroulé de la fonction, alors elle est propagée à la sortie de la fonction. `SubCells` est donc bien une fonction respectant le code de parité.

3.2.6 *MixColumnsSerial*

Pour rappel, cette fonction est composée de 4 multiplications avec la Matrice A. L'état est décomposé en quatre colonnes de quatre mots de 5 bits chacun (4 bits de données et 1 bit de parité).

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}$$

Seule la multiplication par 2 est implémentée, comme la multiplication par 4 revient à faire deux multiplications par 2. Cette multiplication est donnée dans l'Algorithme 14.

Algorithm 13 Fonction `SubCells`.

Require: S-box S, indice i, variable `flag`.
Ensure: S-box S substituée, variable `flag`.
for i de 0 à 15 **do**
for j de 0 à 4 **do**
$x_{j4} \leftarrow S_{4\times i}$
$x_{j3} \leftarrow S_{4\times i+1}$
$x_{j2} \leftarrow S_{4\times i+2}$
$x_{j1} \leftarrow S_{4\times i+3}$
for j de 0 à 4 **do**
`flag` $\leftarrow$ `flag` & $(x_{0j} = x_{1j})$ & $(x_{0j} = x_{2j})$ & $(x_{0j} = x_{3j})$ & $(x_{0j} = x_{4j})$
$S_{64+i} \leftarrow S_{64+i} \oplus x_{01} \wedge x_{02} \wedge x_{04} \oplus x_{01} \wedge x_{02} \oplus x_{01} \wedge x_{03} \wedge x_{04} \oplus x_{01} \wedge x_{04} \oplus x_{01} \oplus x_{02} \oplus x_{03} \wedge x_{04} \oplus x_{03} \oplus x_{04}$
$S_{4\times i+3} \leftarrow x_{11} \oplus x_{12} \wedge x_{13} \oplus x_{13} \oplus x_{14}$
$S_{4\times i+2} \leftarrow x_{21} \wedge x_{22} \wedge x_{23} \oplus x_{21} \wedge x_{22} \wedge x_{24} \oplus x_{21} \wedge x_{23} \wedge x_{24} \oplus x_{22} \wedge x_{24} \oplus x_{22} \oplus x_{23} \wedge x_{24} \oplus x_{24}$
$S_{4\times i+1} \leftarrow x_{31} \wedge x_{32} \wedge x_{34} \oplus x_{31} \wedge x_{32} \oplus x_{31} \wedge x_{33} \wedge x_{34} \oplus x_{31} \wedge x_{34} \oplus x_{32} \wedge x_{34} \oplus x_{33} \oplus x_{34} \oplus 1$
$S_{4\times i} \leftarrow x_{41} \wedge x_{42} \wedge x_{43} \oplus x_{41} \wedge x_{42} \wedge x_{44} \oplus x_{41} \wedge x_{43} \wedge x_{44} \oplus x_{41} \oplus x_{42} \wedge x_{43} \oplus x_{42} \oplus x_{44} \oplus 1$
return S, `flag`

De même que pour l'opération précédente, des variables sont utilisées plusieurs fois, et sont donc copiées avant d'être vérifiées.

Algorithm 14 Fonction `mc2`.

Require: Mot de 5 bits *nibble*, `flag`.
Ensure: $2 \times$ *nibble*, `flag`.
$nib_{30} \leftarrow nibble[3]$
$nib_{31} \leftarrow nibble[3]$
$nib_{32} \leftarrow nibble[3]$
`flag` $\leftarrow$ `flag` & $(nib_{30} = nib31)$ & $(nib_{30} = nib32)$

$nibble[0], nibble[1], nibble[2], nibble[3], nibble[4] \leftarrow nib_{30}, nibble[0] \oplus nib_{31}, nibble[1], nibble[2], nibble[4] \oplus nib_{32}$

return *nibble*, `flag`

L'état est donc divisé en colonnes, qui sont chacune leur tour multipliées quatre fois par la matrice A. Cette opération est présentée dans l'Algorithme 15.

Avec les mêmes observations que pour les fonctions précédentes, si une faute est injectée avant ou pendant l'exécution de la fonction, alors elle sera propagée en sortie de celle-ci.

Toutes les fonctions du chiffrement LED ont donc été adaptées pour respecter le code et garder la caractéristique de parité de l'état et ainsi détecter une injection de fautes sur un registre. La prochaine étape est de valider la sécurité en testant la robustesse de l'implémentation face à des injections de fautes.

Algorithm 15 Fonction `MixColumnsSerial`.

Require: État S.
Ensure: État S mixé.

function `MixSingleColumn`(col)
  for i de 0 à 3 **do**
    $nibble_i \leftarrow [col[4 \times i], col[4 \times i + 1], col[4 \times i + 2], col[4 \times i + 3], col[16 + i]$
  for i de 0 à 3 **do**
    $nibble_0, nibble_1, nibble_2, nibble_3 \leftarrow nibble_1, nibble_2, nibble_3, \texttt{mc2}(\texttt{mc2}(nibble_0) \oplus nibble_1 \oplus \texttt{mc2}(nibble_2) \oplus \texttt{mc2}(nibble_3)$
  return col

function `MixColumnsSerial`(S)
  $col_0 \leftarrow [S_0, S_1, S_2, S_3, S_{16}, S_{17}, S_{18}, S_{19}, S_{32}, S_{33}, S_{34}, S_{35}, S_{48}, S_{49}, S_{50}, S_{51}, S_{64}, S_{68}, S_{72}, S_{76}]$
  $col_1 \leftarrow [S_4, S_5, S_6, S_7, S_{20}, S_{21}, S_{22}, S_{23}, S_{36}, S_{37}, S_{38}, S_{39}, S_{52}, S_{53}, S_{54}, S_{55}, S_{65}, S_{69}, S_{73}, S_{77}]$
  $col_2 \leftarrow [S_8, S_9, S_{10}, S_{11}, S_{24}, S_{15}, S_{26}, S_{27}, S_{40}, S_{41}, S_{42}, S_{43}, S_{56}, S_{57}, S_{58}, S_{59}, S_{66}, S_{70}, S_{74}, S_{78}]$
  $col_3 \leftarrow [S_{12}, S_{13}, S_{14}, S_{15}, S_{28}, S_{29}, S_{30}, S_{31}, S_{44}, S_{45}, S_{46}, S_{47}, S_{60}, S_{61}, S_{62}, S_{63}, S_{67}, S_{71}, S_{75}, S_{79}]$

  $col_0 \leftarrow \texttt{MixSingleColumn}(col_0)$
  $col_1 \leftarrow \texttt{MixSingleColumn}(col_1)$
  $col_2 \leftarrow \texttt{MixSingleColumn}(col_2)$
  $col_3 \leftarrow \texttt{MixSingleColumn}(col_3)$

  return S

3.3 Résultats expérimentaux

Cette section va présenter les différents tests effectués sur la version protégée de LED afin de déterminer sa robustesse ainsi que son coût d'implémentation, à comparer avec une implémentation classique de LED.

3.3.1 Robustesse

Afin de tester la robustesse du chiffrement LED protégé, plusieurs scénarios ont été testés. La détection de la faute est affichée par la variable `flag`, utilisée pour la vérification des différentes copies dans les algorithmes précédents, ainsi que dans la fonction de vérification de la parité, présentée dans l'Algorithme 16. Si le Booléen `flag` est à « 0 » en sortie des tests, c'est qu'une faute a été détectée. Les injections sont simulées, afin de garantir l'effet de la faute, quand une injection réelle peut ne pas avoir d'impact sur le circuit.

Algorithm 16 Fonction `parityCheck`.

Require: État S, `flag`.
Ensure: `flag`
  for i de 0 à 15 **do**
    `flag` $\leftarrow$ `flag` & $(S_{64+i} = S4 \times i \oplus S4 \times i + 1 \oplus S4 \times i + 2 \oplus S4 \times i + 3)$
  return `flag`

Scénario 1 : une faute aléatoire est injectée dans un registre d'état aléatoire et à un moment aléatoire du chiffrement.

La modification d'un ou plusieurs bits du registre induit un changement dans la caractéristique de parité du ou des états concernés par l'erreur. L'erreur est propagée par les fonctions respectant le code jusqu'au moment de l'utilisation de `ParityCheck`. Ainsi, cette erreur est toujours détectée.

Scénario 2 : un bit aléatoire de la clé ou de la constante de la fonction `AddConstant` est modifié durant un tour aléatoire.

Comme l'opération `XOR` respecte le code parité, l'erreur est propagée dans l'état et persiste jusqu'à la vérification de la caractéristique de parité. Cette erreur est ainsi tout le temps détectée.

Scénario 3 : une faute est injectée dans une valeur utilisée plusieurs fois. L'erreur et l'instant de l'injection sont aléatoires.

Les copies étant vérifiées avant l'utilisation, une différence entre au moins deux des copies sera présente et la faute détectée.

Dans tous les scénarios, la faute injectée est toujours détectée, ainsi la solution basée sur le respect du code est robuste contre les injections sur un registre. 1 000 000 de fautes aléatoires ont été injectées dans chaque scénario, résultant à chaque fois sur une détection de l'injection. Cette robustesse est présentée dans le Tableau 3.4.

TABLE 3.4 – Robustesse de l'implémentation sécurisée.

Nombre d'injections	Scénario 1	Scénario 2	Scénario 3
1 000 000	100% détectées	100% détectées	100% détectées

3.3.2 Surcoût d'implémentation

Ajouter le schéma de parité au chiffrement LED engendre forcément un surcoût. En effet, l'algorithme, qui utilisait originellement des blocs de 128 bits découpés en mots de 4 bits est maintenant transformé en algorithme qui opère sur des mots de 5 bits, avec chaque bit placé dans un registre séparé. Ainsi, les nouvelles fonctions utilisées à chaque tour doivent prendre en compte ce cinquième bit associé à un code de parité et sont donc plus coûteuses que les originales. De plus, plusieurs textes en clair sont chiffrés en parallèle. Pour ce calcul du surcoût, nous avons fixé $l = 64$. Ainsi, l'algorithme utilise 80 registres de 64 bits comme état de chiffrement. Le chiffrement sécurisé est donc plus coûteux en terme de mémoire utilisée et de temps d'exécution. Les résultats d'implémentation sont présentés dans le Tableau 3.5. Dans ce tableau, plusieurs versions du chiffrement sont différenciées : *classique* fait référence à la version originelle de l'implémentation software, utilisant des lookup tables stockées en mémoire ; *bitslice* est la version bitslice non protégée de l'algorithme, chiffrant 64 textes en clair en parallèle ; *code abiding* est la version protégée de l'algorithme, ne faisant pas intervenir les copies des variables utilisées plusieurs fois ; et enfin *code abiding + copies* est la version la plus protégée du chiffrement, utilisant le code de parité ainsi que les copies des variables utilisées plusieurs fois. Les résultats sont présentés sous forme de ratio, afin d'avoir une meilleure comparaison entre les différentes versions des implémentations. Le compilateur utilisé est le GNU GCC Compiler sans aucune optimisation. Le CPU est le Intel Core i5 CPU.

TABLE 3.5 – Surcoût des différentes couches de sécurité.

	Ratio *classique*	Ratio *bitslice*	Ratio *code abiding*
classique	1	-	-
bitslice	1.83	1	-
code abding	2.04	1.12	1
code abiding + copies	3.28	1.79	1.6

Les résultats sont à mettre en perspective avec le nombre de chiffrements effectués en un passage de l'algorithme. En effet, la version *classique* ne chiffre qu'un seul texte en clair quand les autres versions chiffrent 64 textes en parallèle. Ainsi, la version la plus sécurisée de l'algorithme est plus de trois fois plus coûteuse que la version non protégée, mais 64 fois plus de textes en clair sont chiffrés. Il est donc plus intéressant de comparer les versions sécurisées avec la version *bitslice*.

Le surcoût engendré par les protections est ainsi meilleur que prévu. En effet, en comparant la version *bitslice* et la version *code abiding*, nous voyons que le surcoût engendré par l'ajout du bit de parité est de 12%, quand l'ajout d'un cinquième bit sur chaque mot de quatre bits pourrait induire un surcoût de 25% (25% de bits supplémentaires sont pris en compte).
Cependant, en ajoutant les copies des variables utilisées plusieurs fois, ce surcoût passe à 79%. Ainsi, c'est bien cette couche de sécurité qui amène le plus grand surcoût d'implémentation.

De plus, une autre comparaison plus précise est mesurée, cette fois-ci directement au niveau des fonctions d'un tour de chiffrement. Le but est de comprendre quelles fonctions ont subi les transformations les plus coûteuses. Ces résultats sont donnés dans le Tableau 3.6. Ils sont toujours présentés sous forme de ratio, afin de mieux comparer les fonctions entre elles. La première partie du tableau est liée au ratio entre l'implémentation *classique* et les autres implémentations ; la seconde entre l'implémentation *bitslice* non sécurisée et les implémentations bitslice sécurisées ; et enfin la dernière partie dénote l'ajout de la contremesure sur les copies.

TABLE 3.6 – Résultats d'implémentation et comparaison des coûts entre les fonctions du tour.

Ratio *classique*	*bitslice*	*code abiding*	*code abiding + copies*
`AddConstant`	6,6	8,6	8,6
`SubCells`	6,0	7,2	9,6
`ShiftRows`	0,8	0,9	0,9
`MixColumns`	1,9	2,1	3,6

Ratio *bitslice*	*code abiding*	*code abiding + copies*
`AddConstant`	1,3	1,3
`SubCells`	1,2	1,6
`ShiftRows`	1,2	1,2
`MixColumns`	1,1	1,9

Ratio *code abiding*	*code abiding + copies*
`AddConstant`	1,0
`SubCells`	1,3
`ShiftRows`	1,0
`MixColumns`	1,7

En comparant avec la version *classique*, les fonctions qui résultent ayant le plus de surcoût engendré par leur transformations sont celles qui sont passées d'une utilisation de lookup tables à une version sans. Ainsi, ce sont les fonctions `AddConstant` et `SubCells` qui sont les plus coûteuses.

Néanmoins, lorsque la version comparée est la version *bitslice* non protégée, on remarque que ce surcoût est beaucoup plus faible. Par exemple, les résultats passent de 860% d'augmentation à seulement 30% pour la fonction `AddConstant` entre les versions *bitslice* non protégée et *code abiding*.

Enfin, la contremesure agissant sur les copies de variables utilisées plusieurs fois engendre forcément un surcoût important sur les fonctions qui nécessitent le plus de copies différentes des variables. C'est donc logiquement la fonction `MixColumnsSerial` qui est la plus coûteuse, suivie de la fonction `SubCells`. Les deux autres fonctions de tour ne sont pas impactées par cette contremesure, comme aucune copie n'est réalisée lors du déroulé de ces fonctions.

3.4 Conclusion du chapitre

Le principe utilisé dans ce chapitre pour se prémunir des injections de fautes est de détecter leur impact grâce à un code détecteur d'erreur. Le code choisi a été le code de parité. Ce code repose sur une redondance d'information contenue dans un mot

chiffré. Appliqué sur des mots de 4 bits, ce code de parité ajoute un cinquième bit, le bit de parité, permettant de détecter les injections d'erreur sur 1 bit. Les fonctions composant le chiffrement doivent ainsi être adaptées afin de respecter ce code de parité et ainsi ne pas supprimer la détection de l'erreur. Afin de renforcer la sécurité, nous avons décidé d'utiliser une version bitslice du chiffrement, pour se prémunir des injections de fautes sur 1 registre entier plutôt que sur 1 bit.

Deux contremesures ont donc été proposées : la copie des termes utilisés plusieurs fois, et l'adaptation des fonctions du chiffrement pour respecter le code détecteur d'erreur. La première contremesure est appliquée afin d'éviter qu'une injection induise une erreur qui se compense dans une opération, et ainsi devient indétectable. La deuxième contremesure permet de propager une erreur injectée avant ou pendant le déroulé d'une fonction dans l'état de sortie de cette fonction. Ainsi, une erreur injectée à tout instant sera propagée jusqu'à l'utilisation de la procédure de détection.

Les différences entre les coûts d'implémentation ont également été présentées, afin d'évaluer le surcoût engendré par la sécurité.

Quand la majorité des fonctions composant le chiffrement ont été plutôt directes à transformer, la fonctions `SubCells` a engendré une question : quelle représentation sur 5 bits d'une S-box sur 4 bits est la plus efficace ? Cette question est la base du Chapitre 4, qui va présenter une méthode de classification des permutations.

Chapitre 4

Classification des S-boxes

Ce chapitre présente la seconde contribution de cette thèse dont la genèse est issue d'une interrogation apparue lors de la représentation sur 5 bits d'une S-box sur 4 bits, afin d'ajouter le bit de parité comme cela a été présenté dans le Chapitre 3. En effet, l'ajout de ce bit de parité à l'état de chiffrement mène à un choix entre *16 !* représentations sur 5 bits de la S-box originale. Comme montré au Chapitre 2, parmi tous les composants d'un chiffrement symétrique par blocs, la S-box est la fonction non linéaire qui permet d'apporter de la confusion à l'état du chiffrement. Le choix de cette S-box est donc crucial, aussi bien en termes de sécurité que de coût d'implémentation. Ce choix est d'autant plus important lors d'une implémentation matérielle du chiffrement, car la S-box a un fort prix. En effet, son implémentation physique nécessite une grande surface et de nombreux éléments logiques et/ou bits de mémoire. De même pour une version logicielle de l'implémentation, comme par exemple celle proposée par C. Ribeiro [RSD06], pour laquelle la fonction de substitution consomme 75% des cycles d'horloge de l'implémentation bitslice de l'AES.

Pour une S-box manipulant des données de longueur n, le nombre de candidats possibles pour la S-box est de $2^n !$. Afin de limiter le coût d'implémentation, le concepteur du chiffrement par blocs doit intégrer cette réflexion dès la conception. La S-box candidate choisie doit être celle qui combine les meilleures propriétés cryptographiques ainsi que le plus faible coût d'implémentation. Cependant, les candidats étant très nombreux, il est plus judicieux de réduire le nombre de candidats avant de choisir. Pour palier ce problème, des relations d'équivalence sont introduites. Une relation d'équivalence permet de classer des éléments d'un ensemble en classes d'équivalence. Tous les éléments d'une même classe vérifient cette relation d'équivalence entre eux. Ainsi, en choisissant avec soin la relation à appliquer à nos S-boxes candidates, il est possible de les classer en fonction de leurs propriétés cryptographiques ainsi que leur coût d'implémentation. Le but est donc de ne parcourir qu'un élément par classe plutôt que l'ensemble des candidats.

4.1 Propriétés des classes d'équivalence

Durant ces travaux, nous nous sommes demandé comment classer différentes S-boxes selon plusieurs critères. Classer les S-boxes permet de réduire le nombre de permutations à étudier. En effet, en ayant au préalable classé les S-boxes selon des critères précis, un concepteur de primitive cryptographique n'a plus qu'à parcourir un élément de chaque classe d'équivalence pour faire son choix de S-box, plutôt que de parcourir l'ensemble des permutations possibles. Pour ce faire, des relations d'équivalences sont utilisées. Si deux S-boxes vérifient une relation d'équivalence, alors elles appartiennent à la même classe d'équivalence. La Définition 11 donne les critères d'une relation

d'équivalence. Dans notre cas, une relation d'équivalence sera particulièrement utile si elle permet de conserver les mêmes propriétés cryptographiques.

Définition 11 (Relation d'équivalence $\sim$) *Une relation binaire $\sim$ sur un ensemble E est une relation d'équivalence si et seulement si cette relation est réflexive, symétrique et transitive. En d'autres termes, $\forall x, y, z \in E$:*

- *$x \sim x$ (réflexive),*
- *$x \sim y \Leftrightarrow y \sim x$ (symétrique),*
- *$x \sim y$ et $y \sim z \Rightarrow x \sim z$ (transitive).*

L'objectif d'une relation d'équivalence est de partitionner un ensemble en classes d'équivalence (Définition 12) afin de ne travailler que sur un élément par classe plutôt que sur tous les éléments de l'ensemble.

Définition 12 (Classe d'équivalence *[x]*) *La classe d'équivalence de l'élément $x \in E$, selon la relation d'équivalence $\sim$, est notée $[x]$ et est définie comme suit :*

$$\forall a \in E,\ a \in [x] \Leftrightarrow a \sim x$$

Tout élément quelconque de [x] peut être choisi comme le représentant de la classe d'équivalence.

