



Implémentation d'une contre-mesure face à une attaque par analyse de la consommation de puissance d'un circuit intégré de chiffrement AES

Travail de fin d'étude présenté par
Thomas ANIZET
En vue de l'obtention du diplôme de
Master en Sciences de l'Ingénieur Industriel orientation Electronique

Abstract

Imaginez que de simples analyses de paramètres physiques (température, consommation de puissance, rayonnement électromagnétique) permettent de révéler les secrets de votre carte d'identité ou de votre carte bancaire ? Ceci est aujourd'hui possible depuis la découverte fin des années 1990 d'un nouveau type d'attaque ciblant les algorithmes de chiffrement : les attaques par canaux auxiliaires. À ce jour, il existe une multitude d'algorithmes de chiffrement. L'algorithme *AES (Advanced Encryption Standard)* est sans nul doute le plus réputé et le plus répandu, de nos jours, dans la majeure partie des systèmes embarqués comportant des applications en sécurité. L'attaque par *l'analyse de la consommation de puissance* est un exemple parmi d'autres d'attaques par canaux auxiliaires d'un système implémentant l'algorithme AES. Cette attaque analyse la consommation de puissance du circuit cryptographique lorsque celui-ci chiffre les données. En effet, la puissance consommée par le circuit intégré reflète directement ses activités internes (instructions exécutées et données manipulées). Depuis ces découvertes, la sécurité matérielle a pris une tournure particulière. C'est désormais sur un nouvel axe de recherche, s'écartant des sentiers traditionnels, que se porte la problématique de protection des données sensibles.

Étant un acteur majeur dans le domaine de la Défense, Thales collabore étroitement avec de nombreux gouvernements. La cybersécurité est au centre des préoccupations actuelles. Le besoin en implantations sécurisées se fait de plus en plus ressentir. Il est nécessaire de se prémunir contre les attaques classiques mais également contre les attaques de plus bas niveau exploitant des caractéristiques physiques. L'attaque ciblée dans ce *Travail de Fin d'Étude* (TFE) concerne l'analyse de la consommation de puissance. Mon objectif est le suivant : dans un premier temps, réaliser une série de démonstrateurs permettant de mettre en évidence l'impact des failles non traitées. Dans un second temps, développer une contre-mesure et justifier son gain en sécurité.

Remerciements

Je tiens à remercier toutes les personnes qui ont permis à ce travail de voir le jour, mais qui m'ont également soutenu tout au long de ces 5 années d'étude à l'ECAM.

Je remercie *Thales Belgium Communications & Security* de m'avoir permis de réaliser mon stage et mon TFE.

Je remercie tout particulièrement Monsieur Liran Lerman, mon promoteur, de m'avoir proposé ce projet très intéressant et qui fut très enrichissant. Je le remercie pour son support précieux, sa disponibilité mais également pour ses conseils et ses encouragements quotidiens.

Je remercie Monsieur François Durvaux, pour ses précieux conseils et ses connaissances avancées dans le domaine des attaques par canaux auxiliaires.

Je remercie Monsieur Arnaud Withoeck, pour son temps précieux consacré à m'apprendre de nouveaux concepts sur la programmation orientée hardware.

Je remercie également monsieur Nicolas Marchand, ma tuteur, qui m'a accompagné et conseillé dans la réalisation de ce travail et qui m'a consacré son temps et son attention durant la période de TFE.

Cahier des charges relatif au travail de fin d'études de

Thomas ANIZET, inscrit en 2^{ème} Master, orientation électronique

- Année académique : 2018-2019
- Titre provisoire : Contre-mesure pour les attaques par canaux cachés
- Objectifs à atteindre : Étant un acteur majeur dans le domaine de la Défense, Thales collabore étroitement avec de nombreux gouvernements. La cybersécurité est au centre des préoccupations actuelles. Le besoin en implémentations sécurisées se fait donc de plus en plus ressentir. L'objectif est de se prémunir contre des attaques classiques et des attaques plus bas niveau exploitant des caractéristiques physiques. Afin de se familiariser avec ce domaine, l'étudiant devra réaliser une recherche bibliographique de règles de bonnes pratiques pour de la programmation sécurisée hardware. Après en avoir étudié les tenants et aboutissants, l'étudiant réalisera (*i*) une série de démonstrateurs permettant de mettre en évidence l'impact des failles non traitées et (*ii*) le gain en sécurité dû aux contre-mesures.
- Principales étapes :
 - Recherches bibliographiques de règles de bonnes pratiques.
 - Implémentation de l'algorithme AES sur FPGA.
 - Réalisation d'une attaque CPA (Correlation Power Analysis).
 - Étude et choix de métrique(s) pour l'analyse de contre-mesures.
 - Développement d'une contre-mesure.
 - Analyse et conclusion sur la contre-mesure développée.

Fait en trois exemplaires à Tubize, le 22 Novembre 2018.

Table des matières

Abstract	I
Remerciements	III
Cahier des charges	IV
Liste des acronymes	XII
1 Introduction	1
1.1 Présentation du problème	1
1.2 Structure du travail	3
I Contexte théorique	4
2 Cryptologie	5
2.1 Concepts et définitions	5
2.1.1 Chiffrement symétrique	6
2.1.2 Chiffrement asymétrique	7
2.2 Algorithme de chiffrement	8
2.2.1 Principe	8
2.2.2 <i>Device cryptographique</i>	8
2.2.3 Algorithme <i>AES (Advanced Encryption Standard)</i>	9
3 Consommation de puissance	14
3.1 Technologie CMOS	14
3.1.1 Consommation de puissance des circuits en technologie CMOS	15
3.1.2 Composition des traces de puissance	18
3.2 Modèles de puissance	19
3.2.1 Modèle de poids de Hamming	19
3.2.2 Modèle de distance de Hamming	19

4 Attaques par canaux auxiliaires	20
4.1 Introduction	20
4.2 Analyse de la consommation de puissance	23
4.2.1 Analyse simple de la consommation	23
4.2.2 Analyse différentielle de la consommation	23
4.2.3 Analyse de la consommation par corrélation	24
5 Contre-mesures	29
5.1 Introduction	29
5.2 Contre-mesures Hiding	32
5.3 Contre-mesures Masking	33
5.4 Contre-mesures Faking	34
5.4.1 Définition	34
5.4.2 Points faibles et améliorations	36
II Mise en application	38
6 Mise en oeuvre de l'attaque CPA	39
6.1 Langages de programmation	39
6.2 Procédure de chiffrement et configuration de l'oscilloscope	39
6.3 Procédure appliquée pour l'attaque CPA	44
7 Attaque CPA sur l'implémentation de l'AES-256	45
7.1 Implémentation de l'algorithme AES-256 et du contrôleur	45
7.1.1 Implémentation AES-256	45
7.1.2 Implémentation contrôleur	47
7.2 Résultats de simulations	48
7.2.1 Traces simulées idéales	48
7.2.2 Traces simulées avec addition de bruit	50
7.3 Résultats expérimentaux	52
7.3.1 Traces réelles brutes	52
7.3.2 Traces réelles filtrées	55
7.3.3 Taux de succès de l'attaque	58
8 Implémentation de la contre-mesure de type <i>faking</i>	59
8.1 Implémentation de la contre-mesure	59
8.2 Résultats de simulations	61
8.3 Résultats expérimentaux	63
8.3.1 Architecture parallèle	63
8.3.2 Architecture séquentielle	65

9 Évaluation de la contre-mesure	67
9.1 Simulations (<i>test benches</i>)	67
9.2 Évaluation par critères de performances	70
10 Conclusion	72
Crédits	73
Références	73
Annexes	76

Table des figures

1.1	Matériel nécessaire pour réaliser une attaque par analyse de la consommation de puissance.	2
2.1	Concepts de chiffrement et de déchiffrement.	6
2.2	Chiffrement symétrique : Une seule clé est utilisée pour chiffrer et déchiffrer les messages.	7
2.3	Chiffrement asymétrique : Un clé publique est utilisée pour chiffrer le message et une clé privée est utilisée pour le déchiffrer.	7
2.4	Concept d'un algorithme de chiffrement.	8
2.5	Fonctionnement général de l'algorithme AES.	9
2.6	Les 3 matrices utilisées par l'algorithme AES.	10
2.7	Opération <i>AddRoundKey</i> entre la matrice STATE et la matrice clé.	10
2.8	Opération <i>SubBytes</i> exécutée sur la matrice STATE.	11
2.9	Opération <i>ShiftRows</i> exécutée sur la matrice STATE.	11
2.10	Opération <i>MixColumns</i> exécutée sur la matrice STATE.	12
2.11	Exemple de l'opération <i>MixColumns</i> exécutée sur la deuxième colonne (i.e colonne 1) de la matrice STATE.	12
3.1	Schéma de l'inverseur CMOS.	15
3.2	Comparaison de la puissance statique et dynamique.	18
4.1	Classification des attaques physiques.	21
4.2	Différentes façons d'opérer une attaque non invasive sur un device cryptographique.	22
4.3	Principe de fonctionnement d'une attaque DPA.	24
4.4	Principe de mesure à l'oscilloscope.	25
4.5	Principe de fonctionnement d'une attaque CPA.	27
4.6	Différents résultats de corrélation en fonction de 4 clés testées.	28
4.7	Corrélation en fonction des 256 clés testées.	28
5.1	Modification de l'amplitude d'une trace de puissance.	30
5.2	Modification de la position dans le temps d'une trace de puissance.	30
5.3	Importance du bruit pour un signal de transmission.	31