Afin de trier les éléments d'une classe d'équivalence, nous allons utiliser l'ordre lexicographique. Pour définir cet ordre, une nouvelle relation entre les éléments de la classe doit être introduite : l'ordre partiel non strict $\preccurlyeq$, présent plus formellement dans la Définition 13.

Définition 13 (Ordre partiel non strict $\preccurlyeq$) *Une relation binaire $\preccurlyeq$ sur un ensemble E est un ordre partiel non strict si et seulement si cette relation est réflexive, antisymétrique et transitive. En d'autres termes, $\forall x, y, z \in E$:*

- *$x \preccurlyeq x$ (réflexive),*
- *$x \preccurlyeq y$ et $y \preccurlyeq x \Leftrightarrow x = y$ (antisymétrique),*
- *$x \preccurlyeq y$ et $y \preccurlyeq z \Rightarrow x \preccurlyeq z$ (transitive).*

Cette nouvelle relation sera utilisée dans la Définition 14, version plus formelle de l'ordre lexicographique.

Définition 14 (Ordre lexicographique) *L'ordre lexicographique est une généralisation de l'ordre alphabétique appliqué à un ensemble de données. Soit un ensemble d'éléments E, avec $S = (S_1, S_2, \ldots, S_s) \in E$ et $T = (T_1, T_2, \ldots, T_t) \in E$:*

$$S \preccurlyeq T \Leftrightarrow \exists i,\ (i \leq s \text{ et } i \leq t),\ \forall j < i,\ S_j = T_j \text{ et } S_i < T_i$$

Afin de mieux illustrer cette notion, le Tableau 4.1 présente une liste de 10 S-boxes rangées selon l'ordre lexicographique. Ces S-boxes sont représentées par leur expression hexadécimale (d1c3f064985e2ab7 signifie que l'entrée 0 est reliée à la sortie d, l'entrée 1 à la sortie 1, ..., et l'entrée f à la sortie 7).

Une classe d'équivalence sera identifiée avec son représentant. Ainsi, la liste des classes d'équivalence est donnée par la liste des représentants de ces classes. Dans notre cas, le représentant d'une classe d'équivalence est l'élément de la classe qui est le premier dans l'ordre lexicographique.

TABLE 4.1 – S-boxes rangées selon l'ordre lexicographique.

S-boxes dans un ordre aléatoire	S-boxes dans l'ordre lexicographique
d1c3f064985e2ab7	1cf37a2dv604859e
ec8021d6fab37549	346d0e9acbf15487
3d84f7bce5206a91	36eda087429c5f1b
4136f20b958ade7c	3d84f7bce5206a91
5dea0f74368192cb	4136f20b958ade7c
84d9bec60723fa51	5dea0f74368192cb
36eda087429c5f1b	7d2084f6cea5b193
7d2084f6cea5b193	84d9bec60723fa51
346d0e9acbf15487	d1c3f064985e2ab7
1cf37a2dv604859e	ec8021d6fab37549

4.2 Propriétés des boîtes de substitution

Afin de choisir la S-box qui remplira la fonction non linéaire d'un algorithme de chiffrement, cette fonction de substitution doit valider plusieurs propriétés.

Les deux premières propriétés sont la résistance face aux analyses différentielles et linéaires. En effet, depuis les années 1980, des analyses différentielles et linéaires ont été exploitées afin de retrouver la clé de chiffrement d'un algorithme [Bar09 ; CB07 ; Nov05 ; DCBP06]. L'analyse différentielle se base sur la corrélation entre la différence de plusieurs entrées et la différence des sorties correspondantes à ces entrées. Une S-box idéale aurait une corrélation nulle, c'est à dire que quelque soit la différence entre deux entrées de la S-box, la différence entre les sorties correspondantes doit être équiprobable. Ce résultat n'est pas possible, car si la différence entre les entrées est nulle (les deux entrées sont les mêmes) alors la différence entre les sorties sera nulle (les entrées sont reliées aux mêmes sorties), ainsi les différences de sorties ne sont pas équivalentes. Pour l'analyse linéaire, ce sont les relations linéaires entre les entrées et les sorties du chiffrement qui sont exploitées. Ainsi, la S-box, principale fonction de la couche non linéaire de l'algorithme, aura pour rôle de réduire la corrélation linéaire entre le texte d'entrée et le message chiffré. Ces deux analyses appliquées aux différentes S-boxes feront le tri entre les candidates afin de ne garder que celles qui ont de « bonnes propriétés » linéaires et différentielles.

La propriété différentielle d'une S-box S prenant des entrées de tailles n et associant des sorties de tailles m est représentée sur un tableau (nommé *DDT* pour *Difference Distribution Table*) à deux dimensions de tailles $2^n \times 2^m$ dont les coefficients, notés a et b, sont le nombre de solutions de l'équation :

$$S(x + a) + S(x) = b$$

La propriété linéaire de cette même S-box S est représentée sur un tableau (nommé *LAT* pour *Linear Approximation Table*) à deux dimensions de tailles $2^n \times 2^m$ avec des coefficients notés W_S tels que :

$$W_F(a, b) = \sum (-1)^{a \cdot x + b \cdot F(x)}$$

Le coefficient maximal de la table DDT, pour $a \neq 0$ est appelé uniformité différentielle de S et le coefficient maximal de la table LAT, pour $b \neq 0$ est nommé linéarité de S.

La S-box optimale aura une uniformité différentielle et une linéarité les plus petites possibles.

La troisième propriété nécessaire ici est le degré algébrique : chaque S-box peut être ramenée à un ensemble de polynômes reliés à chaque sortie. En effet, chaque bit de sortie peut être exprimé sous la forme d'une combinaison des bits d'entrée. Le degré algébrique de la S-box est est le maximum des degrés des termes qui apparaissent dans la forme algébrique normale (ANF). Ces degrés algébriques (différents pour chaque polynôme), sont importants pour de la cryptanalyse algébrique [Bar09 ; CB07 ; Nov05 ; DCBP06].

Lors de l'optimisation du choix d'une S-box selon un critère précis, il est nécessaire de conserver les propriétés cryptographiques de la S-box originelle. En effet, si en souhaitant réduire le coût d'implémentation d'un chiffrement, le choix de la nouvelle S-box réduit la sécurité du chiffrement, alors ce choix n'est pas le bon. Ainsi, il est nécessaire de trouver des relations d'équivalence entre les permutations qui conservent les propriétés cryptographiques.

4.3 Relations d'équivalence de la littérature

La classification des permutations est un problème déjà étudié dans la littérature, même si le critère d'équivalence est défini, la plupart du temps, par rapport aux propriétés cryptographiques et non le coût d'implémentation.

4.3.1 Équivalence linéaire et affine

La première relation d'équivalence largement étudiée dans la littérature est l'équivalence linéaire, qui se base donc sur des fonctions linéaires [DC07 ; Ull10]. Les fonctions linéaires suivent la Définition 15. La relation est donnée dans la Définition 16.

Définition 15 (Fonction linéaire) *Une application est dite linéaire si et seulement si elle vérifie les propriétés d'additivité et d'homogénéité. Soit une application $A : E \to F$ et un ensemble de scalaires K. L est dite linéaire si et seulement si :*

$$\forall (x, y) \in E^2,\ \forall \lambda \in K,\ L(x \oplus \lambda \cdot y) = L(x) \oplus \lambda \cdot L(y)$$

Définition 16 (Relation d'équivalence linéaire) *Soient deux S-boxes S_1 et S_2. La relation d'équivalence est définie telle que :*

$$S_1 \sim S_2 \Leftrightarrow \exists L_1, L_2\ :\ S_1 = L_1 \circ S_2 \circ L_2$$

Avec L_1 et L_2 deux fonctions linéaires.

Très proche de la relation d'équivalence linéaire, la seconde relation très étudiée est la relation d'équivalence affine, donnée dans la Définition 18. Une fonction affine est une fonction linéaire « à une constante près ». Une fonction affine suit la Définition 17.

Définition 17 (Fonction affine) *Une application A de E dans F est dite affine si et seulement si :*

$$\exists L \text{ linéaire},\ \exists k \in F,\ \forall x \in E,\ A(x) = L(x) \oplus k$$

Définition 18 (Relation d'équivalence affine) *Soient deux S-boxes S_1 et S_2. La relation d'équivalence est définie telle que :*

$$S_1 \sim S_2 \Leftrightarrow \exists A_1, A_2 \ : \ S_1 = A_1 \circ S_2 \circ A_2$$

Avec A_1 et A_2 deux fonctions affines. Une autre approche est d'utiliser des fonctions linéaires, auxquelles on ajoute des constantes c_1 et c_2 :

$$S_1 \sim S_2 \Leftrightarrow \exists L_1, L_2, \ \forall x \in \{0,1\}^n, \ S_1(x) = L_1(S_2(L_2(x) \oplus k_2)) \oplus k_1$$

Avec L_1 et L_2 deux fonctions linéaires et k_1, k_2 deux constantes.

En appliquant ces relations aux S-boxes utilisant des données de taille 1 à 5 bits, nous pouvons réduire le nombre total de classes. Ces résultats sont donnés dans le Tableau 4.2. Le nombre total de permutations est, pour une S-box sur n bits, de $n!$ différentes fonctions.

TABLE 4.2 – Nombres de classes d'équivalence linéaire et affine selon la taille (en bits) des données utilisées. Le pourcentage correspond au nombre de classes en fonction du nombre de permutations pour les valeurs qui ne sont pas appréhensibles facilement.

Taille (en bits)	1	2	3	4	5
Permutations	2	24	40 320	16 ! (100%)	32 ! (100%)
Classes linéaires	2	2	10	52 246 (2.50×10^{-7}%)	2.63×10^{21} (1.00×10^{-12}%)
Classes affines	1	1	4	302 (1.44×10^{-9}%)	2.57×10^{18} (9.77×10^{-16}%)

On voit bien ici l'utilité de réduire le travail sur les représentants des classes plutôt que sur toutes les permutations. En effet, on passe par exemple d'un travail à effectuer sur 16 ! (20 922 789 888 000) permutations à seulement 302 pour les S-boxes sur 4 bits.

L'avantage de ces deux relations d'équivalence est que les propriétés cryptographiques vues précédemment sont maintenues dans les classes. Ainsi, les S-boxes d'une même classe ont ces mêmes propriétés cryptographiques.

4.3.2 Equivalences CCZ et Extended Affine

D'autres relations d'équivalence classiques ont déjà été présentées dans la littérature. La forme la plus générale d'équivalence entre fonctions booléennes qui préserve l'uniformité différentielle est appelée équivalence CCZ [CCZ98 ; CP19]. Elle a été présentée en 1998 par Carlet, Chapin et Zinoviev et est définie par la Définition 19.

Définition 19 (Equivalence CCZ) *Deux fonctions F et G (prenant pour entrées des mots de n bits et donnant en sortie des mots de m bits) sont dites CCZ équivalentes s'il existe une permutation affine A telle que :*

$$\forall x \in \mathbb{F}_2^n, \ F(x) = A \circ G(x)$$

De même, un cas particulier de l'équivalence CCZ est appelé équivalence *Extented Affine* [CP19], présentée dans la Définition 20.

Définition 20 (Equivalence EA)

Comme la relation d'équivalence *Extended Affine* est un cas particulier de la relation d'équivalence CCZ, elle préserve également les propriétés différentielles des fonctions

F et G. De plus, elle préserve le degré algébrique des fonctions, ce qui n'est pas le cas de l'équivalence CCZ.

Ainsi, appliquées à une S-box, ces quatre relations d'équivalence (Linéaire, Affine, CCZ et Extended Affine) préservent les « bonnes » propriétés cryptographiques de la S-box. Cependant, dans un objectif d'optimisation des coûts d'implémentation matériels d'une S-box, nous devons ajouter le critère de coût dans le choix de la couche non linéaire d'un chiffrement. C'est à cette question que répond notre nouvelle relation d'équivalence, baptisée relation d'équivalence *WIRE*.

4.4 Relation d'équivalence WIRE

Dans notre objectif de classification prenant en compte le coût d'implémentation des S-boxes, les équivalences présentées précédemment ne sont plus suffisantes. En effet, dans un contexte originel de respect du code, une opération affine conduit à l'augmentation des ressources logiques nécessaires. Ceci peut être présenté simplement dans la Figure 4.1.

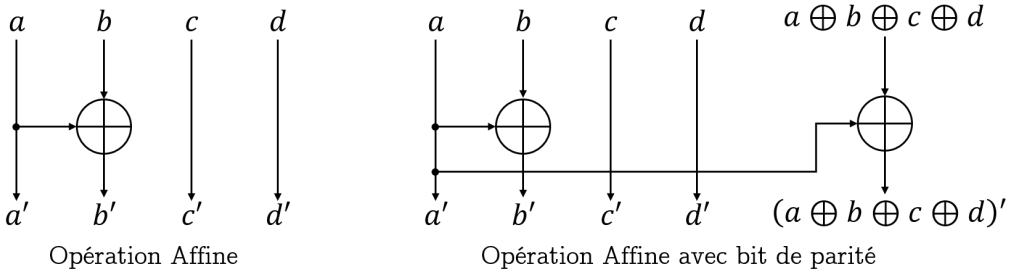

FIGURE 4.1 – Impact de l'implémentation d'une transformation affine dans un contexte de respect du code de parité.

On remarque bien dans cet exemple, très simple, que l'ajout d'une opération affine dans un contexte de bit de parité n'est pas gratuite. En effet, la porte logique XOR présente dans l'opération créant b' doit aussi être ajoutée dans l'opération appliquée au bit de parité. La création d'une nouvelle relation d'équivalence est donc nécessaire.

La relation d'équivalence WIRE est une relation qui part du principe que deux permutations de n bits sur m bits sont dites équivalentes si et seulement si elles sont identiques, à un croisement de fils en entrée et un croisement de fils en sortie près.

4.4.1 Présentation de la relation

Plus formellement, les fonctions suivent la Définition 21.

Définition 21 (Equivalence WIRE) *Deux fonctions F et G (prenant pour entrées des mots de n bits et donnant en sortie des mots de m bits) sont dites WIRE équivalentes si et seulement s'il existe deux croisements de fils $\mathcal{L}_1$ et $\mathcal{L}_2$ tels que :*

$$\forall x \in \mathbb{F}_2^n,\ F(x) = \mathcal{L}_1 \circ G \circ \mathcal{L}_2(x)$$

Un croisement de fils est défini par la Définition 22. Il consiste en l'échange d'au moins deux bits dans un mot. Un exemple d'application d'une fonction croisement de fils est donné dans l'Exemple 4

Définition 22 (Croisement de fils) *Une fonction $\mathcal{L}$ est appelée croisement de fils si, appliquée à un mot x, de longueur n bits, elle permet de mélanger l'ordre des bits du mot sans modifier la valeur de cas bits.*

Exemple 4 (Croisement de fils) *Soit un croisement de fils $\mathcal{L} = [2, 0, 1, 3]$ et un mot x sur n bits tel que $x = x_0||x_1||x_2||x_3 =$ `1001` dans son écriture binaire avec $||$ représentant l'opérateur de concaténation des bits. On a donc $\mathcal{L}(x) = x_2||x_0||x_1||x_3 =$ `0101` sous forme binaire. Ainsi, $\mathcal{L}(9) = 5$.*

Dans le cadre d'une implémentation matérielle, dans laquelle la S-box est décomposée sous forme de portes logiques, effectuer un croisement de fils en entrée (noté $\mathcal{L}_1$) ou en sortie (noté $\mathcal{L}_2$) ne modifie pas le nombre et la nature des portes logiques, ainsi, deux fonctions WIRE équivalentes auront un même coût d'implémentation. Cette équivalence en termes de coût d'implémentation est explicitée sur la Figure 4.2 sur laquelle les deux S-boxes présentées ont le même coût d'implémentation, bien que leur tables respectives (présentées sur le Tableau 4.3) soient différentes. Dans ce tableau, la valeur du mot en entrée est donné par x, décomposé dans ses bits x_0, x_1 et x_2 (x_0 étant le LSB) et la mot de sortie est x', décomposé en x'_0, x'_1 et x'_2. Afin de donner une implémentation matérielle d'une S-box sous forme de portes logiques, c'est la forme ANF (présentée dans le Chapitre 3 dans la Définition 10) qui est utilisée, comportant uniquement des portes logiques AND et XOR.

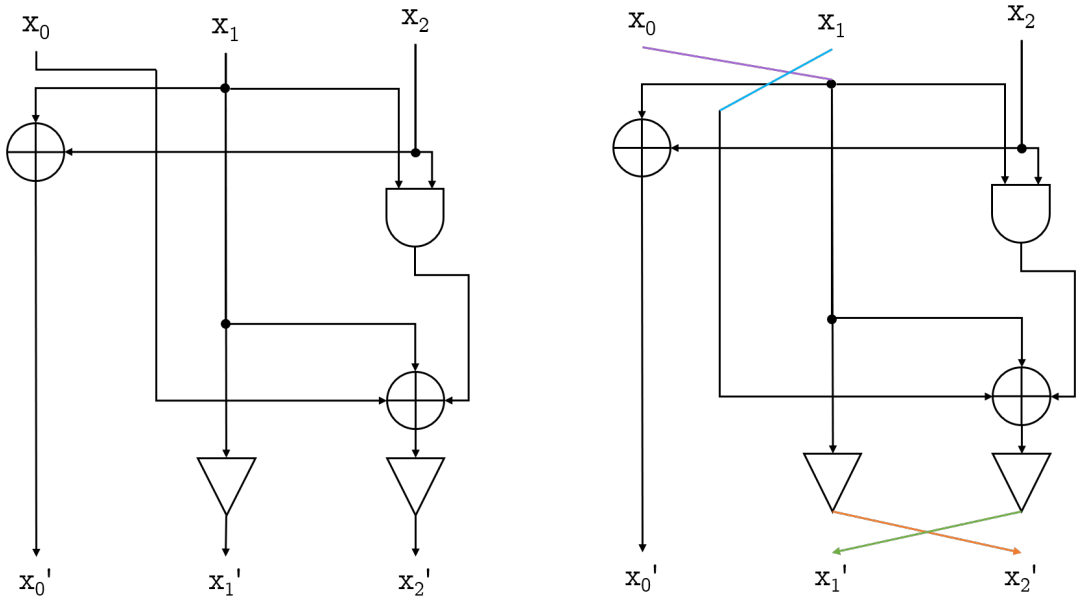

FIGURE 4.2 – Fonctions WIRE équivalentes avec croisement de fils $\mathcal{L}_1 = [0, 1, 2]$ et $\mathcal{L}_2 = [0, 2, 1]$.

Ainsi, les fonctions WIRE équivalentes sont un sous-groupe des permutations Affine équivalentes, elles-mêmes sous-groupe des fonctions bijectives. Cette relation sera utilisée sur les permutations affines afin de classifier efficacement les permutations sur n bits. Nous avons donc besoin d'un algorithme efficace afin de réaliser cette classification. L'algorithme doit prendre en entrée l'ensemble des S-boxes possibles, y appliquer la relation d'équivalence Affine afin de garantir les propriétés cryptographiques, puis appliquer la relation d'équivalence WIRE sur fonctions affines trouvées dans la relation Affine afin d'intégrer le critère du coût d'implémentation. Cela est présenté schématiquement sur la Figure 4.3.

TABLE 4.3 – Tables de vérité de deux fonctions WIRE équivalentes.

S_0							
x	x_2	x_1	x_0	x'	x'_2	x'_1	x'_0
0	0	0	0	6	1	1	0
1	0	0	1	7	1	1	1
2	0	1	0	1	0	0	1
3	0	1	1	4	1	0	0
4	1	0	0	2	0	1	0
5	1	0	1	3	0	1	1
6	1	1	0	5	1	0	1
7	1	1	1	0	0	0	0

S_1							
x	x_2	x_1	x_0	x'	x'_2	x'_1	x'_0
0	0	0	0	6	1	1	0
1	0	0	1	7	1	1	1
2	0	1	0	4	1	0	0
3	0	1	1	5	1	0	0
4	1	0	0	1	0	0	1
5	1	0	1	2	0	1	0
6	1	1	0	3	0	1	1
7	1	1	1	0	0	0	0

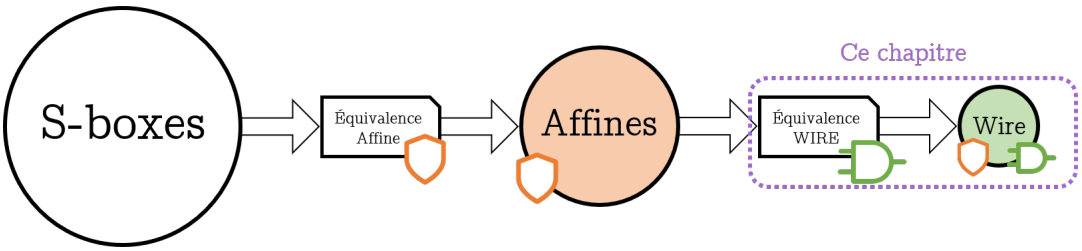

FIGURE 4.3 – Applications successives des relations d'équivalence Affine et WIRE afin de classer les S-boxes

4.4.2 Application de la relation aux S-boxes sur 4 bits

La problématique de la classification des S-boxes selon leur coût est directement issue de l'ajout du bit de parité dans un chiffrement (voir Chapitre 3). Ainsi, la classification des S-boxes sur 4 bits s'impose. La première étape est donc la création des classes d'équivalence, et donc la création de la liste des représentants de ces classes d'équivalence. Cependant, afin d'éviter d'appliquer la relation à chacune des 16 ! permutations sur 4 bits, nous avons décidé d'appliquer en premier lieu la relation d'équivalence affine, présentée dans [DC07]. Ainsi, les fonctions sont réparties dans les 302 classes d'équivalence affine, et pour rappel, deux fonctions F et G sont dites affines équivalentes si et seulement s'il existe deux fonctions affines A_1 et A_2 telles que $F = A_1 \circ G \circ A_2$. La relation d'équivalence WIRE ne sera donc appliquée qu'à ces deux fonctions affines A_1 et A_2, plutôt qu'à l'ensemble des permutations sur 4 bits.

Les représentants de nos classes d'équivalence seront donc choisis parmi les 322 560 fonctions affines sur 4 bits. Afin de faciliter la création de la liste de ces représentants, les fonctions affines sont, dans un premier temps, classées selon l'ordre lexicographique (voir Définition 14). Ainsi, en donnant un index à chacune des fonctions affines (0 étant l'index de la première fonction affine dans l'ordre lexicographique), si l'application de deux croisements de fils en entrée et en sortie permet d'atteindre une fonction dont l'index est plus petit que la fonction de départ, c'est que le représentant de la classe d'équivalence de cette dernière a déjà été choisi. Cette méthode permet de réduire énormément le temps d'exécution de la création de la liste des représentants. Cette création est donnée dans l'Algorithme 17.

L'exécution de cet algorithme renvoie une liste de représentants, dont le nombre ainsi que la durée de création sont donnés dans le Tableau 4.4.

Algorithm 17 Création de la liste des représentants de la relation d'équivalence WIRE classés dans l'ordre lexicographique.

Require: Liste des permutations affines $\texttt{Affine}_{\texttt{list}}$ dans l'ordre lexicographique ainsi que la liste des croisements de fils `crossings`
Ensure: Liste des représentants de la relation d'équivalence WIRE $\texttt{R}_{\texttt{list}}$

```
R_list = [ ]
for A ∈ Affine_list do
    is_representative = true
    for (L1, L2) ∈ crossings × crossings do
        f = L1 ∘ A ∘ L2
        if index(f) < index(A) then
            is_representative = false
            break
    if is_representative then
        R_list.append(A)
return R_list
```

Les croisements de fils peuvent être appliqués aux quatre bits d'entrée et de sortie de la fonction. Cependant, avec le contexte liant un cinquième bit, le bit de parité, chacun des bits d'entrée ou de sortie peut également être échangé avec ce bit de parité. Ainsi, par exemple, le bit x_0 peut être échangé avec le bit x_1, x_2, x_3, le bit de parité $x_0 \oplus x_1 \oplus x_2 \oplus x_3$, ou ne pas être échangé. Les croisements de fils sont donc donnés sous la forme d'une combinaison de 4 éléments de la liste [0,1,2,3,4], où la valeur 4 représente le bit de parité du mot.

Cette création a été réalisée par une implémentation Python qui s'exécute sur un Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz avec 64Go de mémoire RAM. La création des classes depuis les permutations linéaires est beaucoup plus rapide, mais une fois ce délai effectué une fois, le gain durant la recherche du représentant d'une permutation quelconque entre les classes issues des permutations linéaires et celles issues des permutations affines est négligeable.

Table 4.4 – Nombre de classes de la relation d'équivalence WIRE.

	Linéaires	**Affines**
Nombre de fonctions	20 160	322 560
Nombre de classes avec croisements de fils sur 4 bits	51 (7s)	671 (112s)
Nombre de classes avec croisement de fils sur 4+1 bits	6 (8s)	43 (61s)

L'algorithme a été appliqué aux fonctions affines ainsi qu'aux fonctions linéaires. Par exemple, chaque fonction affine est rangée dans une des 43 classes d'équivalence WIRE. Chaque classe comporte toutes les fonctions affines ayant le même coût d'implémentation en termes de nombre de portes logiques, à un croisement de fils près (incluant le croisement de fils avec le bit de parité). Il faut maintenant trouver un moyen d'appliquer la relation d'équivalence WIRE à une permutation quelconque, plutôt que seulement aux fonctions affines ou linéaires. Ainsi, le protocole à suivre doit intégrer l'application de la relation d'équivalence Affine ainsi que la relation d'équivalence WIRE.

La relation d'équivalence générale appliquée à une S-box S est donc composée de plusieurs étapes :

- Application de la relation d'équivalence Affine sur la S-box S afin de retrouver A_1, A_2 et R_A tels que $S = A_1 \circ R_A \circ A_2$
  - R_A est appelé représentant affine de S
  - A_1 et A_2 sont des fonctions affines
- Application de la relation d'équivalence WIRE sur A_1 pour retrouver $\mathcal{L}_1$, $\mathcal{L}_2$ et R_{W_1} tels que $A_1 = \mathcal{L}_1 \circ R_{W_1} \circ \mathcal{L}_2$
  - R_{W_1} est appelé représentant WIRE de A_1
  - $\mathcal{L}_1$ et $\mathcal{L}_2$ sont des fonctions croisements de fils
- Application de la relation d'équivalence WIRE sur A_2 pour retrouver $\mathcal{L}_3$, $\mathcal{L}_4$ et R_{W_2} tels que $A_2 = \mathcal{L}_3 \circ R_{W_2} \circ \mathcal{L}_4$
  - R_{W_2} est appelé représentant wire de A_2
  - $\mathcal{L}_3$ et $\mathcal{L}_4$ sont des fonctions croisements de fils
- La relation générale est donc donnée par $S = \mathcal{L}_1 \circ R_{W_1} \circ \mathcal{L}_2 \circ R_A \circ \mathcal{L}_3 \circ R_{W_2} \circ \mathcal{L}_4$

Durant ses recherches, De Cannière [DC07] a listé les 302 classes d'équivalence affine pour les permutations sur 4 bits. De plus notre côté, nous avons obtenu 6 classes d'équivalence (pour les croisements de fils tenant compte du bit de parité) à partir des 20 160 permutations linéaires. Ainsi, le nombre de classes s'élève à $6 \times 302 \times 6 = 10\ 872$. La même décomposition peut être appliquées aux fonctions affine, résultant en $51 \times 302 \times 51 = 785\ 502$ classes.

Nous pouvons noter que ce nombre peut sembler élevé. Cependant, pour un concepteur d'implémentation, toutes les permutations ne seront pas intéressantes. Ainsi, le concepteur peut, en premier lieu, choisir les propriétés cryptographiques voulues, et ainsi réduire les 302 classes affines en un nombre réduit de classes ayant ces « bonnes » propriétés cryptographiques. Dans un second temps, il peut ensuite réduire ces classes affines selon leur coût avec la relation d'équivalence WIRE. Grâce à une approche « diviser pour mieux régner », il peut donc réduire grandement la liste des potentielles S-boxes candidates à analyser avant de faire son choix.

Ces étapes ont été appliquées à la S-box de l'algorithme PRESENT, et les différentes valeurs (intermédiaires et finales), sont données dans le Tableau 4.5.

TABLE 4.5 – Relation d'équivalence WIRE appliquée à la S-box `PRESENT`.

S	[C 5 6 B 9 0 A D 3 E F 8 4 7 1 2]
A_1	[8 9 7 6 0 1 F E B A 4 5 3 2 C D]
A_2	[C 9 3 6 1 4 E B A F 5 0 7 2 8 D]
R_A	[0 1 2 3 4 6 8 A 5 B C F E D 9 7]
$\mathcal{L}_1$	[3 0 1 2]
$\mathcal{L}_2$	[2 0 3 4]
R_{W_1}	[1 0 2 3 6 7 5 4 F E C D 8 9 B A]
$\mathcal{L}_3$	[4 3 1 2]
$\mathcal{L}_4$	[1 2 0 3]
R_{W_2}	[1 2 4 7 6 5 3 0 A 9 F C D E 8 B]

4.4.3 Exemple appliqué

Dans cette partie, nous allons montrer que la relation d'équivalence WIRE peut réellement permettre de réduire les coûts d'implémentation avec un exemple sur deux S-boxes ayant les mêmes propriétés cryptographiques (dans notre cas, elles appartiennent à la même classe d'équivalence Affine) mais pas les mêmes coûts d'implémentation. Grâce à la relation d'équivalence WIRE, il est donc possible de classer les S-boxes à l'intérieur d'une classe d'équivalence Affine pour sélectionner la moins coûteuse.

La première S-box est celle utilisée jusque là dans les précédents exemples, celle de l'algorithme `PRESENT`, qui est notée S_1 dans le Tableau 4.6. La seconde est une S-box dans la classe d'équivalence Affine de S_1, notée S_2 dans le tableau. S_1 a un score de 46, donc 46 portes logiques `AND` et `XOR` sont nécessaires à son implémentation, quand S_2, ayant les mêmes propriétés cryptographiques que S_1, a un score de 35.

TABLE 4.6 – S-boxes ayant les mêmes propriétés cryptographiques mais avec un coût d'implémentation différent.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_1(x)$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
$S_2(x)$	A	1	4	D	0	B	6	9	3	E	F	7	8	5	C	2

4.5 Conclusion du chapitre

Dans ce chapitre, une nouvelle relation d'équivalence a été introduite. Cette relation est issue de la relation d'équivalence Affine et a pour objectif de classifier les permutations selon leurs propriétés cryptographiques ainsi que leur coût d'implémentation.

La relation d'équivalence Affine conserve les propriétés cryptographiques (linéarité et différentiabilité) d'une S-box. De même, la relation d'équivalence WIRE étant un cas particulier de la relation Affine, elle garde également ces propriétés cryptographiques. Ainsi, deux fonctions WIRE équivalentes ont les mêmes propriétés cryptographiques ainsi que les mêmes coûts d'implémentation. Cette relation permet donc une classification des permutations sur n bits selon leurs propriétés cryptographiques ainsi que leur coût d'implémentation.

En utilisant la relation d'équivalence Affine, un concepteur de primitive cryptographique peut sélectionner les propriétés cryptographiques voulues, afin de garantir une haute sécurité de la couche non linéaire de son algorithme. Il va ensuite choisir, parmi les représentants des classes d'équivalence de la relation WIRE issues de l'équivalence Affine, la permutation ayant un coût d'implémentation plus faible pour limiter ce coût.

Nous pouvons noter que dans ce chapitre, le coût d'implémentation d'une permutation est représenté par le nombre de portes logiques nécessaires à la création de la forme ANF de cette permutation (c'est l'idée du score d'une S-box qui est présenté dans le Chapitre 3). Cependant, pour un concepteur ayant d'autres contraintes que le nombre de portes logiques, un autre critère peut permettre de séparer les fonction affines en différentes classes d'équivalence. Ainsi, ces nouvelles classes permettraient de diviser les permutations en différentes classes en suivant le même procédé : première séparation qui utilise la relation d'équivalence Affine afin de garantir les propriétés

cryptographiques, puis seconde séparation basée sur une relation d'équivalence liée au nouveau critère à évaluer. Cette méthode est donc généralisable à d'autres critères que le nombre de portes logiques nécessaire à l'implémentation matérielle d'une S-box.

Chapitre 5

BALoo

Ce chapitre présente la dernière contribution de cette thèse qui est une contremesure dédiée à l'analyse des fautes persistantes. En effet, comme présenté dans le Chapitre 2, en 2018, Zhang *et al.* [Zha+18] ont proposé une nouvelle analyse basée sur des fautes persistantes. Pour rappel, les fautes persistantes sont des fautes injectées dans une mémoire non volatile, afin de modifier une donnée de cette mémoire de façon persistante, ou plus précisément jusqu'à la réécriture de cette mémoire. Cette analyse, appelée Persistent Fault Analysis (ou PFA) a été détaillée dans le Chapitre 2. Le but de cette analyse est d'exploiter une faute injectée dans la S-box d'un chiffrement stockée en mémoire, afin de supprimer une valeur possible. En exploitant le biais dans les probabilités d'apparition des octets dans le message chiffré, l'adversaire peut récupérer de l'information sur la clé de chiffrement. En effet, en simplifiant le dernier tour du chiffrement à simplement sa couche de substitution et l'intégration de la clé, alors un octet n'apparaîtra jamais dans le message chiffré. Cet octet est lié à la valeur de S-box qui a été fautée, et qui donc n'apparaît jamais non plus. En effectuant une opération de `Xor` entre l'octet n'apparaissant plus et la valeur de S-box qui n'apparaît plus, l'adversaire peut retrouver l'octet de la clé qui a été intégrée.

Le but de la contremesure est donc de détecter si une faute a été injectée dans la S-box du chiffrement. Afin de détecter une erreur, plusieurs propositions ont déjà été présentées dans la littérature. Le Chapitre 3 est un exemple de vérification d'intégrité du chiffrement avec un code détecteur d'erreur appliqué à l'ensemble de l'état du chiffrement. Cependant, ici, le modèle de l'attaquant est différent. En effet, le but de l'adversaire est d'injecter une faute dans la S-box, et pas dans un endroit aléatoire de l'état. Ainsi, il ne sert à rien, dans notre modèle, de protéger l'entièreté du chiffrement, mais plutôt de simplement s'assurer de l'intégrité de la S-box.

Depuis la standardisation de l'AES en 2001, la plupart des chiffrements par blocs, légers ou non, utilisent une S-box bijective pour leur couche de substitution. Ainsi, nous allons utiliser cet aspect de permutation (bijection) afin de vérifier ses propriétés intrinsèques.

5.1 Propriétés des permutations

Dans ce manuscrit, une permutation est définie comme une fonction bijective d'un ensemble dans un autre (qui peut être lui-même). Dans notre cas, l'ensemble de départ et d'arrivée est le même : $\mathbb{F}_2^n$, soit l'ensemble des mots de n bits.

Afin de tester l'intégrité d'une S-box, plusieurs solutions issues des propriétés de l'aspect permutation de la S-box peuvent être utilisées. Par la suite, S_i fera référence à la sortie d'indice i de la S-box.

5.1.1 Propriétés intrinsèques

Propriété 1 (Addition) *La somme de toutes les valeurs de la S-box est connue. En effet, pour une S-box ayant des données de taille n, la S-box comprend toutes les valeurs entre 0 et $(2^n - 1)$. Ainsi, la somme de toutes ces valeurs est égale à :*

$$\sum_{i=1}^{2^n-1} S_i = \sum_{k=1}^{2^n-1} k = \frac{2^n \times (2^n - 1)}{2}$$

Par exemple sur la S-box de l'AES qui comprend des valeurs sur 8 bits, la somme est égale à 32 640. La valeur 32 640 doit donc être pré-calculée, stockée et comparée avec la somme actuelle des valeurs de S-box. Cela peut engendrer un problème lors du calcul : sur un microcontrôleur fonctionnant sur 8 bits, la somme dépasse la valeur maximale codable en binaire avec 8 bits, et donc il faut prendre en compte ce dépassement.

Propriété 2 (Somme binaire) *La somme binaire de tous les éléments de la S-box est également connue. Sur les permutations utilisées dans les algorithmes de chiffrement, cette somme est même égale à :*

$$S_0 \oplus S_1 \oplus \cdots \oplus S_i \oplus \cdots \oplus S_{n-1} = 0$$

Pour la S-box de l'AES, la somme binaire de tous les éléments est égale à 0. En effet, dans une S-box, toutes les valeurs et leur complément à 1 sont comprises, ainsi la somme binaire fait donc 0.

Propriété 3 (Cycles) *Toutes les permutations se décomposent en produit de transpositions, nommées cycles. Un cycle est construit par récurrence en prenant une valeur initiale arbitraire. La seconde valeur est la sortie d'indice de la valeur initiale. De même pour les valeurs suivantes, construites par récurrence jusqu'à retrouver la valeur initiale. Plus formellement soit une valeur initiale i d'une S-box S, les valeurs j appartiennent au cycle contenant i avec $j = S_j$ jusqu'à retrouver $S_j = i$. Chaque cycle a une longueur précise, et la somme de toutes les longueurs des cycles est égale au nombre de valeurs de la permutation, donc 2^n. Afin de représenter la S-box S, il suffit de prendre un élément arbitraire de chaque cycle comme représentant de ce cycle. La S-box est donc représentée par une liste de représentants de chaque cycle.*

L'Algorithme 18 montre la création de l'ensemble des cycles. La dernière propriété peut sembler beaucoup plus compliquée que les deux autres, mais elle a l'avantage d'être une propriété bijective. En effet, la construction des cycles est unique et exclusive pour chaque permutation. Ainsi, chaque permutation n'a qu'une seule représentation sous forme de cycles, et chaque représentation sous forme de cycles n'est associée qu'à une seule permutation. Ce n'est pas le cas des deux autres propriétés, comme cela sera montré par la suite.

La troisième propriété sera nommée BALoo, pour *Bijection Assert with Loops* et peut être utilisée sous deux versions. La première version va permettre de déterminer l'injection d'une erreur dans la S-box en se reposant sur la somme de la longueur des cycles ainsi que leur nombre (calcul séquentiel de la longueur des cycles et des indices de départ des cycles comme présenté dans l'Algorithme 18). La seconde version détecte les injections en comparant individuellement les longueurs des cycles associées à leur indice de départ. Ces indices ainsi que les longueurs individuelles doivent donc être calculés en amont.

Algorithm 18 Création des cycles d'une permutation.

Require: Permutation S
Ensure: Représentation sous forme de cycles de cette permutation

```
function isIn(i, loops)        ▷ Détermine si une valeur est déjà dans un cycle créé
    isin ← False
    for l ∈ loops do
        isin ← isin ∨ (i ∈ l)
    return isin

function CONSTRUCTION(S)
    loops ← [ ]
    for i ∈ [0, len(S)] do
        if not(isIn(i, loops)) then
            l ← [ ]
            l.append(i)
            while S[l[−1]] ≠ i  &  len(l) ≤ len(S) do
                l.append(S[l[−1]])
            loops.append(l)
    return loops
```

L'avantage de la première version est que la seule donnée redondante qui doit être stockée dans la mémoire est le nombre de cycles. En effet, les cycles et leur indice de départ sont créés directement lors de l'application de la contremesure. Dans la seconde version, c'est l'indice de départ ainsi que la longueur de chaque cycle qui doivent être stockés. La première version est donc moins sensible à des potentielles attaques sur la S-box couplées à des injections sur les données redondantes. La seconde version est plus efficace comme les indices sont déjà donnés, notamment dans une version parallélisée de la vérification des longueurs individuelles de cycle. En effet, en ayant les indices de départ et les longueurs de chacun des cycles, il est aisé de paralléliser la vérification de chacun des cycles, quand la première version doit attendre la fin de création d'un cycle pour trouver le futur indice de départ et créer le prochain cycle.

5.1.2 Application sur un exemple

Ces trois propriétés vont être appliquées dans un exemple sur la permutation présentée sur le Tableau 5.1, qui est celle utilisée pour l'algorithme PRESENT [Bog+07a].

TABLE 5.1 – PRESENT S-box.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Propriété 1 (Addition). La somme des éléments de cette permutation sur 4 bits est :

$$\sum_{k=1}^{15} k = \frac{16 \times 15}{2} = 120$$

Propriété 2 (Somme binaire). La somme binaire des éléments de cette permutation sur 4 bits est :

$$0 \oplus 1 \oplus \cdots \oplus 15 = 0$$

Propriété 3 (Cycles). La S-box PRESENT peut être assimilée à 4 cycles, présentés sur la Figure 5.1. En effet, pour le premier cycle, on a bien $S_0 = 12$, $S_{12} = 4$, ..., $S_5 = 0$. Ces cycles sont respectivement de longueurs 7, 4, 3 et 2. On a bien la somme des longueurs égale au nombre de valeurs, ici cette somme est bien égale à $7 + 4 + 3 + 2 = 16$.