5.4	Implémentation de la contre-mesure <i>faking</i> pour l'algorithme de chiffrement AES	35
5.5	Implémentation de la contre-mesure <i>faking</i> pour l'algorithme de chiffrement AES avec prise en compte des améliorations.	37
5.6	Valeur de la clé obtenue avec implémentation d'une contre-mesure <i>faking</i>	37
6.1	Carte SAKURA-G.	40
6.2	Les trois points de mesures prévus pour analyser la consommation de puissance (la tension pour être exact) du FPGA principal. Seul le connecteur SMA J_3 est utilisé.	40
6.3	Le Picoscope est l'instrument de mesure utilisé pour analyser la consommation de puissance (la tension pour être exact) du FPGA.	41
6.4	Interface logicielle utilisée pour le Picoscope. La trace de puissance affichée correspond à l'exécution de l'algorithme AES-256 sur le FPGA.	41
6.5	Formules permettant de déterminer l'intervalle d'échantillonnage (tableaux repris de la <i>datasheet</i> du Picoscope).	42
6.6	Banc de mesures.	43
6.7	Procédure mise en place pour le chiffrement de textes clairs et l'enregistrement des traces de puissance capturées à l'oscilloscope.	43
6.8	Procédure mise en place pour réaliser une attaque par analyse de la consommation de puissance de type CPA.	44
7.1	Représentation haut niveau du bloc de chiffrement.	46
7.2	Architecture de l'algorithme AES-256 implémentée sur le FPGA principal.	46
7.3	Chaîne de communication entre le l'ordinateur et le FPGA principal.	47
7.4	Résultat de l'attaque CPA sur le dixième byte de la clé. Ce graphe présente la valeur du coefficient de corrélation pour chacune des 256 valeurs de clé testées. La valeur de corrélation maximum vaut 1 et correspond à la clé 200. 1000 traces sont utilisées pour obtenir ce résultat.	48
7.5	Influence du nombre de traces employées pour la réalisation de l'attaque CPA. Plus le nombre de traces augmente, plus la corrélation obtenue pour la bonne clé testée est prédominante par rapport aux 255 autres.	49
7.6	Influence du nombre de traces employées pour la réalisation de l'attaque CPA. Plus le nombre de traces augmente, plus il est facile de retrouver la valeur de la clé secrète.	50
7.7	Attaque sur le dixième byte de la clé avec pour objectif la mise en évidence de l'influence du bruit sur le résultat de l'attaque CPA. La présence du bruit complexifie la tâche de l'attaquant.	51
7.8	Taux de succès de l'attaque CPA en fonction du nombre de traces utilisées. Les 3 courbes présentées sur le graphe correspondent à des valeurs différentes de σ (pour le bruit ajouté). La courbe orange présente le résultat pour $\sigma = 15$, qui correspond à l'exemple présenté à la figure 7.7.	51
7.9	Trace brute capturée à l'oscilloscope. Cette trace présente la tension consommée par le FPGA lorsque celui-ci est en cours de chiffrement.	52
7.10	Résultat de l'attaque CPA sur le quatorzième byte de la clé. Ce graphe présente la valeur du coefficient de corrélation pour chacune des 256 valeurs de clé testées. La valeur de corrélation maximum vaut -0,085 et révèle la valeur décimale 13 pour le quatorzième byte de la clé.	53
7.11	Attaque CPA sur le quatorzième byte de la clé de chiffrement. Ce graphe indique qu'il faut un minimum de 1400 traces à l'attaquant pour que l'attaque CPA fonctionne tout le temps.	54
7.12	Comparaison dans le domaine temporel entre une trace brute et une trace filtrée. Le filtre FIR de type passe-bas a pour effet de <i>lisser</i> le signal.	55

7.13	Transformée de Fourier discrète pour une trace brute.	56
7.14	Transformée de Fourier discrète pour une trace filtrée.	56
7.15	Attaque CPA sur le quatorzième byte de la clé de chiffrement. Ce graphe indique qu'il faut un minimum de 1300 traces à l'attaquant pour que l'attaque CPA fonctionne tout le temps.	57
7.16	Taux de succès de l'attaque CPA selon que les traces soient préalablement filtrées (courbe bleue) ou non (courbe orange). L'attaque CPA fonctionne mieux avec filtre.	58
8.1	Représentation haut niveau du bloc de chiffrement lorsque la contre-mesure <i>faking</i> est implémentée.	60
8.2	Architecture de l'algorithme AES-256 avec la contre-mesure <i>faking</i> exécutée en mode parallèle.	60
8.3	Architecture de l'algorithme AES-256 avec la contre-mesure <i>faking</i> exécutée en mode séquentiel.	61
8.4	Valeurs de corrélation pour chacune des 256 valeurs de clé testées (attaque CPA sur un byte de la clé). La fausse clé (125) est révélée tandis que le vraie clé (25) ne laisse fuiter aucune information.	62
8.5	Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La fausse clé (courbe rouge) est retrouvée à partir de 600 traces alors que le vraie clé ne laisse fuiter aucune information.	62
8.6	Trace obtenue lorsque la contre-mesure <i>faking</i> est implémentée de façon parallèle.	63
8.7	Valeurs de corrélation pour chacune des 256 valeurs de clé testées. La clé révélée (138) ne correspond ni à la vraie clé (4) ni à la fausse clé (84).	64
8.8	Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La vraie clé (courbe verte) n'est pas révélée. La fausse clé (courbe rouge) n'est pas non plus révélée. La clé révélée (courbe magenta) est un résultat inattendu.	64
8.9	Trace obtenue lorsque la contre-mesure <i>faking</i> est implémentée de façon séquentielle.	65
8.10	Valeurs de corrélation pour chacune des 256 valeurs de clé testées. La clé révélée (201) ne correspond ni à la vraie clé (4) ni à la fausse clé (84).	66
8.11	Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La vraie clé (courbe verte) n'est pas révélée. La fausse clé (courbe rouge) n'est pas non plus révélée. La clé révélée (courbe magenta) est un résultat inattendu.	66
9.1	Schéma-bloc du principe de fonctionnement du test automatisé mis en place.	68
9.2	Affichage via la logiciel VIVADO des différents ports/signaux utilisés dans l'implémentation de la contre-mesure en mode parallèle. Le graphe montre bien que les opérations principales sont exécutées en parallèle.	69
9.3	Affichage via la logiciel VIVADO des différents ports/signaux utilisés dans l'implémentation de la contre-mesure en mode séquentiel. Le graphe montre bien que les opérations sont exécutées en séquentiel.	69
9.4	Architecture simpliste d'un FPGA.	70

Liste des tableaux

1.1	Conséquences de l'implémentation d'une contre-mesure.	3
2.1	Les trois variantes de l'algorithme AES.	9
3.1	Table de vérité pour la fonction NON (inverseur CMOS).	14
3.2	Type de puissance consommée par une cellule CMOS en fonction des 4 transitions d'état de sa sortie.	16
5.1	Les contre-mesures de type <i>hiding</i> sont utilisées pour rendre aléatoire ou égale la consommation de puissance du device cryptographique..	32
6.1	Paramètres de configuration pour les canaux A, B et Ext du Picoscope.	42
7.1	Identification des données utiles au bloc de chiffrement par l'entremise d'une entête sur un byte.	47
9.1	Tableau récapitulatif des performances observées pour les implémentations de l'AES seul, de l'AES avec contre-mesure <i>faking</i> en mode parallèle et de l'AES avec contre-mesure <i>faking</i> en mode séquentiel.	71

Liste des acronymes

Encore à compléter ...

AES	Advanced Encryption Standard
CMOS	Complementary Metal Oxyde Semiconductor
CPA	Correlation Power Analysis
DPA	Differential Power Analysis
FPGA	Field Programmable Gate Array
NIST	National Institute of Standard and Technology
RSA	Rivest Shamir Adleman
SPA	Simple Power Analysis
SNR	Signal-to-Noise Ratio
VHDL	VHSIC Hardware Description Language
XOR	Exclusive OR

Introduction

1.1 Présentation du problème

Le monde dans lequel nous vivons aujourd’hui est un monde où tout doit être connecté. Notre smartphone, notre montre, notre TV, notre frigo, notre tondeuse, ... Tout système d’électronique embarquée est potentiellement connectable ! Les transferts d’informations sont par conséquent de plus en plus gourmands. Les technologies ne cessent d’évoluer. Le meilleur exemple est le développement de l’IoT (*Internet of Things*) ou encore le développement du réseau 5G, prévu entre-autres pour augmenter les débits de données. Cependant, nous sommes en droit de nous poser plusieurs questions : toutes ces données qui voyagent de système en système sont-elles protégées ? Toutes ces données qui transitent entre les différents appareils connectés de ma maison peuvent-ils être captées par des pirates informatiques ? La réponse à ces deux questions reste bien souvent assez vague.

Depuis toujours, l’homme cherche à dissimuler des informations à ses tiers mais depuis toujours, l’homme cherche également à accéder à des informations auxquelles il n’a pas droit. Cette lutte symbolise le combat entre la **cryptographie** et la **cryptanalyse**. Rappelons brièvement que la cryptographie a pour objectif de crypter de l’information, c’est-à-dire qu’elle vise à élaborer des procédés assurant la confidentialité de données sensibles. La cryptanalyse quant à elle vise à décrypter des informations sensibles auxquelles elle n’a normalement pas accès. Ainsi, lorsqu’il s’agit de manipuler des informations secrètes, la question d’intégrité et de confidentialité des informations manipulées se pose inévitablement. C’est pour cette raison que des **algorithmes de chiffrement** sont développés et par la suite, la cryptanalyse tente d’y déceler des failles. À ce jour, il existe une multitude d’algorithmes de chiffrement, chacun possédant ses avantages et ses inconvénients. Celui étudié dans ce travail est l’**algorithme AES** (*Advanced Encryption Standard*).

Initialement, la cryptanalyse se basait essentiellement sur la compréhension des procédés mathématiques utilisés dans les algorithmes en vue d’y déceler des failles. Devant ces attaques, les ingénieurs renforçèrent la complexité des algorithmes. Par ailleurs, fin du millénaire précédent, le *NIST* (*National Institute of Standards and Technology*) a encouragé le développement d’un nouveau standard d’algorithme de chiffrement difficilement envisageable à casser : l’algorithme *AES*. Sa robustesse aux attaques classiques lui permet d’être considéré aujourd’hui comme l’algorithme le plus connu au monde et implémenté dans la majeure partie des systèmes embarqués comportant des applications en sécurité. Cependant, fin des années 1990, de nouvelles recherches dans le domaine de la cryptanalyse ont été réalisées. Ces recherches ont abouti sur un nouveau type d’attaque : les **attaques par canaux auxiliaires**. Une attaque par canal auxiliaire désigne une attaque informatique qui, sans remettre en cause la robustesse théorique des méthodes et procédures de sécurité, recherche et exploite des failles dans leur implantation logicielle ou matérielle.

Il existe différents types d'attaques par canaux auxiliaires cependant ce travail se concentre sur un type d'attaque en particulier : les **attaques par analyse de la consommation de puissance**. Comme son nom l'indique, cette attaque analyse la consommation de puissance d'un *système cryptographique* en cours de chiffrement. En effet, la puissance consommée par le circuit intégré reflète directement ses activités internes (instructions exécutées et données manipulées). Par la suite, on appellera *device cryptographique* tout système cryptographique implémentant un bloc de chiffrement tel que l'AES (définition 2.2.2). Cette puissance consommée peut être capturée à l'aide d'un instrument de mesure tel que l'oscilloscope. Les mesures prélevées à l'oscilloscope portent le nom de *traces de puissance*. Lorsque la clé d'un algorithme de chiffrement est révélée, le contenu censé rester confidentiel devient accessible à quiconque le souhaite. Ainsi, dès que des traces de puissance sont acquises, il est possible pour un attaquant, via divers calculs, de retrouver la clé utilisée pour le chiffrement des données. Attirons l'attention sur le fait que l'attaque exploite les faiblesses du système cryptographique, c'est-à-dire du système implémentant l'algorithme de chiffrement (l'AES en l'occurrence). En aucun cas, l'attaque exploite les failles des principes mathématiques mis en place. La figure 1.1 présente les liens entre les différents systèmes nécessaires pour réaliser l'attaque.