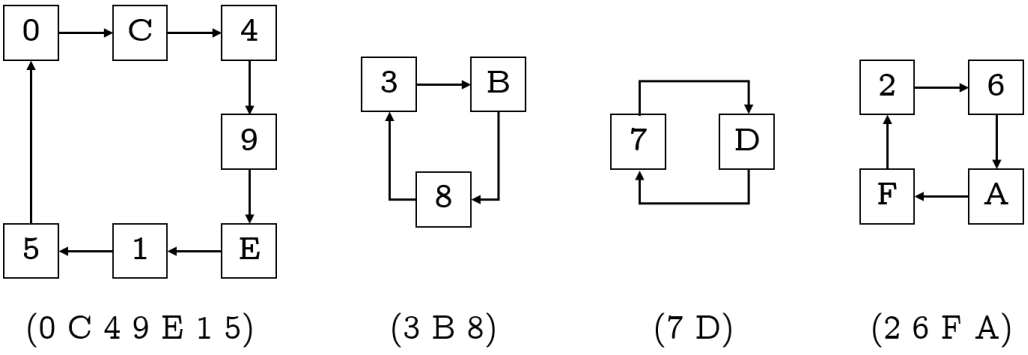

FIGURE 5.1 – Cycles de la S-box PRESENT.

Le but est donc de comparer les valeurs actuelles des propriétés avec celles qui devraient être les vraies valeurs sans faute injectée. Si une faute est injectée, alors les propriétés seront modifiées et la faute détectée. Cependant, quand plusieurs fautes sont injectées, elles peuvent ne pas être détectées. Plusieurs exemples vont être présentés afin de mettre en lumière ces cas.

5.1.3 Exemple de détection

Une faute est injectée sur la S-box précédente, alors sa distribution de probabilité des octets en sortie est biaisée. Cette S-box biaisée est présentée dans le Tableau 5.2, avec S la S-box originale et S^1 la S-box fautée.

TABLE 5.2 – S-box S^1.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S^1[x]$	C	5	6	B	9	0	A	D	3	1	F	8	4	7	1	2

Addition. La somme des éléments de S^1 est :

$$12 + 5 + 6 + 11 + 9 + 0 + 10 + 13 + 3 + 1 + 15 + 8 + 4 + 7 + 1 + 2 = 107 \neq 120$$

La somme trouvée est différente de l'originale, donc cette solution détecte la faute.

Somme binaire. La somme binaire des éléments de S^1 est :

$$12 \oplus 5 \oplus 6 \oplus 11 \oplus 9 \oplus 0 \oplus 10 \oplus 13 \oplus 3 \oplus 1 \oplus 15 \oplus 8 \oplus 4 \oplus 7 \oplus 1 \oplus 2 = 15 \neq 0$$

La somme binaire trouvée est différente de l'originale, donc cette solution détecte la faute.

BALoo. La nouvelle construction sous forme de cycles de cette S-box est donnée dans la Figure 5.2. Avec E en valeur initiale du premier cycle, il est impossible de retrouver cette valeur, car la sortie liée à l'indice 9 est maintenant de 1 et non de E. Ainsi, la longueur de ce cycle, avec E en valeur initiale, est infinie. De même, en prenant une autre valeur quelconque de ce cycle, le nouveau cycle est de longueur 6 plutôt que 7. La somme des longueurs des cycles est donc infinie (donc supérieure à 16). La faute est donc détectée.

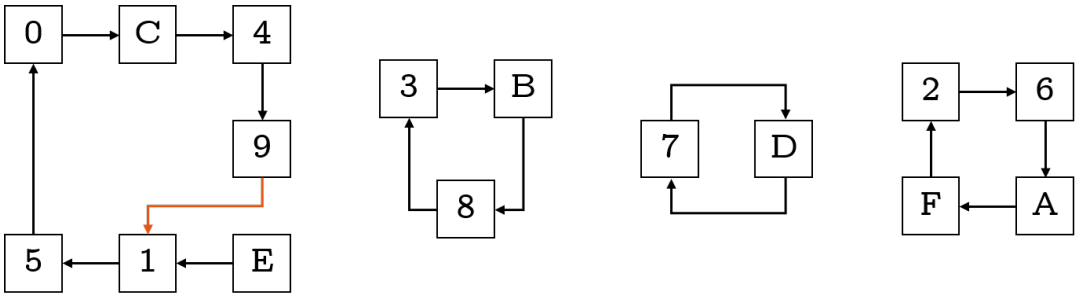

FIGURE 5.2 – Cycles de S^1.

5.2 Solutions contournées

La section précédente a présenté un exemple de faute injectée sur une S-box qui a mené à la détection de cette injection par les trois solutions. Nous allons voir ici des cas qui vont permettre de contourner les solutions, et si ce contournement peut amener à des fuites sur la clé.

5.2.1 Non détection par l'addition

Soit la S-box S^2 fautée présentée dans le Tableau 5.3. La première injection qui transforme C en 8 fait apparaître un « 0 » au bit d'indice 2, la seconde injection qui transforme 5 en 7 fait apparaître un « 1 » au bit d'indice 1 et la dernière injection qui transforme 9 en B fait apparaître un « 1 » au bit d'indice 1. Les trois impacts sont donc différents. Voyons comment les solutions peuvent détecter les injections réalisées.

TABLE 5.3 – S-box S^2.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S^2[x]$	8	7	6	B	B	0	A	D	3	E	F	8	4	7	1	2

Addition. La somme des éléments de S^2 est :

$$8+7+6+11+11+0+10+13+3+14+15+8+4+7+1+2 = 120 = 120$$

La somme trouvée est identique à l'originale, donc cette solution ne détecte pas la faute.

Somme binaire. La somme binaire des éléments de S^2 est :

$$8 \oplus 7 \oplus 6 \oplus 11 \oplus 11 \oplus 0 \oplus 10 \oplus 13 \oplus 3 \oplus 14 \oplus 15 \oplus 8 \oplus 4 \oplus 7 \oplus 1 \oplus 2 = 4 \neq 0$$

La somme binaire trouvée est différente de l'originale, donc cette solution détecte la faute.

BALoo. La nouvelle construction sous forme de cycles de cette S-box est donnée dans la Figure 5.3. Elle est complètement différente de l'originale, avec des cycles infinis et modifiés, donc la faute est bien détectée.

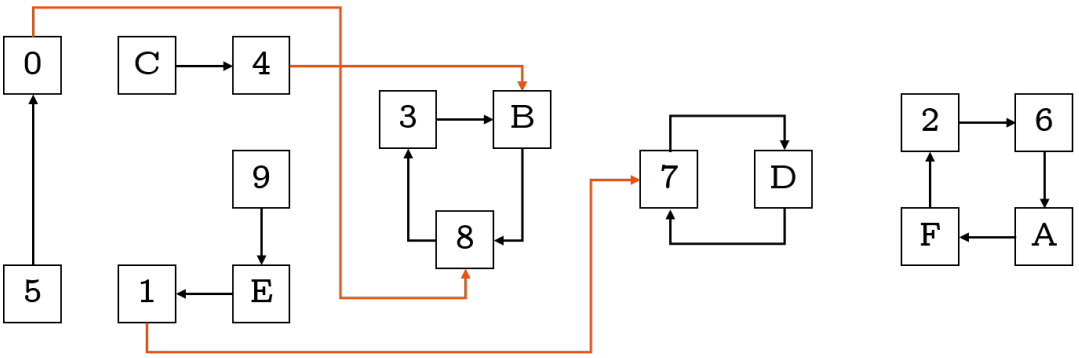

FIGURE 5.3 – Cycles de S^2.

5.2.2 Non détection par la somme binaire

Soit la S-box S^3 fautée présentée dans le Tableau 5.4. La première injection qui transforme le C en D fait intervenir apparaître un « 1 » au bit de poids faible et la seconde injection fait apparaître un « 1 » également au bit de poids faible. Nous avons donc le même impact pour les deux injections. Voyons comment les solutions peuvent détecter les injections réalisées.

TABLE 5.4 – S-box S^3.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S^3[x]$	D	5	7	B	9	0	A	D	3	E	F	8	4	7	1	2

Addition. La somme des éléments de S^3 est :

$$13 + 5 + 7 + 11 + 9 + 0 + 10 + 13 + 3 + 14 + 15 + 8 + 4 + 7 + 1 + 2 = 122 \neq 120$$

La somme trouvée est différente de l'originale, donc cette solution détecte la faute.

Somme binaire. La somme binaire des éléments de S^3 est :

$$13 \oplus 5 \oplus 7 \oplus 11 \oplus 9 \oplus 0 \oplus 10 \oplus 13 \oplus 3 \oplus 14 \oplus 15 \oplus 8 \oplus 4 \oplus 7 \oplus 1 \oplus 2 = 0 = 0$$

La somme binaire trouvée est identique à l'originale, donc cette solution ne détecte pas la faute.

BALoo. La nouvelle construction sous forme de cycles de cette S-box est donnée dans la Figure 5.4. Elle est complètement différente de l'originale, avec des cycles infinis et modifiés, donc la faute est bien détectée.

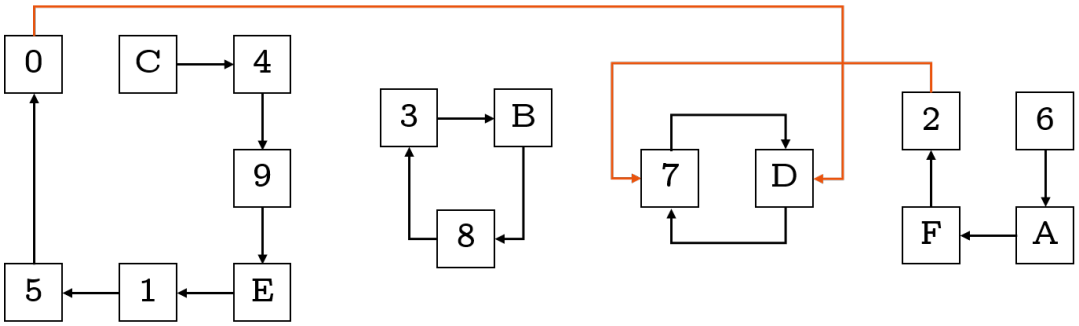

FIGURE 5.4 – Cycles de S^3.

5.2.3 Non détection par BALoo

Soit la S-box S^4 fautée présentée dans le Tableau 5.5. La première injection qui transforme 9 en E fait apparaître un « 1 » au bit d'indice 1 et un « 0 » au bit de poids faible ; la seconde injection qui transforme E en 4 fait apparaître deux « 0 » aux bits d'indice 1 et 3 ; et enfin la dernière injection qui transforme 4 en 9 fait apparaître un « 1 » aux bits d'indice 0 et 3 et un « 0 » au bit d'indice 2. Les impacts sont donc très différents les uns des autres. Voyons comment les solutions peuvent détecter les injections réalisées.

TABLE 5.5 – S-box S^4.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S^4[x]$	C	5	6	B	E	0	A	D	3	4	F	8	9	7	1	2

Addition. La somme des éléments de S^4 est :

$$12 + 5 + 6 + 11 + 15 + 0 + 10 + 13 + 3 + 4 + 15 + 8 + 9 + 7 + 1 + 2 = 120 = 120$$

La somme trouvée est identique à l'originale, donc cette solution ne détecte pas la faute.

Somme binaire. La somme binaire des éléments de S^4 est :

$$12 \oplus 5 \oplus 6 \oplus 11 \oplus 15 \oplus 0 \oplus 10 \oplus 13 \oplus 3 \oplus 4 \oplus 15 \oplus 8 \oplus 9 \oplus 7 \oplus 1 \oplus 2 = 0 = 0$$

La somme binaire trouvée est identique à l'originale, donc cette solution ne détecte pas la faute.

BALoo. La nouvelle construction sous forme de cycles de cette S-box est donnée dans la Figure 5.5. Elle est différente de l'originale, mais les longueurs individuelles des cycles, ainsi que la longueur totale, sont identiques. Quelle que soit la méthode utilisée par BALoo (création en directe des cycles, ou utilisation des valeurs initiales pré enregistrées), le nombre de cycles et leur longueur sont identiques. Ainsi ces injections ne sont pas détectées.

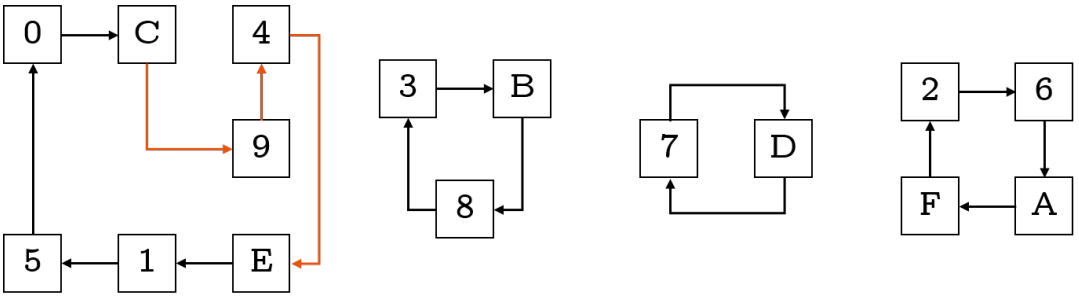

FIGURE 5.5 – Cycles de S^4.

Les différentes cas de détection sont présentés sur le Tableau 5.6. Les trois solutions permettent de détecter une injection sur un octet unique de la S-box, et ainsi contrer efficacement la PFA. Néanmoins, la solution d'addition et la solution de somme binaire peuvent être contournées, et une PFA peut être appliquée. Ce contournement nécessite une injection de multiples fautes, donc l'analyse sera moins efficace, mais pourra quand même grandement réduire le nombre de candidats de clé.

On voit que la contremesure BALoo est plus performante que les deux autres solutions, et que contourner cette contremesure nécessite au minimum 3 injections très précises, mêlant des impacts de BITSET (bit ciblé forcé à 1) et des impacts de BITRESET (forçant le bit ciblé à 0). Néanmoins, ces injections ayant un impact différent sur la mémoire sont un challenge très difficile pour l'attaquant. En effet, la majorité des attaques dans la littérature visant à injecter une faute dans la mémoire non volatile ont pour impact de décharger le transistor qui stock la donnée. Ainsi, si à l'état déchargé, le transistor mémorise la valeur « 1 », alors l'attaquant ne pourra injecter que des 1 dans la mémoire. Avec la contremesure BALoo, il ne pourra donc pas compenser son injection avec d'autres afin de contourner la détection.

TABLE 5.6 – Détection des différentes solutions en fonction du nombre de fautes injectées et du modèle de faute nécessaire au contournement de la détection.

Solution	Détection d'une injection unique	Non détection	
		Minimum de fautes	Impact
Addition	✓	2	BITSET et BITRESET
Somme binaire	✓	2	BITSET et/ou BITRESET
BALoo	✓	3	BITSET et BITRESET

De plus, dans le cas où un attaquant théorique pourrait contourner la contremesure BALoo, la S-box qu'il atteindrait serait toujours une permutation. En effet, la Propriété 3, qui présente la construction sous forme de cycles d'une S-box, est bijective avec la propriété de permutation. Ainsi, la nouvelle S-box, bien que fautée, n'admet pas de biais dans la distribution de probabilité d'apparition de ses octets en sortie. L'adversaire ne peut donc pas exploiter ce biais, comme aucune valeur n'apparaît plus et aucune valeur apparaît plus souvent que les autres. Par conséquent, l'analyse des fautes persistantes PFA ne peut pas être appliquée sur une telle S-box. La contremesure BALoo atteint donc une couverture de fautes de 100% contre les injections sur lesquelles se base un attaquant souhaitant utiliser la PFA.

5.3 Robustesse

Afin de mieux comparer les solutions entre elles, une étude de leur robustesse a été réalisée. Deux modèles d'injection vont être différenciés. Le premier est le modèle le plus réaliste du point du vue de l'attaquant, bien que plus limité. Ce modèle restreint les impacts possibles des fautes injectées à seulement des BITSET ou seulement des BITRESET. Le second modèle, offrant plus de liberté à l'adversaire, combine les deux impacts BITSET et BITRESET pour donner l'impact BITFLIP. Ce modèle est plus puissant pour l'adversaire mais reste dans le cadre théorique car combiner les deux impacts sur une mémoire non volatile est très challengeant pour l'attaquant. Nous voulons néanmoins estimer la robustesse des solutions face à ces deux modèles de faute.

Pour chacun des modèles de faute et chaque nombre de fautes, nous avons décidé de réaliser 10 000 000 injections aléatoires. Les injections sont simulées afin de s'assurer de leur impact sur la S-box. Le chiffrement visé est l'AES.

5.3.1 Modèle BITSET/BITRESET

La Figure 5.6 montre la couverture des fautes pour le premier modèle (BITSET/BITRESET).

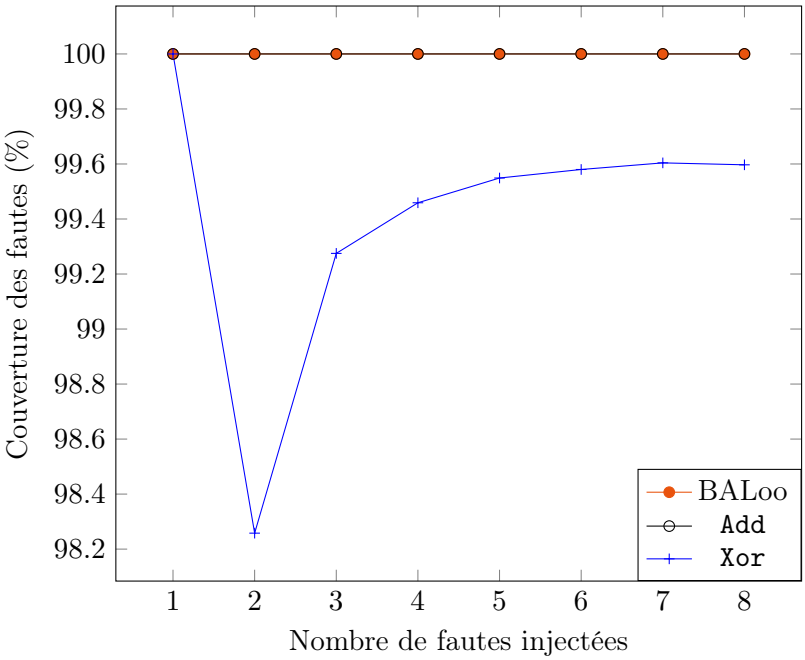

FIGURE 5.6 – Couverture des fautes pour le modèle BITSET/BITRESET. `Add` représente la solution Addition et `Xor` la solution de somme binaire.

Dans ce modèle, les solutions BALoo et d'addition atteignent 100% de couverture de fautes. En effet, afin de contourner la contremesure BALoo, et donc garder le même nombre de cycles de même longueur, il est nécessaire de faire « augmenter » au moins une valeur de la S-box et « diminuer » au moins une autre valeur de la S-box. BALoo ne peut donc pas être contournée dans les modèles d'injection à impact unique (BITSET ou BITRESET).

De même, pour la solution d'addition, injecter des fautes dans ce modèle revient à augmenter (respectivement diminuer) une ou plusieurs valeurs de la S-box et ainsi forcément augmenter (respectivement diminuer) la somme totale des valeurs de la S-box.

La solution de somme binaire peut néanmoins être contournée avec ce modèle d'injection. En effet, afin de contourner cette valeur, il suffit d'injecter plusieurs fautes dont les impacts se compensent. Ainsi, l'attaquant va chercher à passer à 1 (respectivement à 0) un nombre pair de bits positionnés au même indice de plusieurs octets de la S-box. Ainsi, un grand nombre de fautes peuvent se compenser les unes les autres. Cependant, ces fautes nécessitent une grande précision afin de réussir à forcer exactement les bons bits à 1 (respectivement à 0).

5.3.2 Modèle BITFLIP

La Figure 5.7 montre la couverture des fautes pour le second modèle (BITFLIP).