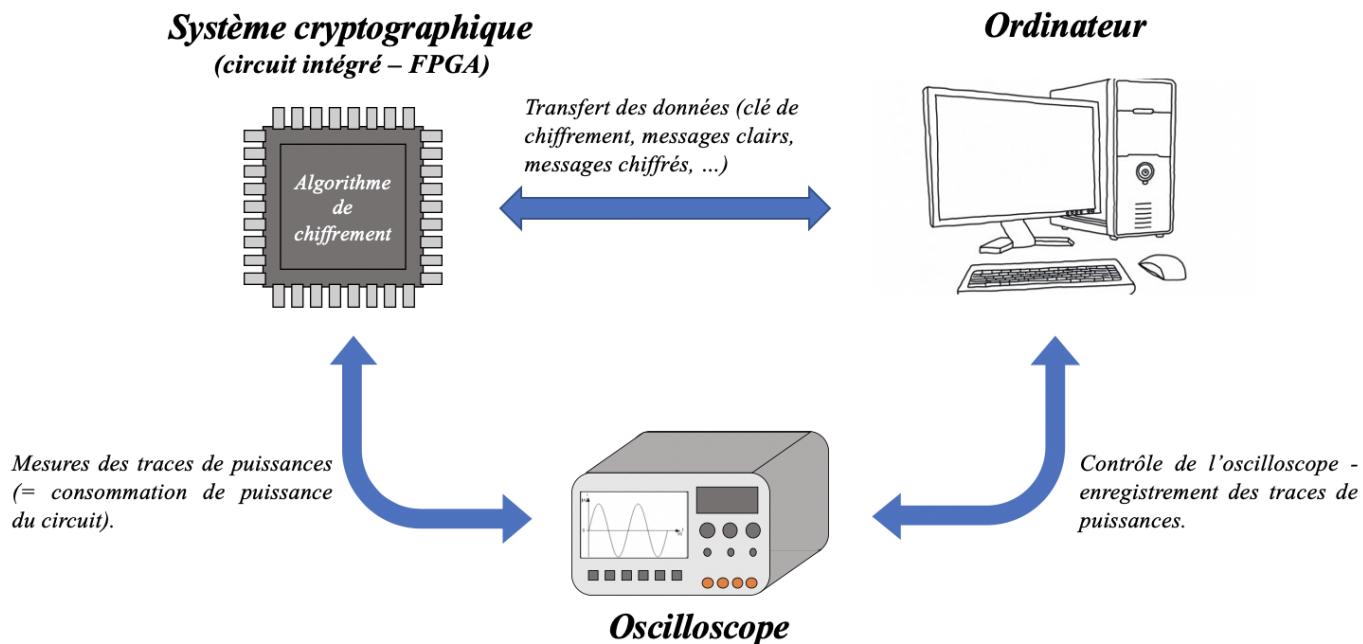


Figure 1.1 : Matériel nécessaire pour réaliser une attaque par analyse de la consommation de puissance.

Dès que ces attaques se sont manifestées, une série de contre-mesures a été développée. Pour obtenir une contre-mesure efficace contre ces attaques par analyse de la consommation de puissance, il est nécessaire de bien appréhender le fonctionnement de l'attaque. Ainsi, retenons que lors de l'exécution d'un algorithme de chiffrement, différentes opérations sont exécutées conduisant à différentes **valeurs intermédiaires calculées**. Si aucune protection n'est mise en place, la consommation de puissance du device cryptographique dépend des données intermédiaires manipulées par ce dernier (par l'algorithme précisément). Par conséquent, l'attaquant peut potentiellement retrouver la clé secrète en analysant la consommation de puissance du système. Dès lors, l'objectif général d'une contre-mesure est de rendre la consommation de puissance du device cryptographique indépendante des valeurs intermédiaires calculées par l'algorithme de chiffrement.

Deux grands types de contre-mesures sont à distinguer : les contre-mesures de type **Masking** et les contre-mesures de type **Hiding**. Brièvement, une contre-mesure de type *Masking* modifie les caractéristiques de l'algorithme de chiffrement alors qu'une contre-mesure de type *Hiding* modifie les caractéristiques physiques du système cryptographique. La contre-mesure développée dans ce travail est une contre-mesure moderne : la contre-mesure de type *Faking*. Le principe de fonctionnement de cette contre-mesure se rapproche de celui de contre-mesure de type *Masking*. Cependant, il existe quelques subtilités qui seront expliquées au chapitre 5. Le tableau 1.1 résume les constats qui peuvent être tirés selon qu'une contre-mesure soit implantée ou non. Ces contre-mesures seront détaillées plus en profondeur au chapitre y faisant référence (chapitre 5).

Avec/Sans contre-mesure	Conséquences
Sans contre-mesure	La consommation de puissance dépend directement des valeurs intermédiaires calculées par l'algorithme de chiffrement.
Avec contre-mesure type Hiding	Modification des caractéristiques de consommation : La consommation de puissance du device cryptographique est identique (ou aléatoire) pour chaque instruction exécutée et chaque donnée manipulée.
Avec contre-mesure type Masking	Modification de l'algorithme : Les valeurs intermédiaires calculées par l'algorithme de chiffrement sont masquées par des valeurs aléatoires. La consommation de puissance est alors modifiée (les traces capturées à l'oscilloscope sont insensées).
Avec contre-mesure type Faking	Modification de l'algorithme : La consommation de puissance du device cryptographique permet de trouver une valeur de clé secrète. Cependant, la clé obtenue est volontairement faussée en appliquant un masque sur la vraie clé.

Table 1.1 : Conséquences de l'implémentation d'une contre-mesure.

Enfin, pour juger de **l'efficacité de la contre-mesure implémentée**, différentes analyses peuvent être réalisées. Ainsi, des métriques telles que le *taux de succès* peuvent être utilisées afin de connaître le taux de succès de l'attaque selon un nombre de traces donné. La mise en place d'un système de tests automatisé est également réalisée afin de démontrer que l'implémentation de la contre-mesure n'entache pas le chiffrement des messages clairs. Enfin, une analyse des performances du système sera également établie. Les principales performances étudiées sont : le temps d'exécution de l'algorithme, la consommation de puissance du device cryptographique, la place mémoire de l'implémentation protégée de l'algorithme sur le device crytpographique, ...

1.2 Structure du travail

Ce travail est scindé en deux parties :

1. La première partie aborde tous les concepts théoriques nécessaires afin de comprendre le fondement des attaques par canaux auxiliaires. Un premier chapitre expose les concepts et définitions de la cryptologie et définit en particulier le fonctionnement de l'algorithme AES. Le second chapitre est consacré à la consommation de puissance des circuits en technologie CMOS. Cette analyse de la consommation de puissance pour ce type de circuit en particulier permet de comprendre le principe de fonctionnement des attaques par analyse de la consommation de puissance. Le troisième chapitre définit le sujet principal de ce travail, à savoir les attaques par canaux auxiliaires. Enfin, le dernier chapitre étudie les différentes contre-mesures existantes afin d'assurer une meilleure protection des données sensibles.
2. La seconde partie aborde l'application de l'attaque CPA et l'étude de la solution *faking*. Un premier chapitre définit la configuration du matériel nécessaire pour implémenter l'attaque par analyse de la consommation de puissance. Un descriptif succinct des langages et matériels utilisés est ainsi détaillé. Le second chapitre reprend les résultats obtenus et leurs conclusions pour la mise en application d'une attaque CPA (*Correlation Power Analysis*). Le troisième chapitre détaille l'implémentation de la contre-mesure de type *faking*. Le quatrième chapitre discute de l'efficacité de la contre-mesure implémentée.

Première partie

Contexte théorique

Chapitre 2

Cryptologie

Ce chapitre introduit les fondamentaux en matière de cryptologie. Les différents concepts et définitions abordés définissent les bases théoriques. Ce chapitre est scindé en deux parties. Dans un premier temps, il abordera les définitions importantes en matière de cryptologie et plus spécifiquement en matière de cryptographie. La cryptographie symétrique sera ainsi distinguée de la cryptographie asymétrique. Dans un second temps, ce chapitre détaillera le principe de fonctionnement général d'un algorithme de chiffrement. Une attention particulière sera finalement consacrée à la compréhension de l'algorithme AES.

2.1 Concepts et définitions

Il y a plus de 2000 ans naissait le concept de **cryptologie** avec la montée au pouvoir de Jules César. Général romain très puissant, César marqua le monde et l'histoire universelle au travers de ses nombreuses conquêtes. Tant adulé par son peuple, César s'était néanmoins fait plus d'un ennemi durant ses conquêtes. Le 15 mars 44 avant J-C, il fut tragiquement assassiné par une conspiration de sénateurs romains dont son fils, Marcus Junius Brutus. Jules César avait pris pour habitude de se méfier de ses ennemis. Arrivé à un stade de méfiance extrême, il n'avait plus confiance en ses propres messagers. C'est pour cette raison qu'il envoyait des messages chiffrés dans lesquels il décalait chaque lettre d'un mot de 3 lettres dans l'alphabet. Seul le destinataire, au courant de la règle de chiffrement, pouvait alors déchiffrer les messages. Ce fut le premier procédé de chiffrement de l'histoire, appelé *Chiffre de César* (voir annexe A.1). C'est ainsi que tout commença...

Étymologiquement, le mot "*cryptologie*" vient du grec et signifie littéralement la *science du secret* (*kryptos* pour caché et *logos* pour science). La cryptologie regroupe deux champs d'études diamétralement opposés : la cryptographie et la cryptanalyse.

Définition 2.1.1. Cryptographie : *La cryptographie est la science qui utilise les mathématiques en vue de chiffrer et déchiffrer de l'information secrète.*

Définition 2.1.2. Cryptanalyse : *La cryptanalyse est la science qui analyse les procédés mathématiques mis en place par la cryptographie en vue de retrouver de l'information secrète.*

Six termes couramment employés en cryptologie doivent également être définis :

Texte clair : Il s'agit de données brutes, considérées comme confidentielles, compréhensibles sans intervention spécifique.

Texte chiffré : Il s'agit de données modifiées, considérées comme confidentielles et devant le rester. Ces données sont incompréhensibles, inintelligibles pour quiconque qui tenterait de les comprendre.

Chiffrement (cryptage) : Procédé utilisé pour dissimuler le texte clair. Le chiffrement transforme donc le texte clair en texte chiffré.

Déchiffrement (décryptage) : Procédé inverse au chiffrement. Le déchiffrement transforme donc le texte chiffré en texte clair, c'est-à-dire en texte d'origine.

Algorithmes de chiffrement : Fonctions mathématiques utilisées lors du processus de chiffrement. Les fonctions mathématiques inverses sont également définies pour le processus de déchiffrement.

Clé de chiffrement : Paramètre associé à l'algorithme de chiffrement pour le cryptage des textes clairs ou pour le décryptage des textes chiffrés.