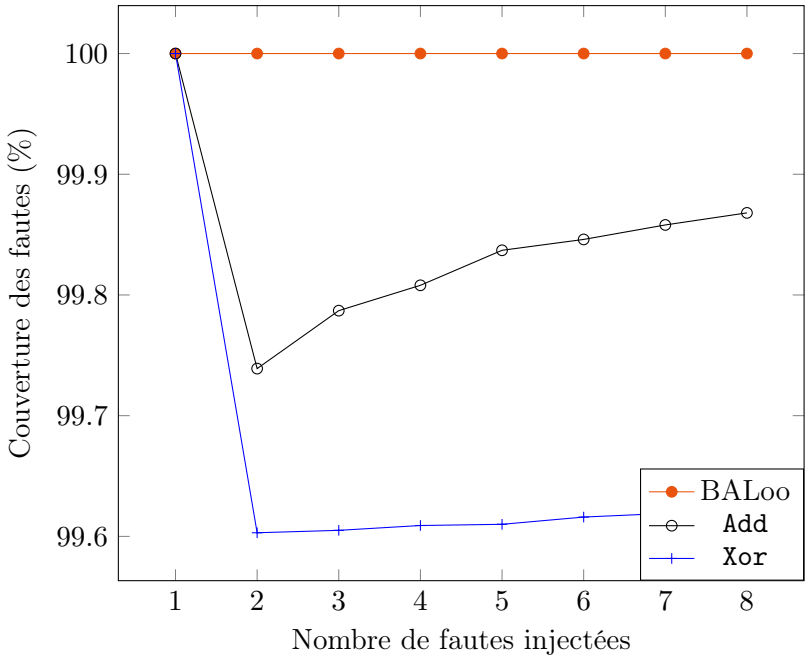

FIGURE 5.7 – Couverture des fautes pour le modèle BITFLIP.

Avec ce modèle d'injection, seule la contremesure BALoo atteint 100% de couverture de faute. En théorie, une partie des fautes injectées pourraient ne pas être détectées par BALoo, cependant, un minimum de trois injections très précises doit être effectué. De plus, même avec une non détection des injections, la S-box fautée ne sera pas exploitable par un attaquant suivant la PFA. C'est pourquoi nous considérons que 100% des fautes sont couvertes par la contremesure BALoo.

Les deux autres solutions ne détectent pas toutes les fautes. En effet, de même que pour le modèle précédent, des injections précises au bit près peuvent contourner la somme et la somme binaire. Pour la somme, il faut qu'un ou plusieurs bits soient passés à 1 et compensés par un ou plusieurs bits passés à 0. Pour la somme binaire, un nombre pair de bits de chaque emplacement peut être basculé sans que la solution permette de le détecter.

Pour la solution de somme binaire, il est même possible de compter précisément le nombre de fautes non détectées, et ce quel que soit le nombre de fautes injectées dans la S-box. Prenons n différentes injections, dont les valeurs des fautes sont dénotées f_1, f_2,..., f_n. Afin de compenser ces fautes, une seule $(n+1)^{\text{ème}}$ valeur d'injection est possible.
Cette valeur est donnée par f_{n+1} :

$$f_{n+1} = f_1 \oplus f_2 \oplus \cdots \oplus f_n$$

Ainsi, quel que soit le nombre d'injections réalisées, afin de contourner la solution de somme binaire, il faut que la dernière injection prenne la valeur de la somme binaire de toutes les autres. Dans le cas de la S-box de l'AES, avec des valeurs sur 8 bits, une seule valeur de faute parmi les 255 valeurs possibles ($f = 0$ n'est pas considérée comme une faute) permet de compenser les autres. La probabilité de couverture des fautes est donc de $\frac{254}{255} \simeq 99,60\%$. Ce calcul simple est réalisable car la détection des fautes n'est basée que sur les valeurs des fautes en elles-même.

Au contraire, dans le modèle BITSET/BITRESET, il est plus compliqué de prédéterminer la couverture de fautes par le calcul. En effet, plusieurs groupes de fautes vont avoir le même impact global sur la somme binaire. Prenons l'exemple des deux premiers octets de la S-box de l'AES : `01100011` et `01111100` dans leur représentation binaire. Toutes les fautes faisant apparaître un 1 à l'indice 7 de S_0 sans modifier les autres bits ont le même impact.

Par exemple avec les trois fautes $f_0^0 =$ `10000000`, $f_0^1 =$ `11000000` et $f_0^2 =$ `11100000` qui ont le même impact sur S_0 :

$$S_0^* = S_0 \vee f_0^0 = S_0 \vee f_0^1 = S_0 \vee f_0^2 = \texttt{1}1100011$$

Afin de compenser une faute ayant un tel impact, l'adversaire peut injecter la faute qu'il souhaite sur S_1 à condition qu'elle mette un 1 à son bit d'indice 7 sans modifier le reste des bits afin de trouver $S_1^* =$ `1`1111100. Ainsi, plusieurs injections peuvent compenser la même faute. L'exemple donné ici n'est que sur deux injections, mais le problème peut aussi se poser sur des injections plus nombreuses. Ainsi, la détection des fautes est liée aux valeurs des fautes injectées ainsi qu'aux octets de la S-box qui sont affectés. Il est donc plus compliqué de déterminer par le calcul la couverture des fautes de la solution de somme binaire dans le modèle de fautes BITSET/BITRESET.

De même pour la solution d'addition dans le modèle BITFLIP. La détection repose sur les valeurs de fautes ainsi que les octets originaux de la S-box. Avec cette solution, plusieurs groupes de fautes peuvent même compenser l'impact des autres, et plusieurs fautes peuvent avoir le même impact sur un même octet. Avec ces deux faits, il est très compliqué de donner mathématiquement une couverture de fautes théorique.

Les injections aléatoires rendent ainsi bien compte de la couverture de fautes des différentes solutions. On retrouve le 99.60% de couverture de la solution de somme binaire dans le modèle de fautes BITFLIP, ainsi que les baisses de couverture dans les autres solutions pour les différents modèles. On voit clairement dans la Figure 5.7 une remontée de la couverture en augmentant le nombre de fautes injectées, qui est due au fait que les compensations de fautes reposent sur des injections très précises. Ainsi, il est plus compliqué aléatoirement d'obtenir un grand nombre de fautes très

précises. Néanmoins, ce qu'il faut retenir est que BALoo permet de couvrir à 100% les fautes utilisables pour une PFA. Les deux autres solutions, ne donnant pas 100% de couverture à partir de deux fautes injectées, un attaquant, dont le modèle d'injection serait plus fort (injection au bit près, choix de ce bit et des octets visés, ...), pourraît, avec ces deux solutions, faire en sorte de toujours tomber dans les quelques injections non détectables. C'est pourquoi nous pensons que BALoo est la seule possibilité de détection, et c'est donc cette solution qui sera implémentée sur un algorithme de chiffrement.

5.4 BALoo appliqué à d'autres chiffrements

Toute fonction de substitution de chiffrement peut être caractérisé par ses cycles, nous allons donc étudier différents chiffrements utilisés dans la littérature, en donnant le nombre de cycles ainsi que la longueur du plus long cycle de la représentation de sa S-box. En effet, afin de détecter l'injection d'une ou plusieurs fautes, tous les cycles doivent être parcourus en entier. Néanmoins, lors d'une implémentation parallélisée de la contremesure, tous les cycles pourraient être parcourus en même temps, le cycle le plus long faisant alors office de chemin critique de la solution. Les chiffrements par blocs étudiés seront les suivant : LOW-MC [Alb+15], PRESENT [Bog+07b], PRINCE[Bor+12], GIFT [Ban+17], SERPENT [ABK98], NOEKEON [DR00], PRIMATE [And+15], ELEPHANT [Bey+21], GIFT-COFB [Ban+20], ASCON [Dob+21], AES [DR02] et ROMULUS [Iwa+20].

Pour les chiffrements SERPENT et GIFT-COFB, plusieurs S-boxes différentes sont utilisées dans la couche de substitution. Un indice permettra ainsi de les différencier. Ces résultats sont donnés dans le Tableau 5.7.

Table 5.7 – Construction sous forme de cycles de plusieurs chiffrements par blocs de la littérature.

Chiffrement par blocs	Taille du bloc (bits)	Nombre de cycles	Longueur du plus long cycle
LOW-MC	3	3	6
PRESENT	4	4	7
PRINCE	4	4	8
GIFT	4	2	9
SERPENT_1	4	4	7
SERPENT_2	4	4	10
SERPENT_3	4	2	13
SERPENT_4	4	5	5
SERPENT_5	4	3	13
SERPENT_6	4	3	14
SERPENT_7	4	4	10
SERPENT_8	4	3	9
NOEKEON	4	10	2
PRIMATE	4	5	11
ELEPHANT	4	2	13
GIFT-COFB_1	5	2	31
GIFT-COFB_2	5	4	10
GIFT-COFB_3	5	2	31
GIFT-COFB_4	5	8	5
ASCON	5	2	26
AES	8	5	87
ROMULUS	8	12	140

Pour le chiffrement qui est plus particulièrement étudié ici, l'AES, les cycles sont au nombre de 5. Ils sont donnés dans le Tableau 5.8. La somme des longueurs est bien

de $87 + 81 + 59 + 27 + 2 = 256$, soit le nombre de valeurs de la S-box. Le cycle le plus long est composé de 27 valeurs.

TABLE 5.8 – Décomposition sous forme de cycles de l'AES.

Premier indice du cycle	**Longueur du cycle**	**Valeurs du cycle**
4	87	4, 242, 137, 167, 92, 74, 214, 246, 66, 44, 113, 163, 10, 103, 133, 151, 136, 196, 28, 156, 222, 29, 164, 73, 59, 226, 152, 70, 90, 190, 174, 228, 105, 249, 153, 238, 40, 52, 24, 173, 149, 42, 229, 217, 53, 150, 144, 96, 208, 112, 81, 209, 62, 178, 55, 154, 184, 108, 80, 83, 237, 85, 252, 176, 231, 148, 34, 147, 220, 134, 68, 27, 175, 121, 182, 78, 47, 21, 89, 203, 31, 192, 186, 244, 191, 8, 48
1	81	1, 124, 16, 202, 116, 146, 79, 132, 95, 207, 138, 126, 243, 13, 215, 14, 171, 98, 170, 172, 145, 129, 12, 254, 187, 234, 135, 23, 240, 140, 100, 67, 26, 162, 58, 128, 205, 189, 122, 218, 87, 91, 57, 18, 201, 221, 193, 120, 188, 101, 77, 227, 17, 130, 19, 125, 255, 22, 71, 160, 224, 225, 248, 65, 131, 236, 206, 139, 61, 39, 204, 75, 179, 109, 60, 235, 233, 30, 114, 64, 9
0	59	0, 99, 251, 15, 118, 56, 7, 197, 166, 36, 54, 5, 107, 127, 210, 181, 213, 3, 123, 33, 253, 84, 32, 183, 169, 211, 102, 51, 195, 46, 49, 199, 198, 180, 141, 93, 76, 41, 165, 6, 111, 168, 194, 37, 63, 117, 157, 94, 88, 106, 2, 119, 245, 230, 142, 25, 212, 72, 82
11	27	11, 43, 241, 161, 50, 35, 38, 247, 104, 69, 110, 159, 219, 185, 86, 177, 200, 232, 155, 20, 250, 45, 216, 97, 239, 223, 158
115	2	115,143

5.5 Coûts d'implémentation

Afin d'évaluer les coûts d'implémentation de la contremesure BALoo, deux types de surcoûts vont être calculés : les coûts d'implémentation logiciel et les coûts d'implémentation matériel. Dans les deux cas, ce surcoût est calculé à partir d'une implémentation de l'AES. Comme c'est l'algorithme de chiffrement par blocs le plus répandu à ce jour, estimer les surcoûts sur cet algorithme est le plus pertinent. Ainsi, nous utiliserons une implémentation logicielle de l'AES-128, issue d'un code Python, ainsi qu'une implémentation matérielle de l'AES-128 réalisée sur deux FPGA différents. Cette dernière sera présentée un peu plus en détail dans la section dédiée.

Afin de garantir la sécurité de ces implémentations, la solution BALoo va être ajoutée au chiffrement de base. Son objectif est de vérifier l'intégrité de la S-box en lisant ses valeurs. Elle ne peut donc pas être appliquée simultanément avec un chiffrement. La question qui en résulte est : au bout de combien de chiffrements est-il nécessaire de vérifier l'intégrité de la S-box ? La réponse est un compromis entre le prix et la sécurité. En effet, plus la S-box est vérifiée souvent, moins l'adversaire pourra récupérer d'information entre deux vérifications, néanmoins, cela va ajouter beaucoup de délai pour effectuer un grand nombre de chiffrements. En effet, comme les deux aspects de cet algorithme protégé (chiffrement et BALoo) ne sont pas compatibles, vérifier trop souvent la S-box empêche le chiffrement des messages.

Pour décider à quelle fréquence appliquer BALoo, il faut déterminer la quantité d'information obtenue par un adversaire en fonction du nombre de chiffrés fautés auxquels il a accès. Dans leur papier, Zhang *et al.* [Zha+18] estiment qu'en moyenne, 2200 chiffrements sont nécessaires pour retrouver la clé, avec une entropie de 1. Cela signifie (d'après la Définition 3) qu'avec environ 2200 messages chiffrés, la clé la plus probable trouvée par l'analyse est la bonne. Avant ces 2200 messages chiffrés, plusieurs valeurs de la S-box sont toujours candidates pour être la valeur qui n'apparaît pas, et ce, pour chacun des octets de la clé. La Figure 5.8 présente ce résultat du nombre d'apparitions de chaque valeur d'un octet selon le nombre de chiffrés obtenus par l'adversaire.

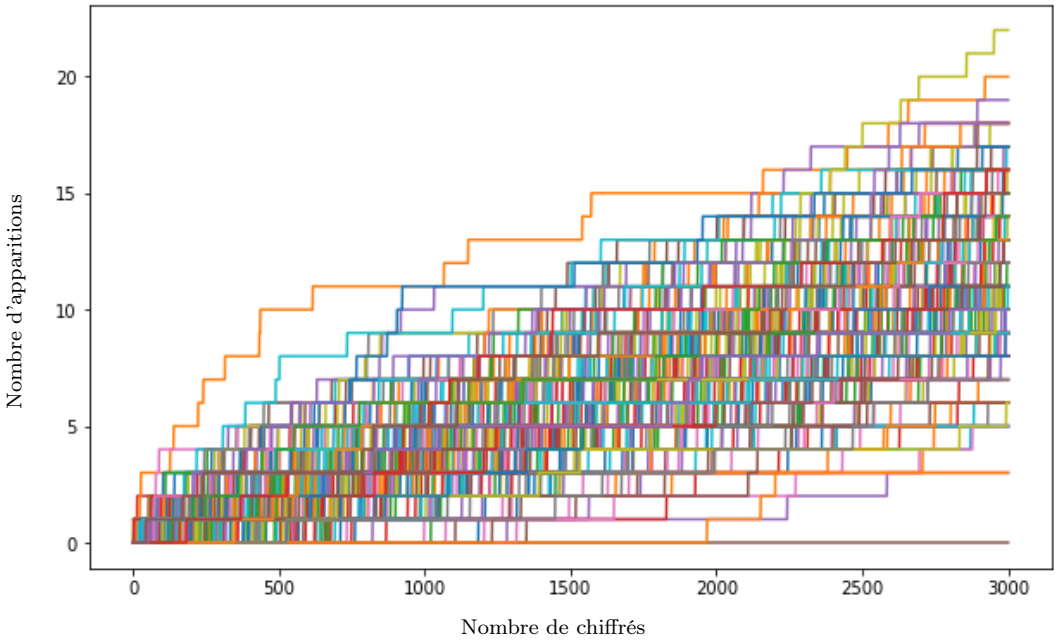

FIGURE 5.8 – Nombre d'apparitions de chaque valeur possible de l'octet 3 des chiffrés.

Sur cette figure, chaque ligne de couleur représente le nombre d'apparitions de chaque valeur possible d'un octet dans les messages chiffrés. On voit bien que toutes les valeurs sauf une apparaissent au bout d'environ 2000 chiffrements. Ainsi, à partir de ces 2000 chiffrements, l'adversaire est en mesure de déterminer, pour cet octet de chiffré, quelle est la valeur non présente et ainsi déterminer cet octet de clé précis. Néanmoins avec seulement 1000 chiffrements, le nombre de candidats qui n'apparaissent pas est porté à 11. Ainsi, même l'adversaire le plus puissant (connaissant déjà l'emplacement et la valeur de la faute injectée) a encore 11 différents candidats pour cet octet de clé. Dans l'état actuel, avec un adversaire plus réaliste ne connaissant ni l'emplacement ni l'impact de l'injection, l'utilisation de la PFA est impossible dans un temps raisonnable.

De même sur un autre octet (Figure 5.9), on remarque bien quelle est la valeur n'apparaissant plus et quelle est la valeur la plus présente. Grâce à ces valeurs, l'adversaire peut déterminer l'impact de la faute injectée sur la S-box. En effet, si on note c_{min} la valeur de l'octet du chiffré qui n'apparaît plus et c_{max} celle qui apparaît deux fois plus souvent, alors la valeur de la faute injectée est $f = c_{min} \oplus c_{max}$. Sur cet octet en

particulier, il peut retrouver cette information ainsi que la valeur de c_{min} plus rapidement que sur l'exemple précédent. Cependant, comme les deux figures proviennent du même ensemble de chiffrements, l'adversaire doit déterminer tous les octets de clé. Il doit donc quand même attendre 2000 chiffrements pour retrouver la clé complète.

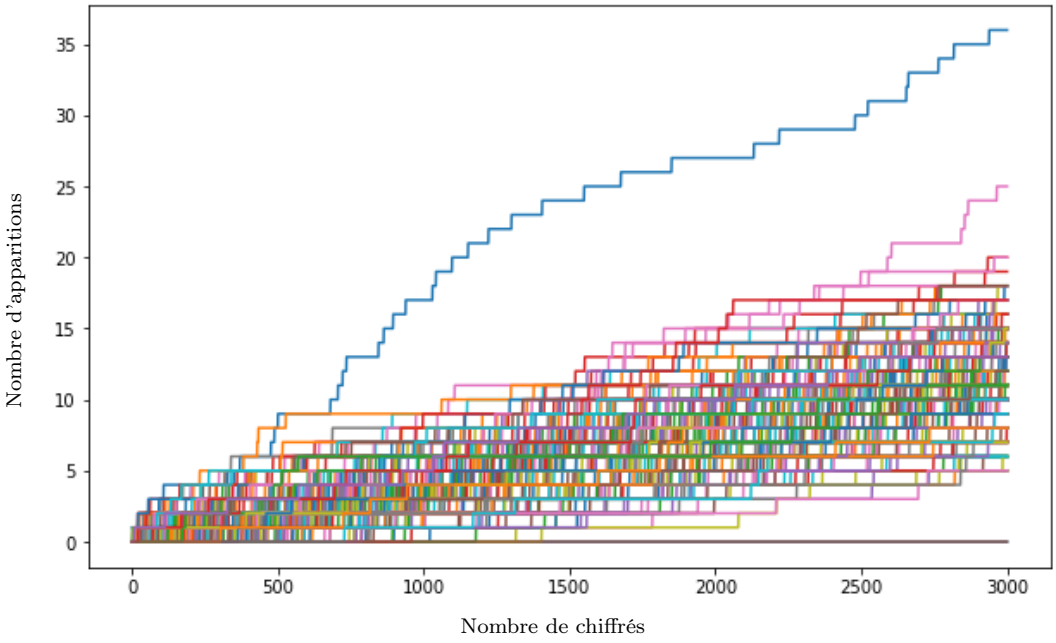

FIGURE 5.9 – Nombre d'apparition de chaque valeurs possible de l'octet 7 des chiffrés.

Les 2200 chiffrements assurent une entropie de la clé de 1, quand seulement 1000 chiffrements font passer cette entropie à 40 bits. Cela signifie qu'avec 1000 chiffrements, l'analyse PFA réduit l'attaque par force brute à réaliser sur 2^{128} clés à une attaque par force brute à réaliser sur 2^{40} clés. On peut alors considérer que l'implémentation reste sécurisée, même avec 1000 chiffrements fautés captés par l'adversaire. Ainsi, réaliser une vérification tous les 1000 chiffrements va permettre à l'adversaire de récupérer de l'information sur au plus 1000 chiffrés fautés (s'il injecte une faute très précisément juste après la dernière vérification), et quand même maintenir un haut degré de sécurité de l'algorithme.

5.5.1 Surcoût logiciel

Comme présenté dans la Chapitre 2, nous allons ici estimer le surcoût logiciel sur un principal aspect : le nombre d'appels mémoires. En effet, le principal surcoût logiciel est le temps d'exécution ajouté à chaque vérification de la S-box. Ce délai est fortement lié au nombre d'appels mémoire nécessaires à cette vérification. Dans notre cas, un appel mémoire est la récupération d'un octet de la S-box. Afin d'évaluer ce surcoût, il faut se placer dans le cas où aucune faute n'est injectée. En effet, dans le cas contraire, la détection va engendrer d'autres décisions et modifications du comportement de l'algorithme. Ces décisions ne rentrent pas dans le cadre de cette étude et ne sont donc pas comptabilisées. Le but de la contremesure est simplement de détecter une injection et non de décider quoi faire par la suite. Le surcoût est donc évaluer dans un fonctionnement « normal » de l'algorithme.

Une vérification complète de la S-box de l'AES nécessite donc 256 appels mémoire, à ajouter aux appels déjà effectués lors des chiffrements. Chaque chiffrement comporte 10 tours, dans lesquels est appelée une fois la fonction `SubBytes` faisant intervenir la S-box. Cette fonction de substitution est appliquée à l'ensemble des octets de l'état du chiffrement. Ainsi, chaque appel à la fonction `SubBytes` engendre 16 appels mémoire vers la S-box. On peut donc comptabiliser 160 appels mémoire vers la S-box à chaque chiffrement.

Comme vu dans la section précédente de ce chapitre, réaliser une vérification tous les 1000 chiffrements garantie la sécurité de l'algorithme. Ainsi, un ajout de 256 appels mémoire pour BALoo est appliqué tous les 160 000 appels mémoire pour les chiffrements. Cela équivaut à un surcoût logiciel en termes d'appels mémoire vers la S-box d'environ $0,16\%$, pour une sécurité de plus de 40 bits.

Si le designer souhaite augmenter la sécurité en passant d'une vérification tous les 1000 chiffrements à une vérification tous les 500 chiffrements, le surcoût logiciel en termes d'appels mémoire vers la S-box passe à environ 0.32% pour une sécurité de plus de 80 bits. Ces résultats sont donnés dans le Tableau 5.9.

TABLE 5.9 – Surcoût logiciel de BAloo en termes d'appels mémoire vers la S-box.

Intervalle de vérification	**Surcoût calculé**	**Entropie de clé**
1000 chiffrements	0.16%	$> 2^{40}$
500 chiffrements	0.32%	$> 2^{80}$

Le surcoût logiciel en termes d'appels mémoire vers la S-box est donc négligeable, ce qui fait de BALoo une contremesure très efficace quand elle est appliquée sur une implémentation logicielle des chiffrements. Cette efficacité est liée au nombre de chiffrements nécessaires à l'attaquant avant de retrouver la clé.

5.6 Surcoût matériel

En premier lieu, l'implémentation matérielle et ses spécifications vont être présentées, puis les solutions apportées pour appliquer BALoo à cette implémentation et enfin le surcoût engendré par ces solutions.

5.6.1 Implémentation matérielle à 16 S-boxes

L'implémentation matérielle de l'AES qui est utilisée intègre 16 S-boxes pour le chiffrement. L'avantage principal de cette implémentation est son optimisation en termes de performances. En effet, un message en clair est chiffré en seulement 11 cycles d'horloge. L'inconvénient, pour nous, est cependant que toutes les 16 S-boxes doivent être protégées face aux injections de fautes.

Dans cette implémentation précise, chaque octet de l'état passe à travers une et une seule S-box durant tout le chiffrement. Ainsi, fauter une S-box permet de biaiser la distribution de probabilités d'apparitions d'un seul octet. Cela permet donc de récupérer un octet de clé. L'attaquant doit donc chercher à injecter une faute dans une valeur de chaque S-box. Il faut donc vérifier l'intégrité de chacune des S-boxes. Un appel de `SubBytes` dans cette configuration est illustré dans la Figure 5.10.

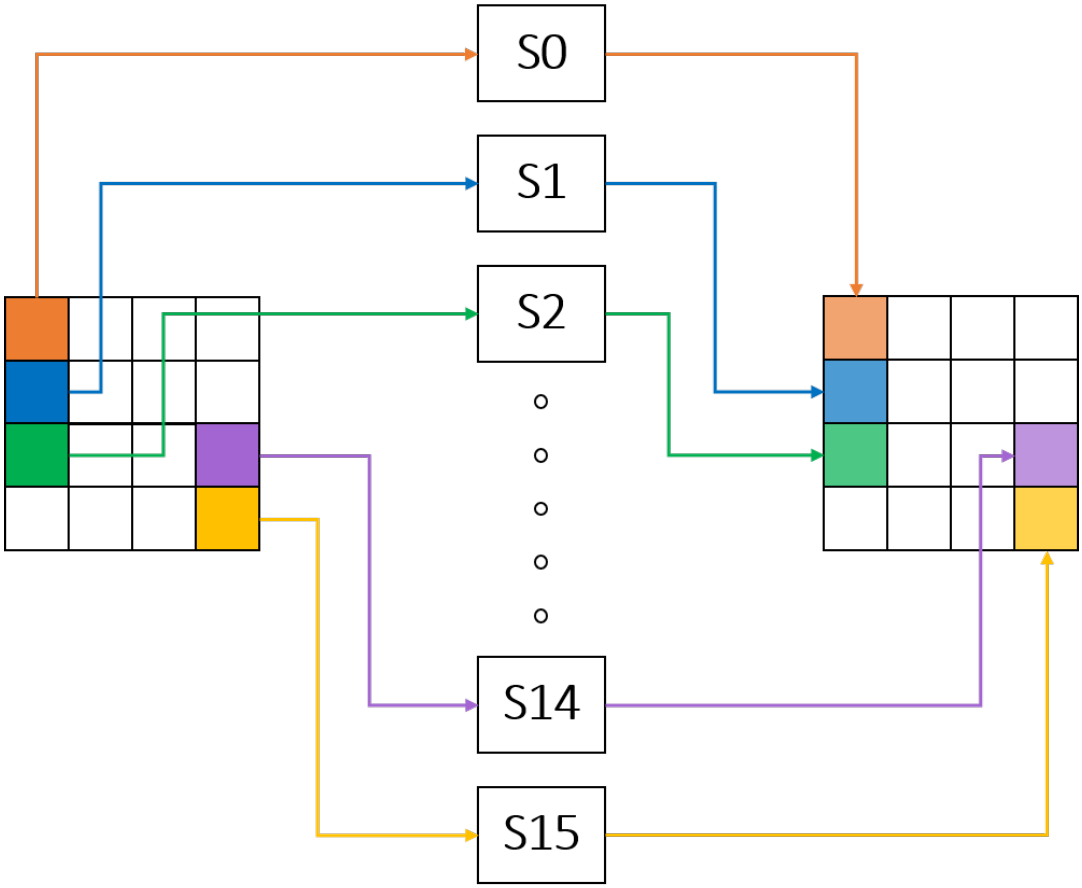

FIGURE 5.10 – Appel de `SubBytes` avec 16 S-boxes.

De plus, comme cela sera présenté par la suite, il est possible que les octets de l'état passent à travers plusieurs S-boxes différentes. Par simplicité, on va faire l'hypothèse que chaque octet peut et va passer à travers toutes les S-boxes au cours des nombreux chiffrements réalisés. Ainsi, même en injectant une faute sur une seule S-box, les distributions de probabilités d'apparition de chacun des octets seront biaisées. En effet, même si toutes les valeurs apparaîtront au bout d'un grand nombre de chiffrements, nous aurons quand même une valeur avec une plus faible probabilité d'apparition (lorsque l'état passe à travers la S-box fautée dans le dernier round) et une valeur avec une plus forte probabilité d'apparition. Cela est illustré dans la Figure 5.11, où l'on voit en bleu une valeur apparaissant plus souvent que les autres et en rouges une valeur apparaissant moins souvent. Ces valeurs sont liées à la valeur n'apparaissant plus et la valeur apparaissant deux fois dans la S-box fautée.

On remarque cependant que pour distinguer la valeur maximale et la valeur minimale, un bien plus grand nombre de chiffrements sont nécessaires. Ainsi, en fonction des capacités de l'adversaire, ce dernier devra soit fauter un seul octet de chacune des S-boxes (ce qui est déjà une difficulté) et n'aura besoin que d'environ 2000 chiffrements pour retrouver la clé, soit fauter une seule S-box, mais son analyse nécessitera un bien plus grand nombre de chiffrements.

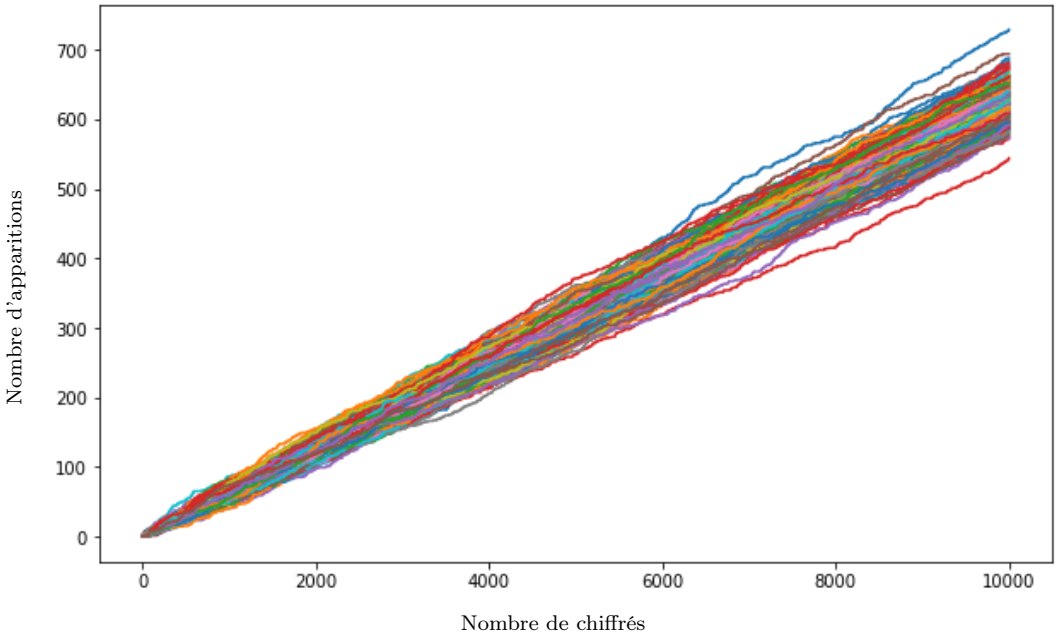

FIGURE 5.11 – Nombre d'apparition de chaque valeur possible de l'octet 2 des chiffrés, au travers de 16 S-boxes dont une est fautée.

5.6.2 Application de BALoo

La première approche afin de protéger toutes les S-boxes est de tenter de vérifier l'ensemble des S-boxes en même temps. Cependant, cette solution a été testée et est revenue très coûteuse (plus de 350% de surcoût en termes de ressources logiques). Ainsi, il a fallu imaginer une autre façon de vérifier l'intégrité des S-boxes à moindre coût tout en garantissant une haute sécurité du chiffrement.

La solution retenue est l'ajout d'une 17^e S-box dans l'implémentation. Cette nouvelle S-box va permettre d'utiliser 16 S-boxes pour le chiffrement alors que une S-box sera vérifiée par BALoo. Le but est donc de vérifier l'intégrité d'une S-box pendant que les 16 autres sont utilisées pour le chiffrement, puis de vérifier l'intégrité de la S-box suivante en réutilisant la première pour le chiffrement. Ainsi, la contremesure BALoo est pleinement appliquée quand les 17 S-boxes sont vérifiées. Cette solution est illustrée dans la Figure 5.12.

Une vérification d'une S-box nécessite 256 cycles d'horloge pour parcourir les 256 valeurs de la S-box. Ainsi, la vérification complète des 17 S-boxes est effectuée en $17 \times 256 = 4352$ cycles d'horloge. En se rappelant qu'un chiffrement dure 11 cycles d'horloge, on peut affirmer que la vérification complète est faite en 396 chiffrements. Dans le meilleur des cas pour l'adversaire, il injecte une faute sur une S-box juste après sa dernière vérification, et obtient des informations sur moins de 400 chiffrés avant que l'injection soit détectée. On a vu dans la Figure 5.11 qu'avec un si petit nombre de chiffrés, l'attaquant ne peut pas distinguer la probabilité minimale et la probabilité maximale. La sécurité est donc assurée dans ce cas.

Cependant, en cas de volonté de plus de sécurité, il est également possible d'ajouter deux nouvelles S-boxes plutôt qu'une seule. Ainsi, avec 18 S-boxes, deux sont vérifiées par BALoo quand les 16 autres sont utilisées pour le chiffrement. BALoo appliquée sur

18 S-boxes est illustré dans la Figure 5.13. Dans ce cas seules 9 étapes sont nécessaires pour vérifier l'intégrité de l'ensemble des 18 S-boxes, ce qui équivaut à 2304 cycles d'horloge, ou 210 chiffrements. Comme dans le paragraphe précédent, cela signifie que dans le meilleur des cas, l'adversaire obtient des informations sur seulement 210 chiffrés. L'analyse résultat de l'exploitation de ces 210 chiffrés est donc encore moins efficace que précédemment. La sécurité en est grandement renforcée.

Le Tableau 5.10 présente les différents niveaux de sécurité apportés par les implémentations avec 17 ou 18 S-boxes.

TABLE 5.10 – Sécurité des différentes implémentations de l'AES sécurisé avec BALoo.

	# S-boxes	# Verifiées	# Cycles d'horloge	# Chiffrements	Entropie de clé
AES avec 17 S-boxes	17	1	4352	396	2^{60}
AES avec 18 S-boxes	18	2	2304	210	2^{120}

5.6.3 Surcoût de ces solutions

Les cibles matérielles choisies sont les FPGAs Cyclone V 5CGXFC9EF35C8 et Cyclone 10 LP 10CK120ZF780I8G, avec le logiciel Quartus II, dont les options choisies sont « *best effort for performance and area synthesis* ». Ainsi, le logiciel va tenter d'optimiser la taille nécessaire pour l'implémentation de l'AES protégé, ainsi que ses performances.

Les résultats de coûts donnés par le logiciel sont présentés dans le Tableau 5.11. Même si la contremesure ajoute une S-box (respectivement deux S-boxes), le surcoût est principalement engendré par la logique autour de ces 17 (respectivement 18) S-boxes permettant de sélectionner quelles sont la ou les S-boxes utilisées pour le chiffrement et sur lesquelles sont appliquée BALoo. Les valeurs de consommation de puissance et de fréquence maximales ne sont que des estimations réalisées par le logiciel.

Le surcoût engendré peut paraître élevé, mais il est à mettre en perspective avec l'implémentation à sécuriser. En effet, cette implémentation de l'AES est très optimisée en termes de performances (chiffrement réalisé en seulement 11 coups d'horloges) et nécessite donc la protection de 16 S-boxes utilisées en parallèle. Le résultat intéressant ici est le fait que le passage de 17 à 18 S-boxes est bien moins coûteux par rapport au passage de 16 à 17 S-boxes en comparaison de la différence de sécurité apportée par les 18 S-boxes contre celle amenée par les 17.

5.7 Conclusion du chapitre

Ce chapitre introduit donc la première contremesure spécialement dédiée à l'analyse des fautes persistantes PFA, nommée BALoo, qui détecte 100% des injections exploitables par un adversaire appliquant une PFA.

Les affirmations sur la sécurité donnée par l'application de BALoo tous les n chiffrements sont basées sur les résultats de Zhang *et al.* [Zha+18] qui partent du principe que l'adversaire n'a pas de contrôle sur les entrées du chiffrement, et on peut donc associer ces messages en clair à des entrées aléatoires. Cependant, il est possible qu'un modèle d'adversaire plus puissant, ayant le contrôle sur les messages d'entrées, puisse

TABLE 5.11 – Coûts d'implémentation des AES sécurisés.

Cyclone V 5CGXFC9E7F35C8

	Ressources FPGA				
	Logique (ALM)	Registres	Mémoire (bits)	Puissance (mW)	F_{max} (MHz)
AES	339	13	32 768	566,32	94
AES avec 17 S-boxes	608	34	34 816	568,78	58
AES avec 18 S-boxes	763	43	36 864	568,48	51

	Surcoût (%)				
	Logique	Registres	Mémoire	Puissance	F_{max}
AES avec 17 S-boxes	79	162	6	0,43	-38
AES avec 18 S-boxes	125	231	13	0,38	-46

Cyclone 10 LP 10CL120ZF780I8G

	FPGA resources				
	Logique (LE)	Registres	Mémoire (bits)	Puissance (mW)	F_{max} (MHz)
AES	649	13	32 768	233,23	119
AES avec 17 S-boxes	1263	34	34 816	238,55	78
AES avec 18 S-boxes	1648	43	36 864	242,79	93

	Surcoût (%)				
	Logique	Registres	Mémoire	Puissance	F_{max}
AES avec 17 S-boxes	95	162	6	2,28	-34
AES avec 18 S-boxes	154	231	13	4,10	-22

choisir avec soin ces derniers afin de réduire le nombre de chiffrements nécessaires à l'analyse. Cette analyse avec le choix des textes à chiffrer a été réalisée par Zhang *et al* [Zha+23] et donne une grande amélioration de l'analyse. En effet, l'analyse est ici réalisée en seulement 256 chiffrements bien précis (un octet de l'entrée varie de 0 à 255 afin de parcourir toutes les valeurs de la S-box). Cette nouvelle analyse perd totalement l'avantage qu'avait l'ancienne : l'attaquant doit avoir accès aux entrées et doit même être capable de choisir précisément ces entrées. Dans le cas d'une implémentation logicielle, on peut réduire l'intervalle entre les vérifications sans pour autant rendre la contremesure trop lourde. Par exemple, on retrouve 3% de surcoût pour une vérification tous les 50 chiffrements. Comme le premier octet du message à chiffrer varie de 0 à 255, le premier octet de la sortie de ce message aura parcouru au plus 50 valeurs différentes. Ainsi, il restera encore au moins 206 candidats pour la valeur de l'octet qui n'apparaît pas. Cette affirmation reste vraie pour tous les octets des chiffrés. Ainsi, la sécurité est assurée avec un surcoût très faible. En revanche, pour l'implémentation matérielle, on voit que rajouter seulement une S-box ne suffit plus à assurer la sécurité comme 396 chiffrements sont nécessaires à la vérification complète des S-boxes. Dans ce cas, l'attaque pourra avoir assez de chiffrés pour retrouver la clé. Ajouter une deuxième S-box permet de donner au plus 210 valeurs différentes du premier octet de la sortie, donc au moins 46 valeurs qui n'apparaissent pas. Ainsi, la sécurité reste assez élevée. Il peut être judicieux (mais coûteux) de rajouter une troisième S-box afin de vérifier la totalité en 163 chiffrements, réduisant encore plus les informations perçues.

Dans tous les cas, BALoo a prouvé son efficacité pour la détection de fautes dans la ou les S-boxes. De plus, nous avons proposé des solutions afin de l'appliquer à des implémentations matérielles et logicielles en essayant de réduire au mieux le surcoût engendré par la contremesure.