La figure 2.1 illustre cinq des six termes fondamentaux énoncés. Le sixième, l'algorithme de chiffrement, pourrait être confondu avec les étapes de *chiffrement* et *déchiffrement*. Cependant, une explication plus détaillée est donnée à la section 2.2.



Figure 2.1 : Concepts de chiffrement et de déchiffrement.

Revenons sur la notion de cryptographie. La cryptographie doit remplir trois conditions majeures qui sont :

- **Confidentialité** : La confidentialité spécifie que seules les personnes autorisées à accéder à une certaine information ont la possibilité de l'atteindre.
- **Intégrité** : L'intégrité spécifie que l'information, lors de son traitement ou de sa transmission, ne subit aucun modification volontaire ou accidentelle. Autrement dit, il s'agit d'une garantie que l'information conserve son état d'origine.
- **Disponibilité** : La disponibilité spécifie que l'information est en tout temps disponible pour une personne autorisée à y accéder et en demandant l'accès.

Ainsi, la cryptographie a pour objectif de chiffrer des informations tout en assurant la confidentialité, l'intégrité et la disponibilité de ces informations. La cryptographie se divise en deux catégories distinctes : la cryptographie symétrique et la cryptographie asymétrique.

2.1.1 Chiffrement symétrique

Définition 2.1.3. Chiffrement symétrique : Le chiffrement est dit symétrique lorsque la clé utilisée pour le chiffrement et le déchiffrement des données est la même. La clé est alors appelée clé secrète.

Par convention, ce type de chiffrement permet à la fois de chiffrer et de déchiffrer des messages à partir d'une seule et unique clé : la clé secrète. Le désavantage de ce type de chiffrement est que si une personne parvient à subtiliser la clé, elle sera en mesure de déchiffrer tout message qu'elle intercepte.

Exemple : L'algorithme AES. Une explication plus détaillée de cet algorithme est reprise à la section 2.2.3.

La figure 2.2 présente le principe de fonctionnement du chiffrement symétrique. Si un attaquant intercepte un message, ce dernier sera chiffré. Il doit donc être en possession de la clé secrète s'il souhaite comprendre le message envoyé par l'expéditeur.



Figure 2.2 : Chiffrement symétrique : Une seule clé est utilisée pour chiffrer et déchiffrer les messages.

2.1.2 Chiffrement asymétrique

Définition 2.1.4. Chiffrement asymétrique : Le chiffrement est dit asymétrique lorsqu'il existe une clé publique utilisée pour le chiffrement des textes clairs et une clé privée utilisée pour le déchiffrement des textes chiffrés.

Par convention, la clé publique est la clé de chiffrement du message clair, elle peut être communiquée sans aucune restriction tandis que la clé privée est la clé de déchiffrement du message chiffré, elle ne doit être communiquée sous aucun prétexte. Le fonctionnement est le suivant : Avec une clé publique, l'expéditeur envoie un message chiffré selon un algorithme de chiffrement préalablement défini. Ce message, une fois transmis, ne pourra être déchiffré que par le destinataire, détenteur de la clé privée.

Exemple : L'algorithme RSA (Rivest Shamir Adleman).

La figure 2.3 ci-dessous présente le principe de fonctionnement du chiffrement asymétrique.



Figure 2.3 : Chiffrement asymétrique : Un clé publique est utilisée pour chiffrer le message et une clé privée est utilisée pour le déchiffrer.

2.2 Algorithme de chiffrement

2.2.1 Principe

Définition 2.2.1. *Algorithme de chiffrement* : Un algorithme de chiffrement est un ensemble de fonctions mathématiques utilisé lors du processus de chiffrement et de déchiffrement de données sensibles. Une clé publique ou privée est associée à l'algorithme de chiffrement selon qu'il s'agisse d'un chiffrement symétrique ou asymétrique.

Les systèmes de sécurité modernes utilisent des algorithmes de chiffrement pour assurer la confidentialité et l'intégrité de données sensibles. Ces algorithmes prennent typiquement :

- 2 paramètres en entrée : un *message clair* et une *clé de chiffrement*.
- 1 paramètre en sortie : le *message chiffré*.

Comme décrit précédemment, le procédé transformant les données claires en entrée en données chiffrées en sortie est appelé *chiffrement*. Ce procédé est réalisé grâce à un algorithme de chiffrement manipulant une clé de chiffrement, un message clair et diverses opérations mathématiques. Avec des clés différentes, le résultat du cryptage variera également. La figure 2.4 ci-dessous conceptualise le principe de fonctionnement d'un algorithme de chiffrement.

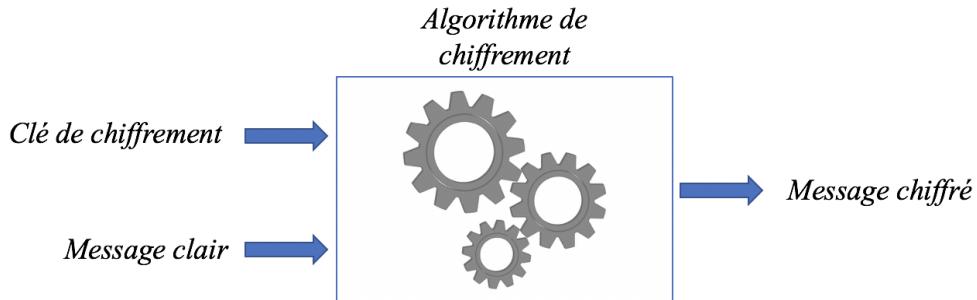


Figure 2.4 : Concept d'un algorithme de chiffrement.

Il est important de préciser que tous les détails décrivant le fonctionnement d'un algorithme cryptographique sont disponibles publiquement, seule la clé de chiffrement doit rester secrète. En effet, un principe fondamental de la cryptographie est le principe de Kerckhoffs. Ce principe stipule que la sécurité de tout algorithme de chiffrement ne doit pas reposer sur la connaissance du fonctionnement de ce dernier. Autrement dit, la sécurité offerte par un algorithme de chiffrement ne doit pas dépendre du secret de son implémentation mais doit uniquement reposer sur la protection de la clé, ou plus généralement d'un secret. Par conséquent, un bon algorithme est un algorithme dont on ne parviendra pas à déchiffrer les données chiffrées, même en connaissant son fonctionnement interne. Lorsque la clé d'un algorithme est trouvée, le déchiffrement des données confidentielles devient possible. On dit que l'algorithme de chiffrement est *cassé*.

2.2.2 Device cryptographique

Que ce soit pour un algorithme de chiffrement symétrique ou asymétrique, la clé de chiffrement tout comme l'algorithme en lui-même doivent être stockés sur un support physique. Ce support, appelé *device cryptographique*, doit être suffisamment sécurisé que pour contenir la clé de manière protégée. Par la suite, les messages clairs seront envoyés au device cryptographique qui se chargera de les chiffrer selon l'algorithme implémenté et la clé utilisée. Un exemple de device cryptographique utilisé est le *FPGA* (*Field-Programmable Gate Array*).

Définition 2.2.2. *Device cryptographique* : Un device cryptographique est un système cryptographique qui implémente des algorithmes de chiffrement et qui stocke des clés de chiffrement.

2.2.3 Algorithme AES (*Advanced Encryption Standard*)

En 1997, le NIST (*National Institute of Standards and Technology*) décida qu'il était temps de développer un nouveau standard d'algorithme de chiffrement. Ce nouveau standard, nommé **AES** (pour *Advanced Encryption Standard*), était appelé à remplacer l'ancien standard de chiffrement, l'algorithme DES (pour *Data Encryption Standard*). Pour ce faire, le NIST organisa un concours cryptographique. Les chercheurs du monde entier furent invités à soumettre leurs propositions. En Octobre 2000, le NIST annonça le vainqueur du concours : l'algorithme de Rijndael, du nom de ses concepteurs Joan Daemen et Vincent Rijmen, tous deux de nationalité belge.

L'algorithme de Rijndael, désormais plus connu sous le nom d'algorithme AES, est un **algorithme de chiffrement symétrique par blocs**. Par *blocs* signifie que les données sont traitées par blocs de 128 bits. La clé secrète peut posséder différentes tailles : 128 bits (AES-128), 192 bits (AES-192) ou encore 256 bits (AES-256). À noter qu'en théorie, plus la taille de la clé est élevée, moins il y a de chance de casser l'algorithme. Cela est dû au fait que, plus la taille de la clé est grande, plus il existe un nombre important de possibilités de clé à tester avant de retrouver celle qui correspond à la valeur exacte. L'algorithme AES n'a ainsi, à ce jour, jamais été cassé par des méthodes d'attaques classiques. Cependant, comme nous le verrons par la suite avec les attaques par canal auxiliaire (section 4.1), le problème peut vite être contourné. Pour ce travail, il m'a été demandé d'attaquer une implémentation de l'algorithme AES-256. La description qui suit est, dans un premier temps basée sur l'algorithme AES-128. Dans un second temps, une adaptation sera insérée afin de comprendre le fonctionnement de l'algorithme AES-256.

Avant de détailler le fonctionnement de l'algorithme AES-128, précisons-en quelques caractéristiques principales. L'algorithme AES est caractérisé par une série de tours (*rounds* en anglais) dépendant de la taille de la clé. Pour une clé dont la taille est 128 bits, on dénombre 10 tours. Pour une clé de taille 192 bits, on dénombre 12 tours et pour une clé de taille 256 bits, on dénombre 14 tours. Un *round* est défini par 4 opérations appliquées succinctement sur une matrice de données. Ces 4 opérations sont : *AddRoundKey*, *SubBytes*, *ShiftRows* et *MixColumns*. Elles sont appliquées à divers instants dans l'exécution de l'algorithme AES. Le tableau 2.1 présente un récapitulatif des principales caractéristiques des trois variantes de l'algorithme AES. La figure 2.5 permet de visualiser l'ordre d'exécution chronologique des 4 opérations opérées par l'algorithme. Les annexes A.2 et A.3 reprennent respectivement le code réalisé pour l'exécution de l'algorithme AES-128 et l'algorithme AES-256.

	Taille de la clé (bits)	Taille du bloc de données (bits)	Nombre de <i>rounds</i>
AES-128	128	128	10
AES-192	192	128	12
AES-256	256	128	14

Table 2.1 : Les trois variantes de l'algorithme AES.



Figure 2.5 : Fonctionnement général de l'algorithme AES.