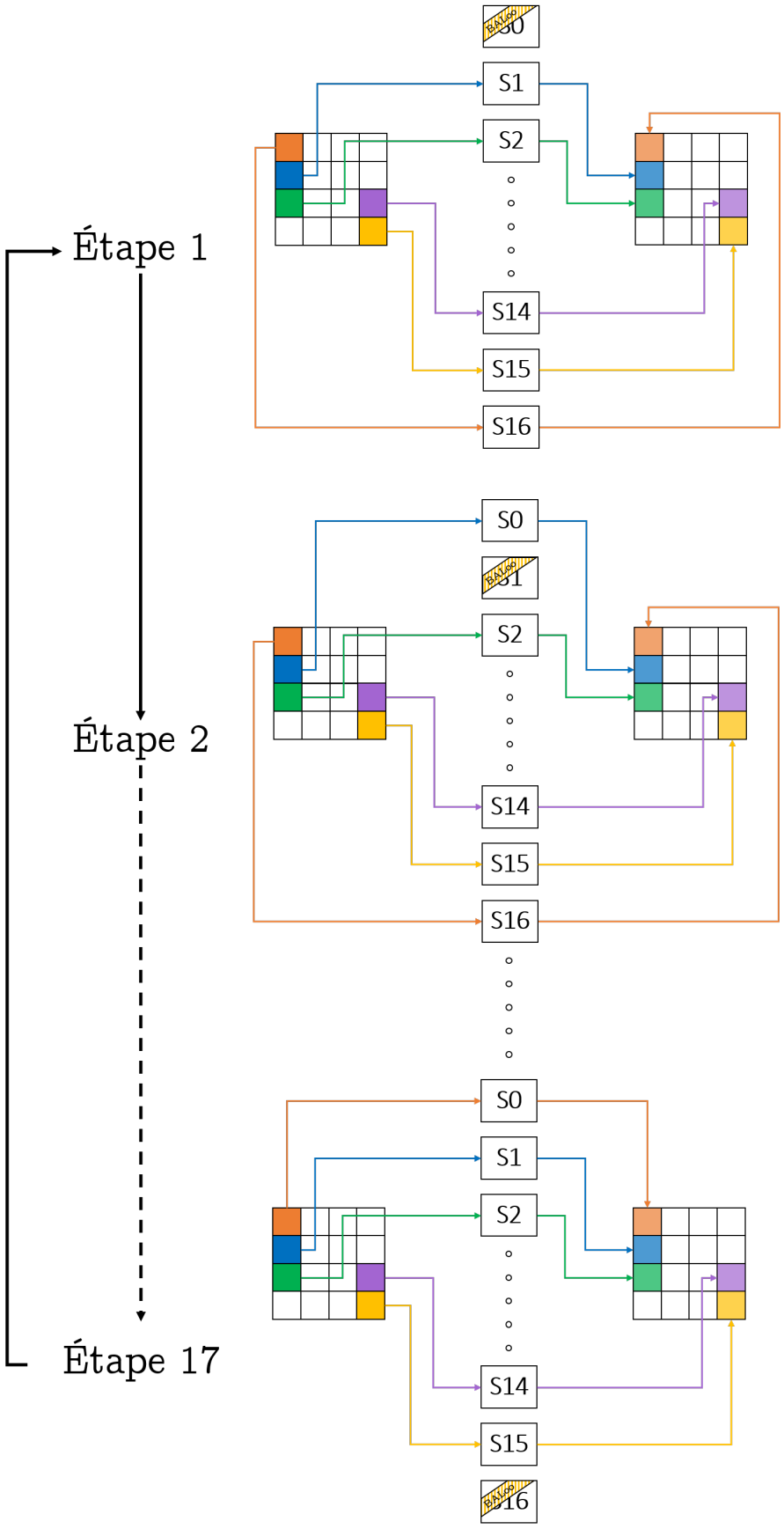

FIGURE 5.12 – Appel de `SubBytes` avec 17 S-boxes.

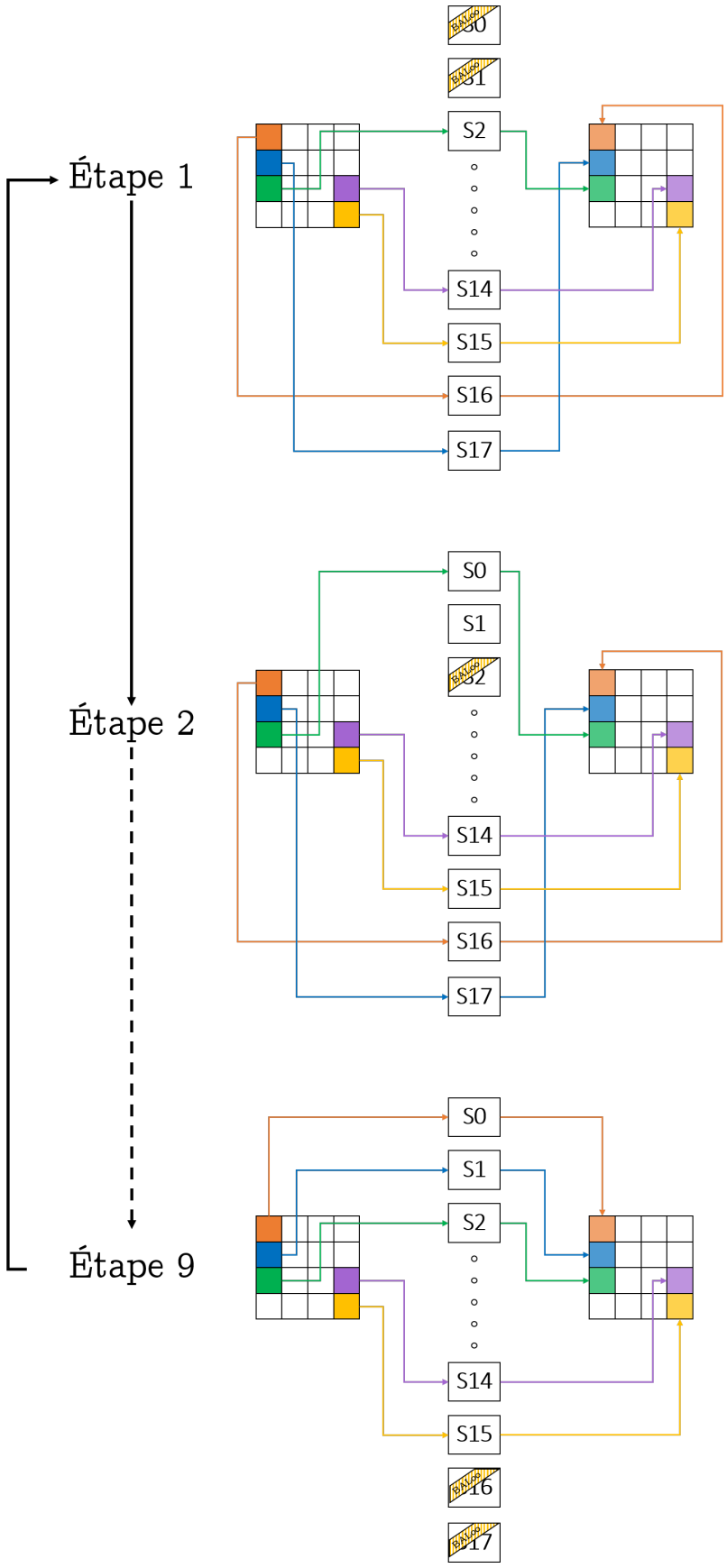

FIGURE 5.13 – Appel de `SubBytes` avec 18 S-boxes.

Conclusion

Ce manuscrit a présenté deux contremesures faces à des injections multiples de fautes sur une implémentation matérielle d'un algorithme de chiffrement. Ce type d'attaque, qui combine une injection physique grâce à des moyens matériels d'injection (bancs laser, glitchs d'horloge, rayons X, ...) et une analyse mathématique de l'impact de cette injection (étude différentielle, recherche d'une erreur dans la distribution des valeurs des messages chiffrés, ...) est largement appliquée et améliorée sur tous ses aspects. En effet, de nouveaux bancs laser multispots permettent d'injecter des fautes de plus en plus précises et de plus en plus nombreuses en même temps, quand en parallèle les améliorations des analyses peuvent récupérer une clé de chiffrement en utilisant de moins en moins de chiffrements. Au milieu de ces améliorations, les contremesures sont également de plus en plus poussées pour faire face au développement des attaques et à leur diversité. Néanmoins, les contraintes de conception de primitives cryptographiques embarquées, liées à la consommation d'énergie ou à la taille qui se veulent de plus en plus faibles, ne sont plus en phase avec le développement de contremesures de plus en plus lourdes. Ainsi, ce travail vise à améliorer des contremesures existantes ou à créer de nouvelles défenses pour les algorithmes, en tentant de limiter au mieux le surcoût engendré par la sécurité apportée.

Contributions

La première contribution proposée au Chapitre 3 est concentrée sur l'amélioration de la solution amenée dans la primitive de chiffrement FRIET, publiée par Simon *et al.* [Sim+20]. Cette solution est basée sur la notion de « respect du code » qui intègre un code détecteur d'erreur directement à la création de la primitive. Ce code sépare les données manipulées entre les données correctes (dans le code) et les données fautées (hors du code). Les fonctions utilisées dans le chiffrement respectent ce code, donc une données correcte en entrée donne une donnée correcte en sortie et une donnée fautée en entrée est propagée à travers la fonction. Cela permet de propager l'erreur jusqu'à sa détection. Néanmoins, l'étude de cette solution a mis en lumière des cas d'injection qui permettaient de compenser les impacts des fautes, et ainsi empêcher leur détection. De plus, cette application du code est donnée au niveau du bit, quand les algorithmes traditionnels fonctionnent sur des chiffrements par blocs, la solution est donc difficilement exportable sur des chiffrements existants. Ainsi, notre rôle a été de généraliser cette notion de respect du code à tout algorithme déjà existant en utilisant une conversion bitslice du chiffrement. Combinée à un code de parité, cette transformation permet de détecter tout injection réalisée sur un registre entier d'une cible embarquée. De plus, une solution de copies des données utilisées plusieurs fois permet d'empêcher la compensation des fautes entre elles. Suite à une estimation du surcoût de notre contremesure appliquée au chiffrement léger LED, nous avons conclu que la fonction qui engendrait le plus gros coût d'implémentation était la fonction de substitution. C'est à cette problématique de coût de la S-box que va s'intéresser la deuxième contribution du Chapitre 4. Ces travaux ont été publiés dans la revue MDPI, lors de l'édition spéciale Electronics [TBG23b], et présentés sous forme de

poster lors de l'école d'automne NESSY (Nano-Electronics for Secure Systems), école qui précédait la conférence CARDIS 2021.

La seconde contribution est la suite logique de la première. En effet, le Chapitre 4 va introduire une relation d'équivalence permettant de classifier les permutations selon leur coût d'implémentation. Cette problématique est liée au choix de la représentation sur 5 bits de la S-box (sur 4 bits) du chiffrement LED, sur lequel a été appliquée la solution du Chapitre 3. En effet, comme un des principaux coûts d'implémentation matérielle d'un chiffrement est lié à sa partie non linéaire, et donc sa S-box, il est primordial d'avoir un moyen de réduire ce coût sans interférer sur les preuves de sécurités de l'algorithme. Cependant, sur un chiffrement utilisant des mots de 4 bits, nous avons déjà 16 ! candidats possibles pour la S-box. Ainsi, il est impossible pour un concepteur de tester les propriétés cryptographiques ainsi que les coûts d'implémentation de chacun d'entre elles. C'est là qu'interviennent les relations d'équivalence, permettant de classifier les S-boxes selon des critères bien précis. Le premier critère, et certainement le plus important, est la sécurité apportée par la S-box. Ainsi, une relation d'équivalence de la littérature, la relation d'équivalence Affine, permet de classifier les S-boxes sur 4 bits en 302 classes tout en gardant leurs propriétés cryptographiques. Cependant, plusieurs S-boxes de la même classe d'équivalence Affine, donc ayant les même propriétés cryptographiques, n'ont pas forcément les même coûts d'implémentation. Ainsi, notre rôle a été d'introduire une nouvelle relation d'équivalence, appelée relation WIRE, qui va permettre de classifier les fonctions selon leur nombre de portes logiques nécessaires à leur implémentation. Couplée à la relation Affine, la relation WIRE va permettre de classifier les S-boxes en fonction de leurs propriétés cryptographiques ainsi que leur coût d'implémentation représenté par le nombre de portes logiques de leur implémentation matérielle. Par une méthode dite « diviser pour mieux régner », un designer qui souhaite choisir une S-box va pouvoir sélectionner ses propriétés cryptographiques voulues grâce à la relation d'équivalence Affine, puis sélectionner la classe d'équivalence WIRE qui offre le plus faible coût d'implémentation. Ces travaux sont présentés dans un article qui est soumis à la conférence ICISC 2023 (International Conference on Information Security and Cryptology).

La dernière contribution s'intéresse à une analyse basée sur des injections de fautes persistantes, la PFA. Cette analyse étudie la distribution de probabilité d'apparition de chaque valeur dans les octets des chiffrés issus d'un algorithme de chiffrement. Suite à une injection de faute persistante dans une S-box stockée dans une mémoire non volatile (donc présente jusqu'à la réécriture de la mémoire), une valeur de cette S-box est remplacée par une autre valeur. Ainsi, en sortie de la S-box, une valeur n'apparaît plus, et une valeur apparaît deux fois. En ayant connaissance de la valeur de S-box fautée et de la valeur des octets des chiffrés qui n'apparaissent plus, un adversaire peut alors simplement retrouver la clé en effectuant une somme binaire entre la valeur fautée de la S-box et chacune des valeurs qui manque à chacun des octets. Explicitée récemment, cette analyse n'avait (au moment de l'étude de la contremesure), aucune solution dédiée afin de la contrer. Le Chapitre 5 introduit BALoo, la première contremesure dédiée à la PFA. BALoo exploite la représentation sous forme de cycles d'une permutation. Cette représentation est unique et exclusive à chaque permutation. Ainsi, une faute injectée dans la S-box amène une modification dans la construction de BALoo, et donc une détection de cette faute. Afin de contourner la contremesure, des injections multiples et très précises peuvent être effectuées en théorie. Cependant, lorsque BALoo ne détecte pas la faute, la S-box fautée est toujours une permutation, et l'adversaire ne peut donc pas exploiter de biais dans la distribution des probabilités

d'apparition et effectuer une PFA. BALoo est donc 100% efficace contre les injections avec lesquelles sont couplées une PFA. Dans la suite du chapitre, une évaluation du coût d'implémentation logiciel et matériel de la contremesure est proposée. Cette évaluation résulte en un coût négligeable sur un implémentation logicielle, et une proposition de solution très efficace et avec un surcoût le plus réduit possible lors d'une implémentation matérielle très optimisée. Ces travaux ont été publiés et présentés à la conférence IOLTS 2023 [TBG23a], et ils ont été présentés lors de la journée JAIF 2023 sous forme de poster.

Ouvertures et perspectives

Bien que largement étudiées, les attaques par injection de fautes restent un domaine dans lequel des découvertes sont encore largement envisageables. Les perspectives proposées ici se concentreront sur la partie analyse des injections.

La première piste, liée à la notion de respect du code, serait de changer le code détecteur d'erreur du bit de parité, qui est limité en nombre de fautes détectées, pour le transformer en code plus conséquent, pour détecter plus de fautes ou même corriger certaines injections. Ainsi, la même notion de respect du code pourrait être appliquée à un nouvel algorithme de chiffrement afin d'avoir une primitive qui soit résistante dès sa conception aux attaques par injections multiples de fautes. De plus, le concepteur de la couche non linéaire de ce nouvel algorithme pourrait utiliser la relation d'équivalence WIRE afin d'optimiser ses coûts d'implémentation. Une deuxième idée liée à cette notion de respect du code serait d'ajouter du masquage à une implémentation protégée, afin d'étudier le comportement d'un algorithme de chiffrement protégé à la fois contre les attaques par canaux auxiliaires (avec le masquage) et les attaques par injection de fautes (avec le respect d'un code détecteur ou correcteur d'erreurs). L'étude des deux contremesures combinées peut être intéressante pour vérifier qu'une contremesure face à un type d'attaque ne rend pas la cible trop vulnérable face au second type d'attaque, et que le surcoût combiné des deux contremesure ne soit pas trop conséquent en comparaison des deux surcoûts séparés.

Dans le Chapitre 5, nous avons affirmé la couverture, par la contremesure BALoo, de 100% des fautes exploitables dans le cadre d'un attaquant souhaitant appliquer la PFA. Nous avons néanmoins imaginé un scénario dans lequel une faute non détectée pourrait quand même amener à des fuites d'information sur la clé de chiffrement. En effet, le Chapitre 4 présente plusieurs caractéristiques d'une S-box qui sont indispensables à la sécurité du chiffrement face aux analyses différentielles et linéaires. Ce sont ces caractéristiques qui sont le critère principal du choix d'une S-box parmi l'ensemble des candidats. Ces analyses sont efficaces dans le cas d'une S-box possédant de faibles propriétés cryptographiques. Ainsi, nous pouvons nous demander si les injections de plusieurs fautes très précises qui mèneraient à une modification de la S-box (originellement avec de bonnes caractéristiques), tout en contournant BALoo, ne pourraient pas atteindre une S-box (fautée mais toujours validant les propriétés de permutation vérifiées) qui posséderait de faibles propriétés cryptographiques sur laquelle des analyses différentielles et linéaires pourraient être appliquées. Injecter des fautes qui ne sont pas détectées, mais qui réduisent le degré algébrique ou qui augmentent l'uniformité différentielle et la linéarité de la S-box, pourrait mener à une faiblesse exploitable de la S-box.

Enfin, une dernière perspective, cette fois-ci plutôt dans le point de vue d'un attaquant, serait de trouver une injection différente qui cible une autre partie du chiffrement et qui ne serait pas exploitée dans la littérature. C'est par exemple le cas d'une injection sur la constante `Rcon`, utilisée lors de l'expansion de clé de l'algorithme AES. L'idée serait d'injecter une faute sur un octet précis de `Rcon`, qui pourrait amener à une fuite d'information sur la clé. Une étude de faisabilité de cette faute physiquement (précision de l'injection, valeur de l'erreur à injecter) ainsi que le développement d'une analyse basée sur cette injection pourrait mettre en valeur une nouvelle faiblesse de l'AES face aux attaques par injections de fautes et également ouvrir la voie à de nouvelles contremesures pour parer ce type d'analyse.

Communications et publications

Publication dans un journal international

Pierre-Antoine Tissot, Lilian Bossuet, Vincent Grosso. **"Generalized Code-Abiding Countermeasure. Electronics"**, MDPI, *Electronics*, February 2023 [TBG23b].

Conférence internationale avec comité de lecture

Pierre-Antoine Tissot, Lilian Bossuet, Vincent Grosso. **"BALoo : First and Efficient Countermeasure dedicated to Persistent Fault Attacks"**. *The 29th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS 2023)*, July 2023 [TBG23a].

Présentation à un congrès international sans acte

Pierre-Antoine Tissot. **"Implementation challenges of Photon-Beetle, a NIST Standardization Finalist"**. *Cryptographic Architectures embedded in logic devices (CryptArchi 2022)*, Porquerolles, France, Mai 2022.

Posters

Pierre-Antoine Tissot. **"Fault Resistant Symmetric Encryption Implementation"**. *20th Smart Card Research and Advanced Application Conference (CARDIS 2021)*, Lübeck, Allemagne, Novembre 2021.

Pierre-Antoine Tissot. **"Optimisation d'Implémentation de Contremesures face aux Injections de Fautes"**. *Journée de la recherche de l'école doctorale EDSIS*, Saint-Étienne, France, Juin 2022.

Pierre-Antoine Tissot. **"BALoo : First and Efficient Countermeasure dedicated to Persistent Fault Attacks"**. *Journée thématique sur les attaques par injection de fautes (JAIF 2023)*, Gardanne, France, Septembre 2023.

Bibliographie

[ABK98] Ross J. Anderson, Eli Biham et Lars R. Knudsen. « Serpent and Smartcards ». In : *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*. Sous la dir. de Jean-Jacques Quisquater et Bruce Schneier. T. 1820. Lecture Notes in Computer Science. Springer, 1998, p. 246-253. isbn : 3-540-67923-5. doi : 10.1007/10721064_23. url : https://doi.org/10.1007/10721064_23.

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen et Michael Zohner. « Ciphers for MPC and FHE ». In : *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*. Sous la dir. d'Elisabeth Oswald et Marc Fischlin. T. 9056. Lecture Notes in Computer Science. Springer, 2015, p. 430-454. isbn : 978-3-662-46799-2. doi : 10.1007/978-3-662-46800-5_17. url : https://doi.org/10.1007/978-3-662-46800-5_17.

[Anc+17] Stéphanie Anceau, Pierre Bleuet, Jessy Clédière, Laurent Maingault, Jean-luc Rainard et Rémi Tucoulou. « Nanofocused X-Ray Beam to Reprogram Secure Circuits ». en. In : *Cryptographic Hardware and Embedded Systems – CHES 2017*. Sous la dir. de Wieland Fischer et Naofumi Homma. T. 10529. Series Title : Lecture Notes in Computer Science. Cham : Springer International Publishing, 2017, p. 175-188. isbn : 978-3-319-66786-7 978-3-319-66787-4. doi : 10.1007/978-3-319-66787-4_9. url : http://link.springer.com/10.1007/978-3-319-66787-4_9 (visité le 18/11/2021).