L’AES-128 a donc pour rôle de chiffrer des blocs de données de 128 bits avec une clé de 128 bits. Les données et la clé sont représentées par une matrice où chaque élément de la matrice correspond à un byte (un octet, i.e 8 bits). Étant donné que 128 bits correspondent à 16 bytes, la matrice de données, au même titre que la matrice de clé, correspond à une matrice de 4 lignes et 4 colonnes (formant ainsi les 4x4 soit 16 bytes). Une matrice particulière (de taille 4x4 également) appelée STATE contient l’ensemble des résultats intermédiaires résultant des diverses opérations que subissent les données (depuis leur état initial). La figure 2.6 présente les 3 matrices qui viennent d’être citées : la matrice de données (message clair initial de 128 bits), la matrice STATE (qui va contenir les résultats intermédiaires des données suite aux différentes opérations) et la matrice clé (clé de 128 bits).

d_0	d_4	d_8	d_{12}
d_1	d_5	d_9	d_{13}
d_2	d_6	d_{10}	d_{14}
d_3	d_7	d_{11}	d_{15}

Matrice de données

S_0	S_4	S_8	S_{12}
S_1	S_5	S_9	S_{13}
S_2	S_6	S_{10}	S_{14}
S_3	S_7	S_{11}	S_{15}

Matrice STATE

k_0	k_4	k_8	k_{12}
k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}

Matrice clé

Figure 2.6 : Les 3 matrices utilisées par l’algorithme AES.

Remarque : En pratique, la matrice de données est directement confondue avec la matrice STATE. Autrement dit, les premiers éléments à être placés dans la matrice STATE représentent les bytes de données. Ainsi, on n’utilise que deux matrices durant le fonctionnement de l’algorithme AES : la matrice STATE et la matrice clé.

Fonctionnement AES-128 :

Initialement, deux matrices vont être utilisées : la matrice STATE, contenant les données claires, et la matrice clé, contenant la clé secrète initiale. Comme on peut le voir sur la figure 2.5, la première opération à être appliquée sur ces deux matrices est l’opération *AddRoundKey*. Cette opération réalise un XOR (symbole \oplus) entre chaque élément de la matrice STATE et chaque élément respectif de la matrice clé. Le résultat est ré-écrit dans la matrice STATE. La figure 2.7 ci-dessous présente le principe de fonctionnement de cette première opération :

12	e0	13	28
04	ab	f8	d3
23	19	69	26
e5	b9	7a	4c

\oplus

ab	88	23	2a
10	fe	a3	6c
fe	2c	39	75
17	b1	39	05

=

b9	68	30	02
14	55	5b	bf
dd	35	50	53
f2	08	43	49

Matrice STATE

Matrice clé

Nouvelle matrice STATE

Figure 2.7 : Opération *AddRoundKey* entre la matrice STATE et la matrice clé.

Ensuite, toujours selon la figure 2.5, une série de quatre opérations se répétant neuf fois (car AES-128) est exécutée. Ces quatres opérations sont appliquées dans l’ordre chronologique suivant : *SubBytes*, *ShiftRows*, *MixColumns*, *AddRoundKey*.

Enfin, une fois les neufs rounds exécutés, trois opérations terminent le chiffrement : *SubBytes*, *ShiftRows* et *AddRoundKey*. À la fin de ces trois dernières opérations, une matrice de taille 4x4 (la matrice STATE) présente le message chiffré de 128 bits.

Description des 4 opérations :

- 1. SubBytes** : La figure 2.8 ci-dessous présente le principe de fonctionnement de l'opération *SubBytes*. Le principe de fonctionnement présenté dans ce cas-ci est simple : il repose sur une table de substitution, appelée Sbox et présentée en annexe A.4. La matrice STATE avant l'exécution de l'opération contient 16 bytes. Chacun de ses 16 bytes (notés $S_{i,j}$) va fournir une nouvelle valeur de byte (noté $S'_{i,j}$) en fonction de la table de substitution. Un exemple est donné afin de comprendre le principe : Si le byte $S_{1,2}$ vaut 53 (en hexadécimal) alors, selon la table Sbox, la valeur du byte résultant $S'_{1,2}$ vaut ed (en hexadécimal).

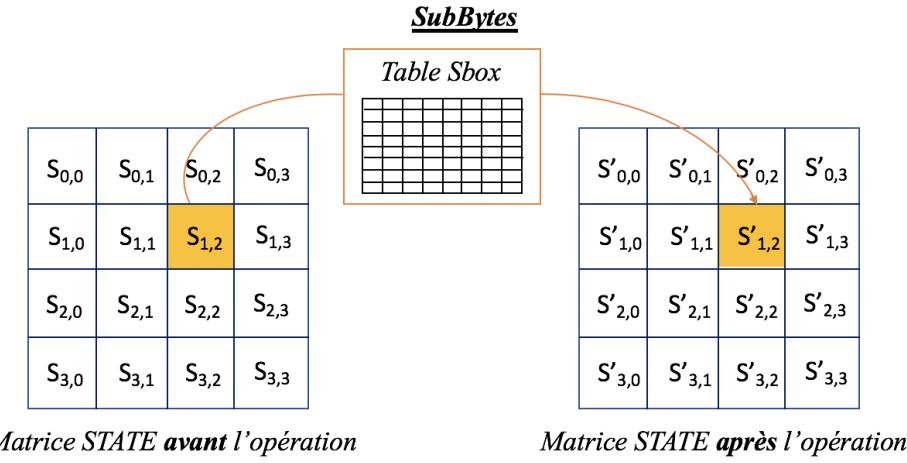


Figure 2.8 : Opération *SubBytes* exécutée sur la matrice STATE.

- 2. ShiftRows** : Comme son nom l'indique, cette opération concerne les lignes de la matrice STATE. Cette opération réalise une permutation cyclique des octets sur les lignes de la matrice STATE. Plus précisément, pour la *i*-ième ligne, on décalera chaque élément de la matrice STATE de *i* positions vers la gauche, en considérant que la première ligne a pour indice 0. La figure 2.9 ci-dessous présente le principe de fonctionnement de l'opération *ShiftRows* :

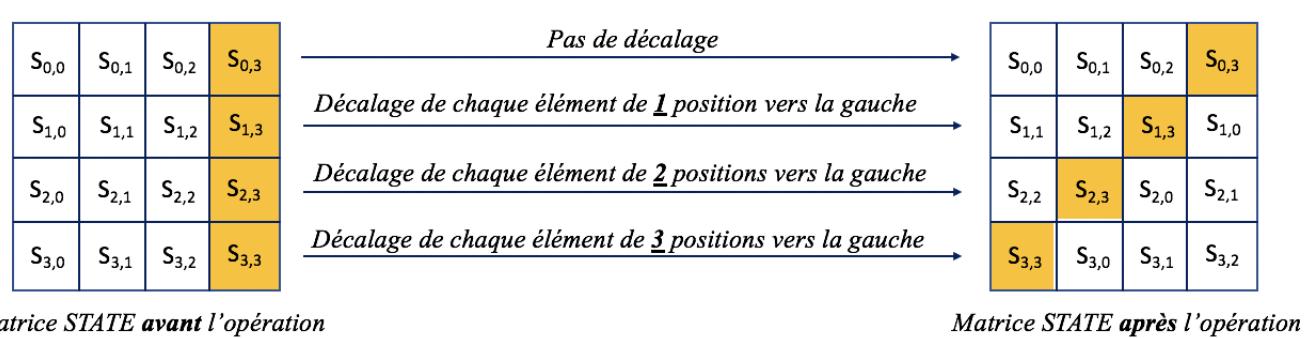


Figure 2.9 : Opération *ShiftRows* exécutée sur la matrice STATE.

3. MixColumns : Comme son nom l'indique, cette opération concerne les colonnes de la matrice STATE. Cette opération réalise un produit matriciel entre une matrice fixée (taille 4x4) définie ci-dessous (figure 2.11) et un vecteur colonne (taille 4x1) de la matrice STATE. Il en résulte un nouveau vecteur colonne (taille 4x1) permettant de définir la nouvelle matrice STATE. La figure 2.10 ci-dessous présente le principe de fonctionnement de l'opération *MixColumns*. Un exemple est ensuite donné à la figure 2.11.



Figure 2.10 : Opération *MixColumns* exécutée sur la matrice STATE.

$$\begin{array}{c}
 \left[\begin{array}{cccc} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{array} \right] \quad \left[\begin{array}{c} S_{0,1} \\ S_{1,1} \\ S_{2,1} \\ S_{3,1} \end{array} \right] = \left[\begin{array}{c} S'_{0,1} \\ S'_{1,1} \\ S'_{2,1} \\ S'_{3,1} \end{array} \right]
 \end{array}$$

Matrice fixée Colonne 1 de la matrice STATE avant opération Colonne 1 de la matrice STATE après opération

Exemple : 1^{er} élément du vecteur colonne

$$S'_{0,1} = (02 * S_{0,1}) + (03 * S_{1,1}) + (01 * S_{2,1}) + (01 * S_{3,1})$$

Figure 2.11 : Exemple de l'opération *MixColumns* exécutée sur la deuxième colonne (i.e colonne 1) de la matrice STATE.

4. AddRoundKey : Comme précisé précédemment dans le fonctionnement initial, l'opération *AddRoundKey* réalise un XOR entre la matrice STATE et la matrice de clé. Le résultat de ce XOR est placé dans la nouvelle matrice STATE (voir figure 2.7).

L'opération *KeySchedule* :

Les quatre opérations principales ont été expliquées. Il reste l'opération *KeySchedule* qui s'exécute uniquement sur la matrice clé. La gestion des clés se fait donc au travers des deux fonctions qui sont les fonctions *AddRoundKey* et *KeySchedule*.

Jusqu'à présent, la matrice STATE était modifiée après chaque opération exécutée. Il en est de même pour la matrice clé, qui est également modifiée à chaque round afin d'éviter d'utiliser toujours la même valeur de clé. Cela permet ainsi de complexifier le chiffrement des données. C'est l'opération ***KeySchedule*** qui modifie la matrice clé initiale en une nouvelle matrice clé. Cette opération génère une nouvelle clé sur base de la clé précédemment employée. Ainsi, pour générer la première nouvelle clé, l'opération *KeySchedule* emploiera la clé initialement donnée. Étant donné que pour l'AES-128, nous avons un total de dix rounds à exécuter, cela signifie que l'opération *KeySchedule* s'exécute dix fois afin de générer dix nouvelles clés de 128 bits. En rajoutant la clé initiale, nous avons alors onze clés de chiffrement différentes permettant d'exécuter les onze opérations *AddRoundKey* présentes dans l'algorithme AES-128. Ainsi, comme présenté à l'annexe A.2, l'opération *KeySchedule* s'exécute en premier lieu dans le code afin de générer dix nouvelles clés. En comptant la clé initialement donnée, il y a donc onze clés. Chacune de ces onze clés sera utilisée lors de l'appel de la fonction *AddRoundKey*.

L'annexe A.5 permet de comprendre le principe de fonctionnement de l'opération *KeySchedule*, utilisée afin de générer les nouvelles clés. Plus précisément, cette annexe présente un exemple pour générer la première nouvelle clé (taille de 128 bits). Le fonctionnement est le suivant : l'opération *KeySchedule* s'exécute colonne par colonne sur la matrice clé de taille 4x4. On va d'abord générer la première colonne de la nouvelle clé, ensuite on générera les colonnes 2, 3 et 4. C'est la première colonne qui est la plus compliquée à générer. Les 3 autres colonnes réalisent simplement un XOR pour générer la nouvelle colonne. Ainsi, pour obtenir la 1ère colonne de la nouvelle clé (soit les quatre premiers bytes), on va prendre trois vecteurs colonnes (taille 4x1), à savoir :

1. Prendre la 4ème colonne de la clé initiale.
 - (a) Décaler chaque élément de cette colonne d'une position vers le haut.
 - (b) Appliquer l'opération *SubBytes* sur chaque élément de la colonne.
2. Prendre la 1ère colonne de la clé initiale.
3. Prendre la 1ère colonne de la matrice RCON.

Une fois ces trois vecteurs colonnes obtenus, on réalise un XOR entre eux. Le résultat de ce XOR représente la 1ère colonne de la nouvelle matrice clé. Pour les colonnes 2, 3 et 4, le principe est le suivant : la colonne i de la nouvelle clé est obtenue en réalisant un XOR entre la colonne i de l'ancienne clé et la colonne $i-1$ de la nouvelle clé (avec $i \in \{2; 4\}$). L'annexe A.5 donne un exemple concret et didactique de l'opération *KeySchedule*.

Adaptation pour l'AES-256 :

En se basant sur le tableau 2.1 et la figure 2.5, les deux différences majeures entre l'algorithme AES-128 et l'algorithme AES-256 sont :

- La taille de la clé : on passe de 128 bits à 256 bits.
- Le nombre de rounds : on passe de 10 rounds à 14 rounds.

Ainsi, l'opération *KeySchedule* s'exécute 14 fois. Cependant, cette opération ne génère pas 14 nouvelles clés de 256 bits mais seulement 7 nouvelles clés de 256 bits. La raison est la suivante : les 256 bits de la clé sont scindés en deux parties égales, c'est-à-dire en 128 bits, afin que les 4 autres opérations décrites précédemment (*SubBytes*, *ShiftRows*, *MixColumns* et *AddRoundKey*) puissent s'exécuter sans problème. En effet, la matrice STATE ayant toujours une taille de 128 bits, la matrice clé doit obligatoirement avoir une taille équivalente (128 bits) afin que les quatre opérations puissent être exécutées comme décrit pour l'algorithme AES-128. Le premier round utilisera donc les seize premiers bytes de la clé initiale ; Le second round utilisera les seize derniers bytes de la clé initiale (les trente-deux bytes de la clé initiale ont ainsi été utilisés) ; Le troisième round utilisera les seize premiers bytes de la seconde clé, générée par l'opération *KeySchedule* ; et ainsi de suite jusqu'au quatorzième et dernier round.

Chapitre 3

Consommation de puissance

Ce chapitre introduit une technologie bien répandue dans les systèmes informatiques modernes : la technologie CMOS. C'est en étudiant le fonctionnement de cette technologie que les attaques par analyse de la consommation de puissance sont rendues praticables. Ce chapitre se scinde en deux parties. Dans un premier temps, l'objectif est de comprendre le fonctionnement de la technologie CMOS ainsi que ses répercussions sur les différentes puissances intervenant. Dans un second temps, l'objectif est d'assimiler le concept de *modèle de puissance*. Pour ce faire, deux modèles seront judicieusement développés.

3.1 Technologie CMOS

La **technologie CMOS** (pour *Complementary Metal Oxyde Semiconductor*) est la technologie la plus répandue parmi toutes les technologies de semi-conducteurs. En effet, on la retrouve dans la majorité des systèmes informatiques modernes. En 2001, elle couvrait 86% de la production mondiale des circuits intégrés. Pour cette raison, nous nous intéressons à leur conception afin de détecter des anomalies qui pourraient se révéler être utiles pour la cryptanalyse.

Le nom de cette technologie vient du fait que toutes les fonctions logiques (portes OR, AND, etc.) peuvent être réalisées moyennant l'utilisation d'une paire de transistors MOS complémentaires (N-MOS et P-MOS) associés symétriquement et fonctionnant en régime de commutation. Cela signifie que lorsqu'un des deux transistors MOS conduit, l'autre est par conséquent fermé. Grâce à ce principe, une porte logique CMOS ne consomme de l'énergie qu'au moment de la commutation. Cette caractéristique permet de distinguer le CMOS de toutes les autres technologies.

Pour expliquer le fonctionnement de cette technologie, on peut prendre un exemple simple : **l'inverseur CMOS**. Un inverseur CMOS est une fonction "NON" dont voici la table de vérité (table 3.1) :

<i>Entrée</i>	<i>Sortie</i>
0	1
1	0

Table 3.1 : Table de vérité pour la fonction NON (inverseur CMOS).

La figure 3.1 ci-dessous (schéma a) présente le schéma de l'inverseur CMOS :

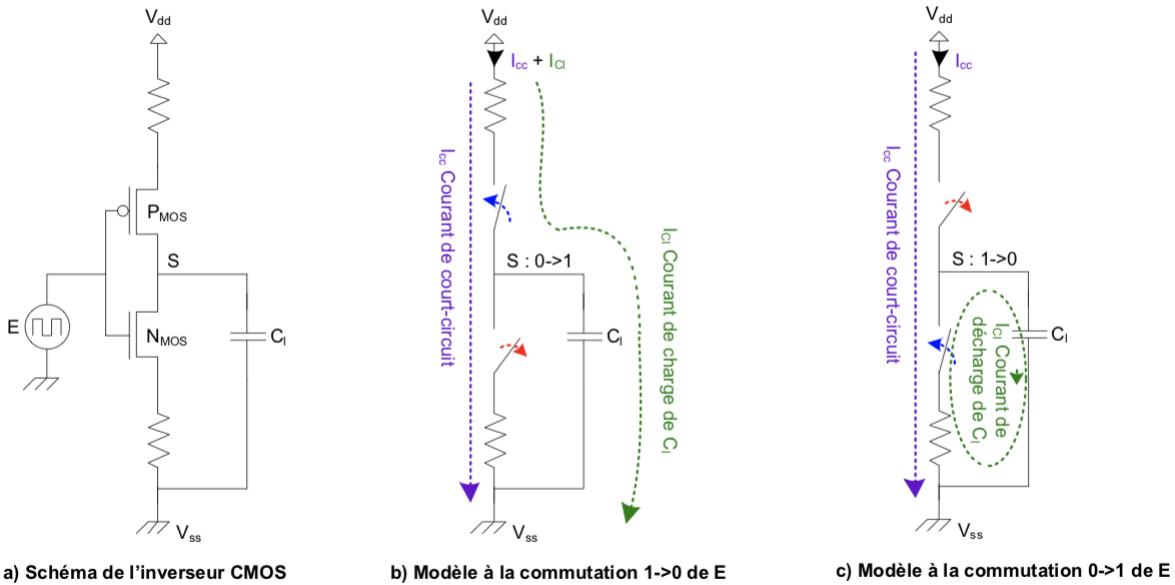


Figure 3.1 : Schéma de l'inverseur CMOS.

Si on applique à l'entrée (E) un état bas, le transistor N est bloqué et le P est passant (schéma b). On place ainsi la sortie au potentiel Vdd (la tension d'alimentation), c'est-à-dire à l'état haut. Inversement, quand on met l'entrée à l'état haut, le transistor P est bloqué et le N est passant (schéma c). La sortie est donc à l'état bas. On a donc bien réalisé une fonction inversion. Dans la suite de ce travail, on considérera toujours que les circuits attaqués sont réalisés en technologie CMOS.

3.1.1 Consommation de puissance des circuits en technologie CMOS

Il est évident que les circuits modernes consomment de la puissance lorsqu'ils exécutent des instructions. Dans notre cas, c'est un FPGA sur lequel est implanté l'algorithme AES qui consomme de la puissance. Dans le domaine de la cryptanalyse, cette puissance va être mesurée et analysée afin de déterminer si le device cryptographique laisse fuiter des informations. Si tel est le cas, nous verrons comment ces informations peuvent être utilisées afin de retrouver la clé de chiffrement de l'algorithme AES (chapitre 4). Pour ce faire, nous devons tout d'abord étudier les différentes puissances intervenant dans les circuits réalisés en technologie CMOS.

La consommation totale de puissance d'un circuit CMOS peut être obtenue en sommant les consommations de puissance respectives de chaque cellule logique (cellule conçue pour remplir une certaine fonction logique) du circuit CMOS. De cette façon, la consommation de puissance totale dépend essentiellement du nombre de cellules logiques dans le circuit CMOS.

En prenant comme exemple de cellule logique CMOS, l'inverseur CMOS (expliqué à la section 3.1), nous allons tenter de comprendre quand et pour quelles raisons ces cellules CMOS dissipent de la puissance. Pour ce faire, il faut savoir que la consommation de puissance est essentiellement divisée en deux parties :

- La puissance statique (notée P_{stat}) : C'est la puissance qui est consommée lorsqu'il n'y a pas de commutation dans une cellule (c'est-à-dire dans l'inverseur). Autrement dit, c'est la puissance qui est consommée lorsque l'inverseur est en fonctionnement normal mais ne commute pas.
- La puissance dynamique (notée P_{dyn}) : C'est la puissance qui est consommée par une cellule si la sortie de celle-ci commute.

Ainsi, la puissance totale consommée par une cellule vaut la somme de ces deux composantes, soit :

$$P_{total} = P_{stat} + P_{dyn} \quad (3.1)$$

Puissance statique :

Les cellules CMOS sont toujours construites de façon à ce que les deux transistors complémentaires ne soient jamais passants au même moment. En effet, si on reprend l'exemple de l'inverseur CMOS (3.1), lorsqu'on met le signal d'entrée à GND alors le transistor P1 est passant et N1 est bloqué. Par contre, lorsqu'on met le signal d'entrée à V_{DD} , le transistor P1 devient bloqué tandis que le transistor N1 devient passant. Ainsi, en théorie, un seul transistor fonctionne et laisse passer du courant. Cependant, en pratique, lorsqu'un transistor MOS est bloqué, le courant qui le traverse n'est jamais entièrement nul. En effet, une très petite valeur de courant circule dans le canal du transistor. Ce courant, que l'on appelle *courant de fuite* et que l'on note I_{fuite} , produit une consommation de puissance **statique** pouvant être calculée de la façon suivante :

$$P_{stat} = I_{fuite} \cdot V_{DD} \quad (3.2)$$

Ainsi, on peut conclure que la consommation de puissance statique des circuits CMOS correspond à la puissance qui est consommée par le circuit lorsqu'il n'y a pas de commutation dans une cellule. Cette puissance est typiquement très faible et sera, en pratique, négligée.