[And+15] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Bart Mennink, Nicky Mouha et Kan Yasuda. « APE : Authenticated Permutation-Based Encryption for Lightweight Cryptography ». In : *Fast Software Encryption*. Sous la dir. de Carlos Cid et Christian Rechberger. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015, p. 168-186. isbn : 978-3-662-46706-0.

[Ban+17] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim et Yosuke Todo. « GIFT : A Small Present - Towards Reaching the Limit of Lightweight Encryption ». In : *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Sous la dir. de Wieland Fischer et Naofumi Homma. T. 10529. Lecture Notes in Computer Science. Springer, 2017, p. 321-345. isbn : 978-3-319-66786-7. doi : 10.1007/978-3-319-66787-4_16. url : https://doi.org/10.1007/978-3-319-66787-4_16.

[Ban+20] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim et Yosuke Todo. « GIFT-COFB ». In : *IACR Cryptol. ePrint Arch.* (2020), p. 738. URL : https://eprint.iacr.org/2020/738.

[Bar09] Gregory Bard. *Algebraic cryptanalysis*. Springer Science & Business Media, 2009.

[Bar+10] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi et Francesco Regazzoni. « Countermeasures against Fault Attacks on Software Implemented AES : Effectiveness and Cost ». In : *Proceedings of the 5th Workshop on Embedded Systems Security*. WESS '10. New York, NY, USA : Association for Computing Machinery, 2010. ISBN : 9781450300780. DOI : 10.1145/1873548.1873555. URL : https://doi.org/10.1145/1873548.1873555.

[Ber+02] Guido Bertoni, Luca Breveglieri, Israel Koren, Paolo Maistri et Vincenzo Piuri. « A Parity Code Based Fault Detection for an Implementation of the Advanced Encryption Standard ». In : *17th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT 2002), 6-8 November 2002, Vancouver, BC, Canada, Proceedings*. IEEE Computer Society, 2002, p. 51-59. ISBN : 0-7695-1831-1. DOI : 10.1109/DFTVS.2002.1173501. URL : https://doi.org/10.1109/DFTVS.2002.1173501.

[Bey+21] Tim Beyne, Yu Long Chen, Christoph Dobraunig et Bart Mennink. « Multi-user Security of the Elephant v2 Authenticated Encryption Mode ». In : *Selected Areas in Cryptography - 28th International Conference, SAC 2021, Virtual Event, September 29 - October 1, 2021, Revised Selected Papers*. Sous la dir. de Riham AlTawy et Andreas Hülsing. T. 13203. Lecture Notes in Computer Science. Springer, 2021, p. 155-178. ISBN : 978-3-030-99276-7. DOI : 10.1007/978-3-030-99277-4_8. URL : https://doi.org/10.1007/978-3-030-99277-4_8.

[Bih97] Eli Biham. « A Fast New DES Implementation in Software ». In : *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*. Sous la dir. d'Eli Biham. T. 1267. Lecture Notes in Computer Science. Springer, 1997, p. 260-272. ISBN : 3-540-63247-6. DOI : 10.1007/BFb0052352. URL : https://doi.org/10.1007/BFb0052352.

[Bog+07a] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin et C. Vikkelsoe. « PRESENT : An Ultra-Lightweight Block Cipher ». In : *Cryptographic Hardware and Embedded Systems - CHES 2007*. Sous la dir. de Pascal Paillier et Ingrid Verbauwhede. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 450-466. ISBN : 978-3-540-74735-2.

[Bog+07b] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin et C. Vikkelsoe. « PRESENT : An Ultra-Lightweight Block Cipher ». In : *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*. Sous la dir. de Pascal Paillier et Ingrid Verbauwhede. T. 4727. Lecture Notes in Computer Science. Springer, 2007, p. 450-466. ISBN :

978-3-540-74734-5. DOI : 10.1007/978-3-540-74735-2_31. URL : https://doi.org/10.1007/978-3-540-74735-2_31.

[Bor+12] Julia BORGHOFF, Anne CANTEAUT, Tim GÜNEYSU, Elif Bilge KAVUN, Miroslav KNEZEVIC, Lars R. KNUDSEN, Gregor LEANDER, Ventzislav NIKOV, Christof PAAR, Christian RECHBERGER, Peter ROMBOUTS, Søren S. THOMSEN et Tolga YALÇIN. « PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract ». In : *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. Sous la dir. de Xiaoyun WANG et Kazue SAKO. T. 7658. Lecture Notes in Computer Science. Springer, 2012, p. 208-225. ISBN : 978-3-642-34960-7. DOI : 10.1007/978-3-642-34961-4_14. URL : https://doi.org/10.1007/978-3-642-34961-4_14.

[BS97] Eli BIHAM et Adi SHAMIR. « Differential Fault Analysis of Secret Key Cryptosystems ». In : *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Sous la dir. de Burton S. Kaliski JR. T. 1294. Lecture Notes in Computer Science. Springer, 1997, p. 513-525. ISBN : 3-540-63384-7. DOI : 10.1007/BFb0052259. URL : https://doi.org/10.1007/BFb0052259.

[BW99] Alex BIRYUKOV et David A. WAGNER. « Slide Attacks ». In : *Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings*. Sous la dir. de Lars R. KNUDSEN. T. 1636. Lecture Notes in Computer Science. Springer, 1999, p. 245-259. ISBN : 3-540-66226-X. DOI : 10.1007/3-540-48519-8_18. URL : https://doi.org/10.1007/3-540-48519-8_18.

[CB07] Nicolas T COURTOIS et Gregory V BARD. « Algebraic cryptanalysis of the data encryption standard ». In : *Cryptography and Coding : 11th IMA International Conference, Cirencester, UK, December 18-20, 2007. Proceedings 11*. Springer. 2007, p. 152-169.

[CCZ98] Claude CARLET, Pascale CHARPIN et Victor ZINOVIEV. « Codes, bent functions and permutations suitable for DES-like cryptosystems ». In : *Designs, Codes and Cryptography* 15 (1998), p. 125-156.

[Che+23] Yukun CHENG, Changhai OU, Fan ZHANG et Shihui ZHENG. « DLPFA : Deep Learning based Persistent Fault Analysis against Block Ciphers ». In : *IACR Cryptol. ePrint Arch.* (2023), p. 21. URL : https://eprint.iacr.org/2023/021.

[Cla07] Christophe CLAVIER. « Secret External Encodings Do Not Prevent Transient Fault Analysis ». In : *Cryptographic Hardware and Embedded Systems - CHES 2007*. Sous la dir. de Pascal PAILLIER et Ingrid VERBAUWHEDE. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007, p. 181-194. ISBN : 978-3-540-74735-2.

[Col+19] Brice COLOMBIER, Alexandre MENU, Jean-Max DUTERTRE, Pierre-Alain MOELLIC, Jean-Baptiste RIGAUD et Jean-Luc DANGER. « Laser-induced Single-bit Faults in Flash Memory : Instructions Corruption on a 32-bit Microcontroller ». en. In : *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. McLean, VA, USA : IEEE,

mai 2019, p. 1-10. ISBN : 978-1-5386-8064-3. DOI : 10.1109/HST.2019.8741030. URL : https://ieeexplore.ieee.org/document/8741030/ (visité le 18/11/2021).

[Col+21] Brice COLOMBIER, Paul GRANDAMME, Julien VERNAY, Émilie CHANAVAT, Lilian BOSSUET, Lucie de LAULANIÉ et Bruno CHASSAGNE. « Multi-spot laser fault injection setup : New possibilities for fault injection attacks ». In : *International Conference on Smart Card Research and Advanced Applications*. Springer. 2021, p. 151-166.

[CP19] Anne CANTEAUT et Léo PERRIN. « On CCZ-equivalence, extended-affine equivalence, and function twisting ». In : *Finite Fields and Their Applications* 56 (2019), p. 209-246. ISSN : 1071-5797. DOI : https://doi.org/10.1016/j.ffa.2018.11.008. URL : https://www.sciencedirect.com/science/article/pii/S1071579718301485.

[DC07] Christophe DE CANNIÈRE. « Analysis and design of symmetric encryption algorithms ». In : *Doctoral Dissertaion, KULeuven* (2007).

[DCBP06] Christophe DE CANNIERE, Alex BIRYUKOV et Bart PRENEEL. « An introduction to block cipher cryptanalysis ». In : *Proceedings of the IEEE* 94.2 (2006), p. 346-356.

[Dob+18a] Christoph DOBRAUNIG, Maria EICHLSEDER, Hannes GROSS, Stefan MANGARD, Florian MENDEL et Robert PRIMAS. « Statistical Ineffective Fault Attacks on Masked AES with Fault Countermeasures ». In : *Advances in Cryptology – ASIACRYPT 2018*. Sous la dir. de Thomas PEYRIN et Steven GALBRAITH. Cham : Springer International Publishing, 2018, p. 315-342. ISBN : 978-3-030-03329-3.

[Dob+18b] Christoph DOBRAUNIG, Maria EICHLSEDER, Thomas KORAK, Stefan MANGARD, Florian MENDEL et Robert PRIMAS. « SIFA : Exploiting Ineffective Fault Inductions on Symmetric Cryptography ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), 547–572. DOI : 10.13154/tches.v2018.i3.547-572. URL : https://tches.iacr.org/index.php/TCHES/article/view/7286.

[Dob+21] Christoph DOBRAUNIG, Maria EICHLSEDER, Florian MENDEL et Martin SCHLÄFFER. « Ascon v1.2 : Lightweight Authenticated Encryption and Hashing ». In : *J. Cryptol.* 34.3 (2021), p. 33. DOI : 10.1007/s00145-021-09398-9. URL : https://doi.org/10.1007/s00145-021-09398-9.

[DR00] Joan DAEMEN et Vincent RIJMEN. « Rijndael for AES ». In : *The Third Advanced Encryption Standard Candidate Conference, April 13-14, 2000, New York, New York, USA*. National Institute of Standards et Technology, 2000, p. 343-348. URL : http://csrc.nist.gov/encryption/aes/round2/conf3/aes3conf.htm.

[DR02] Joan DAEMEN et Vincent RIJMEN. *The Design of Rijndael : AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. ISBN : 3-540-42580-2. DOI : 10.1007/978-3-662-04722-4. URL : https://doi.org/10.1007/978-3-662-04722-4.

[End+14] Sho ENDO, Naofumi HOMMA, Yu-ichi HAYASHI, Junko TAKAHASHI, Hitoshi FUJI et Takafumi AOKI. « A Multiple-Fault Injection Attack by Adaptive Timing Control Under Black-Box Conditions and a Countermeasure ». In : *Constructive Side-Channel Analysis and Secure Design*.

Sous la dir. d'Emmanuel PROUFF. Cham : Springer International Publishing, 2014, p. 214-228. ISBN : 978-3-319-10175-0.

[Fuh+13] Thomas FUHR, Éliane JAULMES, Victor LOMNÉ et Adrian THILLARD. « Fault Attacks on AES with Faulty Ciphertexts Only ». In : *2013 Workshop on Fault Diagnosis and Tolerance in Cryptography* (2013), p. 108-118. URL : https://api.semanticscholar.org/CorpusID:14777490.

[Gir05] Christophe GIRAUD. « DFA on AES ». In : *Advanced Encryption Standard – AES*. Sous la dir. d'Hans DOBBERTIN, Vincent RIJMEN et Aleksandra SOWA. Berlin, Heidelberg : Springer Berlin Heidelberg, 2005, p. 27-41. ISBN : 978-3-540-31840-8.

[Guo+11] Jian GUO, Thomas PEYRIN, Axel POSCHMANN et Matthew J. B. ROBSHAW. « The LED Block Cipher ». In : *Cryptographic Hardware and Embedded Systems - CHES 2011 - 13th International Workshop, Nara, Japan, September 28 - October 1, 2011. Proceedings*. Sous la dir. de Bart PRENEEL et Tsuyoshi TAKAGI. T. 6917. Lecture Notes in Computer Science. Springer, 2011, p. 326-341. ISBN : 978-3-642-23950-2. DOI : 10.1007/978-3-642-23951-9_22. URL : https://doi.org/10.1007/978-3-642-23951-9_22.

[Iwa+20] Tetsu IWATA, Mustafa KHAIRALLAH, Kazuhiko MINEMATSU et Thomas PEYRIN. « Duel of the Titans : The Romulus and Remus Families of Lightweight AEAD Algorithms ». In : *IACR Transactions on Symmetric Cryptology* 2020.1 (2020), 43–120. DOI : 10.13154/tosc.v2020.i1.43-120. URL : https://tosc.iacr.org/index.php/ToSC/article/view/8560.

[Kwa00] Matthew KWAN. « Reducing the Gate Count of Bitslice DES ». In : *IACR Cryptol. ePrint Arch.* (2000), p. 51. URL : http://eprint.iacr.org/2000/051.

[Lac+17] Benjamin LAC, Anne CANTEAUT, Jacques J. A. FOURNIER et Renaud SIRDEY. *Thwarting Fault Attacks using the Internal Redundancy Countermeasure (IRC)*. Cryptology ePrint Archive, Paper 2017/910. https://eprint.iacr.org/2017/910. 2017. URL : https://eprint.iacr.org/2017/910.

[Mai+22] Laurent MAINGAULT, Stéphanie ANCEAU, Manuel SULMONT, Luc SALVO, Jessy CLEDIERE, Pierre LHUISSIER, Emrick BELIARD et Jean Luc RAINARD. « Laboratory X-rays Operando Single Bit Attacks on Flash Memory Cells ». en. In : *Smart Card Research and Advanced Applications*. Sous la dir. de Vincent GROSSO et Thomas PÖPPELMANN. T. 13173. Series Title : Lecture Notes in Computer Science. Cham : Springer International Publishing, 2022, p. 139-150. ISBN : 978-3-030-97347-6 978-3-030-97348-3. DOI : 10.1007/978-3-030-97348-3_8. URL : https://link.springer.com/10.1007/978-3-030-97348-3_8 (visité le 05/04/2022).

[Men+20] Alexandre MENU, Jean-Max DUTERTRE, Jean-Baptiste RIGAUD, Brice COLOMBIER, Pierre-Alain MOELLIC et Jean-Luc DANGER. « Single-bit Laser Fault Model in NOR Flash Memories : Analysis and Exploitation ». en. In : *2020 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*. Milan, Italy : IEEE, sept. 2020, p. 41-48. ISBN : 978-1-72819-562-9. DOI : 10.1109/FDTC51366.2020.00013. URL : https://ieeexplore.ieee.org/document/9237312/ (visité le 08/11/2022).

[NFR07] Giorgio Di Natale, Marie-Lise Flottes et Bruno Rouzeyre. « A Novel Parity Bit Scheme for SBox in AES Circuits ». In : *Proceedings of the 10th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2007), Kraków, Poland, April 11-13, 2007*. Sous la dir. de Patrick Girard, Andrzej Krasniewski, Elena Gramatová, Adam Pawlak et Tomasz Garbolino. IEEE Computer Society, 2007, p. 267-271. isbn : 1-4244-1161-0. doi : 10.1109/DDECS.2007.4295295. url : https://doi.org/10.1109/DDECS.2007.4295295.

[Nov05] Harris Nover. « Algebraic cryptanalysis of AES : an overview ». In : *University of Wisconsin, USA* (2005), p. 1-16.

[Pat+17] Conor Patrick, Bilgiday Yuce, Nahid Farhady Ghalaty et Patrick Schaumont. « Lightweight Fault Attack Resistance in Software Using Intra-instruction Redundancy ». In : *Selected Areas in Cryptography – SAC 2016*. Sous la dir. de Roberto Avanzi et Howard Heys. Cham : Springer International Publishing, 2017, p. 231-244. isbn : 978-3-319-69453-5.

[Pet83] Fabien Petitcolas. « La cryptographie militaire ». In : *J. des Sci. Militaires* 9 (1883), p. 161-191.

[Pou+11] F. Poucheret, L. Chusseau, B. Robisson et P. Maurine. « Local electromagnetic coupling with CMOS integrated circuits ». In : *2011 8th Workshop on Electromagnetic Compatibility of Integrated Circuits*. Nov. 2011, p. 137-141.

[RSD06] Chester Rebeiro, A. David Selvakumar et A. S. L. Devi. « Bitslice Implementation of AES ». In : *Cryptology and Network Security, 5th International Conference, CANS 2006, Suzhou, China, December 8-10, 2006, Proceedings*. Sous la dir. de David Pointcheval, Yi Mu et Kefei Chen. T. 4301. Lecture Notes in Computer Science. Springer, 2006, p. 203-212. isbn : 3-540-49462-6. doi : 10.1007/11935070_14. url : https://doi.org/10.1007/11935070_14.

[SA03] Sergei P. Skorobogatov et Ross J. Anderson. « Optical Fault Induction Attacks ». In : *Cryptographic Hardware and Embedded Systems - CHES 2002*. Sous la dir. de Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Burton S. Kaliski, çetin K. Koç et Christof Paar. T. 2523. Series Title : Lecture Notes in Computer Science. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 2-12. isbn : 978-3-540-00409-7 978-3-540-36400-9. doi : 10.1007/3-540-36400-5_2. url : http://link.springer.com/10.1007/3-540-36400-5_2 (visité le 23/11/2021).

[SH] Jörn-Marc Schmidt et Michael Hutter. « Optical and EM Fault-Attacks on CRT-based RSA : Concrete Results ». en. In : ().

[Sim+20] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni et Niels Samwel. « Friet : An Authenticated Encryption Scheme with Built-in Fault Detection ». In : *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*. Sous la dir. d'Anne Canteaut et Yuval Ishai. T. 12105. Lecture Notes in Computer Science. Springer,

2020, p. 581-611. ISBN : 978-3-030-45720-4. DOI : 10.1007/978-3-030-45721-1_21. URL : https://doi.org/10.1007/978-3-030-45721-1_21.

[SL22] Dor SALOMON et Itamar LEVI. « On the Performance Gap of a Generic C Optimized Assembler and Wide Vector Extensions for Masked Software with an Ascon-{\it{p}} test case ». In : *IACR Cryptol. ePrint Arch.* (2022), p. 124. URL : https://eprint.iacr.org/2022/124.

[Sta+77] Data Encryption STANDARD et al. « Federal information processing standards publication 46 ». In : *National Bureau of Standards, US Department of Commerce* 23 (1977), p. 1-18.

[TBG23a] Pierre-Antoine TISSOT, Lilian BOSSUET et Vincent GROSSO. « BALoo : First and Efficient Countermeasure dedicated to Persistent Fault Attacks ». In : *IACR Cryptol. ePrint Arch.* (2023), p. 944. URL : https://eprint.iacr.org/2023/944.

[TBG23b] Pierre-Antoine TISSOT, Lilian BOSSUET et Vincent GROSSO. « Generalized Code-Abiding Countermeasure ». In : *Electronics* 12.4 (fév. 2023), p. 976. ISSN : 2079-9292. DOI : 10.3390/electronics12040976. URL : http://dx.doi.org/10.3390/electronics12040976.

[Ull10] MarkUS ULLRICH. « The design and efficient software implementation of S-boxes ». Thèse de doct. Ph. D. thesis, KU Leuven, 2010.

[Zha+18] Fan ZHANG, Xiaoxuan LOU, Xinjie ZHAO, Shivam BHASIN, Wei HE, Ruyi DING, Samiya QURESHI et Kui REN. « Persistent Fault Analysis on Block Ciphers ». In : *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.3 (2018), p. 150-172. DOI : 10.13154/tches.v2018.i3.150-172. URL : https://doi.org/10.13154/tches.v2018.i3.150-172.

[Zha+20] Fan ZHANG, Yiran ZHANG, Huilong JIANG, Xiang ZHU, Shivam BHASIN, Xinjie ZHAO, Zhe LIU, Dawu GU et Kui REN. « Persistent Fault Attack in Practice ». In : *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.2 (mars 2020), 172–195. DOI : 10.13154/tches.v2020.i2.172-195. URL : https://tches.iacr.org/index.php/TCHES/article/view/8548.

[Zha+23] Fan ZHANG, Run HUANG, Tianxiang FENG, Xue GONG, Yulong TAO, Kui REN, Xinjie ZHAO et Shize GUO. « Efficient Persistent Fault Analysis with Small Number of Chosen Plaintexts ». In : *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2023.2 (2023), p. 519-542. DOI : 10.46586/tches.v2023.i2.519-542. URL : https://doi.org/10.46586/tches.v2023.i2.519-542.