Puissance dynamique :

La consommation de puissance dynamique apparaît typiquement lors d'une commutation des transistors. Une commutation est le passage d'un état haut à un état bas ou d'un état bas à un état haut. En réalité, il existe 4 transitions d'état possibles. Ces 4 possibilités sont reprises dans le tableau 3.2 ci-dessous.

Transitions	Type de puissance consommée
$0 \rightarrow 0$	Statique
$0 \rightarrow 1$	Statique + Dynamique
$1 \rightarrow 0$	Statique + Dynamique
$1 \rightarrow 1$	Statique

Table 3.2 : Type de puissance consommée par une cellule CMOS en fonction des 4 transitions d'état de sa sortie.

On constate que pour chaque transition possible, il y a présence de puissance statique. Cependant, il n'y a présence de puissance dynamique que dans le cas d'une commutation, c'est-à-dire lors des deux transitions : $0 \rightarrow 1$ et $1 \rightarrow 0$. La consommation de puissance totale dépend du type de cellule et de la technologie employée. Cependant, en général, on constate que :

- **Lorsqu'il n'y a pas de commutation** (transitions $0 \rightarrow 0$ et $1 \rightarrow 1$), la puissance totale reste plus ou moins constante. En effet, la puissance dynamique étant nulle, on ne retrouve dans le calcul de puissance totale que la puissance statique. Autrement dit : $P_{total} = P_{stat}$.
- **Lorsqu'il y a une commutation** (transitions $0 \rightarrow 1$ et $1 \rightarrow 0$), la puissance totale augmente. En effet, en plus de la puissance statique déjà présente, vient s'ajouter la puissance dynamique. Autrement dit : $P_{total} = P_{stat} + P_{dyn}$.

Ainsi, on peut conclure que la consommation de puissance dynamique des circuits CMOS correspond à la puissance qui est consommée par le circuit lorsqu'il y a une commutation dans la cellule. Cette puissance constitue un facteur dominant dans la consommation de puissance totale. Il est donc primordial de pouvoir la calculer.

Calcul de la puissance dynamique : Le calcul de la consommation de puissance dynamique se divise en deux parties. Pour mieux comprendre pourquoi, reprenons l'exemple de l'inverseur CMOS.

1. **Puissance moyenne de chargement de la capacité :** La figure 3.1 présente le schéma de l'inverseur CMOS lorsqu'il y a une commutation de sa sortie de l'état 0 à l'état 1 (schéma b) et lorsqu'il y a une commutation de sa sortie de l'état 1 à l'état 0 (schéma c). Pour rappel, le fonctionnement de l'inverseur est le suivant : Lorsque le signal d'entrée est à 1 (ou 0), le transistor du dessous est passant (ou bloqué) alors que celui du haut est bloqué (ou passant), la sortie est donc à 0 (ou 1). Maintenant, il faut regarder dans le cas d'une commutation à la sortie de l'inverseur CMOS. Si il y a une commutation de l'état 0 à l'état 1 en sortie (schéma b), l'inverseur dessine un courant provenant de l'alimentation (V_{DD}) et venant charger le condensateur C_L . Ce courant est appelé *courant de charge*. À contrario, lors d'une commutation de l'état 1 à l'état 0, l'inverseur décharge le courant du condensateur C_L vers la masse (GND). Ainsi, on constate bien que la commutation à la sortie de l'inverseur génère un courant qui produira une partie de la consommation de puissance dynamique. La consommation de puissance moyenne de chargement de la capacité durant un temps T peut être calculée de la façon suivante (3.3) :

$$P_{chrg} = \frac{1}{T} \int_0^T p_{chrg}(t) dt = \alpha \cdot f \cdot C_L \cdot V_{DD}^2 \quad (3.3)$$

Où :

- $p_{chrg}(t)$ représente la consommation de puissance de chargement instantanée de la cellule.
- α est le facteur d'activité. Il correspond au nombre moyen de transitions (0-1) en sortie de la cellule à chaque coup de clock.
- f représente la fréquence de clock.
- C_L représente la valeur de capacité du condensateur.
- V_{DD} représente la tension positive de l'alimentation.

2. **Puissance moyenne causée par les courants de court-circuit :** En plus de la puissance moyenne de chargement de la capacité, il existe lors d'une commutation un bref instant, durant lequel les deux transistors conduisent le courant. Cela a pour effet de créer un court-circuit entre V_{DD} en GND . Ce court-circuit va dissiper, le temps de son passage, de la puissance. La consommation de puissance moyenne qui est causée par les courants de court-circuit dans une cellule durant un temps T peut être calculée de la façon suivante (3.4) :

$$P_{cc} = \frac{1}{T} \int_0^T p_{cc}(t) dt = \alpha \cdot f \cdot C_L \cdot V_{DD} \cdot I_{fuite} \cdot t_{cc} \quad (3.4)$$

Où :

- $p_{cc}(t)$ représente la puissance de court-circuit consommée par la cellule.
- α est le facteur d'activité. Il correspond au nombre moyen de transitions (0-1) en sortie de la cellule à chaque coup de clock.
- f représente la fréquence de clock.
- V_{DD} représente la tension positive de l'alimentation.
- I_{fuite} représente le courant de fuite causé par le court-circuit.
- t_{cc} représente le temps durant lequel le court-circuit se produit.

En conclusion, les devices cryptographiques modernes (comme le FPGA) possèdent deux composantes en puissance :

- Une **puissance statique** de faible valeur, requise pour garder le device en fonctionnement continu. Elle dépend du nombre de transistors dans la cellule. Elle est **négligée**.
- Une **puissance dynamique** de haute valeur, qui apparaît lors d'une commutation. Elle dépend des opérations exécutées et des données manipulées. Elle n'est **pas négligée**.

Ceci nous conduit donc à l'équation suivante (3.5) :

$$P_{total} = P_{stat} + P_{dyn} \cong P_{dyn} = P_{chrg} + P_{cc} \quad (3.5)$$

La figure 3.2 ci-dessous présente deux mesures différentes de consommation de puissance prises à l'oscilloscope. Attention, comme il sera précisé au chapitre 4 (section 4.2.3), la tension est proportionnelle à la puissance mesurée. Dès lors, ce graphe ne représente pas à proprement parler la puissance consommée mais plutôt la tension. Ces mesures ont été réalisées sur un FPGA implémentant l'algorithme AES-256. On y distingue la puissance statique seule (en bleu) de la puissance statique et dynamique (en orange). La courbe bleue a été enregistrée lorsque le FPGA était en attente de textes clairs, autrement dit il ne chiffrait rien (\Rightarrow puissance statique). La courbe orange a été enregistrée lorsque le FPGA chiffrait des données (\Rightarrow puissance statique et dynamique). On constate donc bien que la puissance statique est de très faible valeur comparativement à la puissance dynamique.

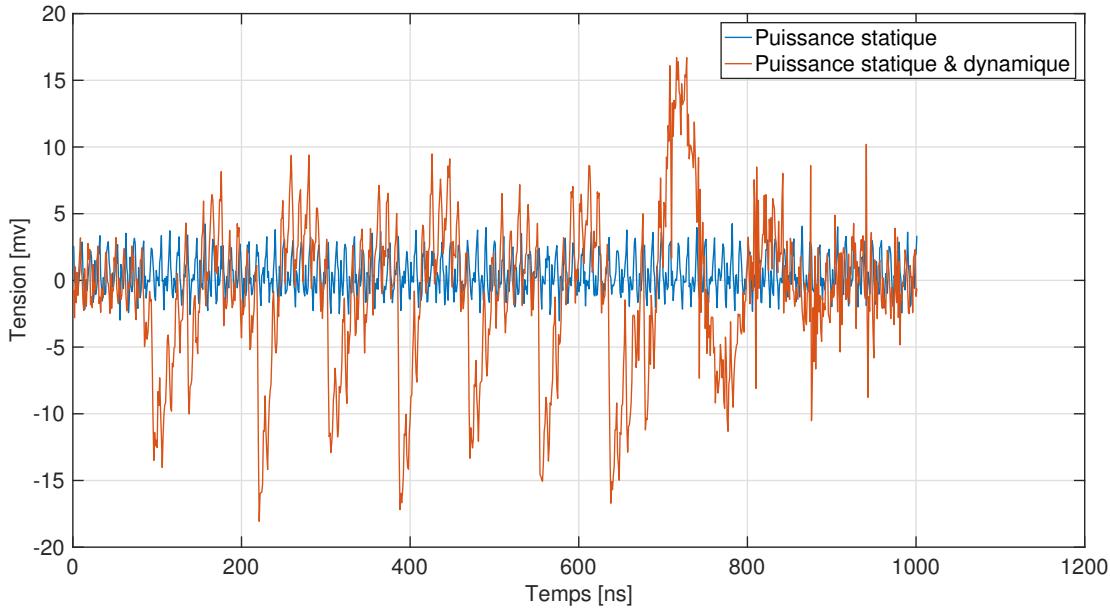


Figure 3.2 : Comparaison de la puissance statique et dynamique.

3.1.2 Composition des traces de puissance

Les attaques basées sur l'analyse de la consommation de puissance exploitent le fait que la consommation de puissance d'un device cryptographique dépend des **opérations qu'il exécute** et des **données qu'il manipule**. Ces deux informations vont ainsi permettre de définir différentes propriétés intéressantes. Pour chaque point analysé dans une trace de puissance, on notera :

- P_{op} la composante dépendante de l'opération exécutée ;
- P_{data} la composante dépendante de la donnée manipulée.

De plus, une troisième composante doit également être prise en compte. Cette composante fait référence au **bruit électrique** (aussi appelé bruit de fond) et sera notée P_{noise} . En effet, un signal est toujours affecté de petites fluctuations plus ou moins importantes. Ces fluctuations, dont les origines peuvent être diverses, sont appelées "bruit électrique" (ou simplement bruit). Le bruit est considéré comme un élément parasite aléatoire, c'est-à-dire qu'on ne sait pas le déterminer à l'avance. Au plus cette composante est élevée et au plus l'analyse de la consommation de puissance est difficile.

Ainsi, chaque point d'une trace de puissance peut être modélisé comme la somme des 3 composantes définies ci-dessus, soit : $P_{total} = P_{op} + P_{data} + P_{noise}$. Par ailleurs, en reprenant l'équation (3.5) définie dans la section 3.1.1, nous pouvons conclure que (équation 3.6) :

$$P_{total} = P_{op} + P_{data} + P_{noise} = P_{stat} + P_{dyn} \cong P_{dyn} = P_{chrg} + P_{cc} \quad (3.6)$$

En conclusion, la puissance consommée par un device cryptographique reflète directement ses activités internes. En effet, on remarque que cette puissance dépend aussi bien des opérations exécutées que des données manipulées, auxquelles il faut également ajouter la présence d'un bruit parasite inconnu (aléatoire).

3.2 Modèles de puissance

Dans une attaque par analyse de la consommation de puissance, l'attaquant doit utiliser ce que l'on appelle un **modèle de puissance** afin de prédire la consommation de puissance du device cryptographique attaqué. Différents modèles de prédictions existent. Chacun de ces modèles se base sur les valeurs de bits dans des *sets* de données. La qualité du modèle employé a un impact important sur l'efficacité de l'attaque. Deux modèles sont généralement définis et utilisés : Il s'agit des modèles de **Poids de Hamming** (*Hamming Weight - HW*) et de **Distance de Hamming** (*Hamming Distance - HD*). Il faut noter que ces deux modèles restent généraux, c'est-à-dire qu'ils ne requièrent pratiquement aucune connaissance à propos du design du circuit et sont par conséquent parfois imprécis. Une fois les prédictions obtenues, celles-ci sont comparées selon diverses méthodes (voir chapitre 4) aux mesures réelles de consommation de puissance du device capturées à l'oscilloscope (traces de puissance).

3.2.1 Modèle de poids de Hamming

Le poids de Hamming est le modèle de consommation de puissance le plus élémentaire. C'est celui le plus utilisé par un attaquant lorsqu'il s'agit d'estimer la consommation de puissance d'un circuit dont on ne connaît pas certaines valeurs intermédiaires consécutives calculées durant l'exécution de l'algorithme. Ce modèle considère qu'un 0 ne mène à aucune quantité significative de consommation de puissance tandis qu'un 1 implique une quantité significative de puissance consommée. Ainsi, pour ce modèle, on assume que la consommation de puissance prédictive est proportionnelle au nombre de bits à 1 d'une donnée traitée. Dit vulgairement, le poids de Hamming calcule le nombre de bits à 1 présents dans un nombre binaire. Mathématiquement, cela se traduit par l'expression 3.7 :

$$HW(b_0) = \sum_{i=0}^{N-1} b_{0,i} \quad (3.7)$$

Où b_0 est un mot de N bits. Par conséquent, $b_{0,i}$ est le i^{me} bit du mot binaire b_0 .
Exemple : $HW(100110) = 3$.

3.2.2 Modèle de distance de Hamming

La distance de Hamming est un modèle de consommation de puissance proposé par Brier et Al. Il est basé sur la relation entre la consommation de puissance et l'activité de commutation dans les circuits en technologie CMOS. En effet, comme indiqué en conclusion de la section 3.1.1, la consommation de puissance d'un device en technologie CMOS est principalement d'ordre dynamique, c'est-à-dire due aux activités de commutation des cellules dans le circuit. Ainsi, ce modèle assume que la puissance totale consommée par un circuit CMOS (définie selon l'équation 3.5) est équivalente à la consommation de puissance lors de commutations (transitions $0 \rightarrow 1$ et $1 \rightarrow 0$). Ce modèle est donc proportionnel au nombre de transitions $0 \rightarrow 1$ et $1 \rightarrow 0$. La distance de Hamming entre 2 nombres binaires, b_1 et b_2 se calcule en comptant le nombre de transitions ($0 \rightarrow 1$ et $1 \rightarrow 0$) entre ces 2 nombres, soit l'expression 3.8 :

$$HD(b_1, b_2) = HW(b_1 \oplus b_2) \quad (3.8)$$

Exemple : $HD(110110, 100100) = HW(110110 \oplus 100100) = HW(010010) = 2$.

Chapitre 4

Attaques par canaux auxiliaires

Ce quatrième chapitre est sans conteste celui le plus attendu. Tout d'abord, une introduction générale est donnée afin d'avoir un aperçu des différentes possibilités d'attaques par canaux auxiliaires existantes. Ensuite, l'attention est plus particulièrement portée sur trois types d'attaques par analyse de la consommation de puissance, à savoir : les attaques par analyse simple de la consommation, les attaques par analyse différentielle de la consommation et les attaques par analyse de la consommation par corrélation. L'attaque mise en oeuvre dans ce travail est celle d'analyse de la consommation par corrélation. Ultérieurement, le but sera ainsi d'attaquer l'implémentation de l'algorithme AES sur un FPGA. Les résultats de cette attaque seront présentés au chapitre 7 (Partie 2).

4.1 Introduction

À la fin des années 1990, une nouvelle contrainte pour la conception de systèmes informatiques a vu le jour : la sécurité matérielle. Bien souvent, la sécurité d'un système informatique s'appuie plus sur les concepts *software* que *hardware*. Cependant, un nouveau mode d'attaque s'est développé. Il s'agit d'attaques physiques, c'est-à-dire d'attaques exploitant différents types d'information physique (consommation de puissance, rayonnement électromagnétique, temps de calcul, ...). Ces attaques posent un véritable problème à la théorie cryptographique. En effet, alors que les algorithmes de chiffrement utilisés dans des systèmes sécurisés sont habituellement considérés comme des éléments de confiance, la *mise en œuvre* (implémentation) de ces algorithmes permet des attaques dévastatrices. En général, le but d'une attaque par canaux auxiliaires est de retrouver la clé de chiffrement utilisée par l'algorithme afin de déchiffrer des données sensibles. On distingue deux grandes familles d'attaques :

- **Attaques invasives** : Une attaque est dite invasive lorsque l'environnement interne du device cryptographique est manipulé et observé par l'attaquant. Accéder à l'environnement interne signifie accéder au *silicium*. Ainsi, ce type d'attaque a généralement pour conséquence d'endommager voir de détruire entièrement le device cryptographique. La violation d'informations se fait donc au détriment du device. On distingue deux types d'attaques invasives :
 - Les attaques invasives **irréversibles** qui conduisent à la destruction totale du device cryptographique. Ce type d'attaque est souvent réalisé pour connaître la conception physique d'un device. *Exemple* : Découpage laser d'un circuit intégré.
 - Les attaques invasives **pseudo-réversibles** qui n'entraînent pas forcément la destruction totale du device cryptographique, mais qui sont souvent tout de même invasives puisqu'elles nécessitent la préparation du circuit (découpe partielle du boîtier du circuit intégré par exemple). Un exemple typique de ce type d'attaque est ce qu'on appelle les *attaques en fautes*. Le principe consiste à manipuler les conditions environnementales du système (tension, température, lumière, *clock*, etc.) pour générer volontairement des fautes. Les fautes ainsi créées peuvent entraîner le circuit dans des modes de fonctionnement conduisant à des erreurs et ces erreurs peuvent ensuite être exploitées pour déterminer la clé.

- **Attaques non invasives** : Une attaque est dite non invasive lorsqu'elle ne requiert pas que le device cryptographique soit ouvert. Cette attaquant exploite ainsi l'analyse, en fonctionnement normal, d'informations s'échappant d'un device cryptographique. Cela peut être l'analyse de la consommation de puissance, l'analyse temporelle, l'analyse par rayonnement électromagnétique, etc... C'est ce type d'attaque qui sera étudié pour la réalisation de ce travail.

La figure 4.1 ci-dessous présente les 3 ensembles d'attaques définis précédemment et donne pour chacun d'entre eux quelques exemples d'applications.



Figure 4.1 : Classification des attaques physiques.

Les attaques non invasives sont globalement beaucoup plus simples à mettre en oeuvre que les attaques invasives. En effet, du fait qu'il faille ouvrir le device cryptographique, les attaques invasives requièrent une méthodologie et une infrastructure relativement plus compliquée et plus chère que les attaques non invasives qui ne nécessitent pas de connaissances préalables du système interne et qui sont donc plus facilement déployées.

Les attaques non invasives consistent donc à analyser des données issues de canaux auxiliaires au device cryptographique lorsque ce dernier est en cours de chiffrement. Ces canaux auxiliaires sont des canaux présents physiquement sur le circuit attaqué et le long desquels de l'information s'échappe. Cette information peut se présenter sous différentes formes : rayonnement électromagnétique, rayonnement photonique, consommation de puissance, etc. C'est donc pour cette raison qu'on parle de *side-channel attacks* ou en français *l'attaque par canal auxiliaire*. En effet, les fonctions cryptographiques, bien que pouvant être extrêmement robustes théoriquement (c'est-à-dire mathématiquement) sont très sensibles aux fuites d'informations. C'est-à-dire qu'une quantité très faible d'informations peut être exploitée pour retrouver la clé d'un algorithme cryptographique très fort. On insistera à nouveau sur le fait que l'attaque exploite les faiblesses physiques du système cryptographique, c'est-à-dire du système implémentant l'algorithme de chiffrement (l'AES en l'occurrence). En aucun cas, l'attaque exploite les failles des principes mathématiques mis en place.

La figure 4.2 ci-dessous présente différentes possibilités **d'attaques non invasives** sur un device cryptographique.



Figure 4.2 : Différentes façons d'opérer une attaque non invasive sur un device cryptographique.

Dans la suite de ce travail, on se concentrera sur un type précis d'attaque non invasive : l'attaque par **l'analyse de la consommation de puissance**. Les trois sous-sections de la section suivante (4.2) décrivent chacune un principe de fonctionnement différent d'attaque pour ce type d'analyse (consommation de puissance). L'attaque qu'il m'a été demandé de réaliser pour ce travail porte sur **l'analyse de la consommation par corrélation**. Pour cette raison, les deux autres possibilités que sont l'analyse simple de la consommation et l'analyse différentielle de la consommation ne seront que brièvement parcourues. Toutefois, ces deux attaques sont apparues avant celle par corrélation et il convient donc de les citer afin de comprendre l'ordre chronologique d'apparition et d'amélioration des attaques par analyse de la consommation.

4.2 Analyse de la consommation de puissance

Une attaque par analyse de la consommation de puissance a pour objectif d'analyser la consommation de puissance d'un device cryptographique en cours de chiffrement afin de retrouver la clé de l'algorithme implémenté sur ce device.

4.2.1 Analyse simple de la consommation

Une analyse simple de la consommation (SPA en anglais pour *Simple Power Analysis*) est une technique qui interprète directement la ou les mesure(s) de consommation capturée(s) sur un device cryptographique en cours de chiffrement. En d'autres mots, l'attaquant enregistre une trace de puissance et tente d'identifier sur celle-ci certaines opérations exécutées ou certaines données manipulées. En général, cela exige des connaissances avancées sur l'implémentation de l'algorithme cryptographique présent sur le device. Pour cette raison, ce type d'attaque est assez compliqué à réaliser en pratique même s'il n'exige qu'un nombre très limité de traces. Ainsi, ce type d'attaque permet en général d'identifier un algorithme cryptographique et dans des cas extrêmes et sous certaines conditions, il est même possible de lire la clé de chiffrement utilisée par l'algorithme.

4.2.2 Analyse différentielle de la consommation

Une attaque par analyse différentielle de la consommation (DPA en anglais pour *Differential Power Analysis*) est une attaque beaucoup plus populaire que l'attaque SPA. Ceci est dû au fait que l'attaque DPA ne requiert aucune connaissance approfondie sur le device cryptographique attaqué. De plus, cette attaque permet de retrouver la clé de chiffrement même si les traces capturées sont fortement bruitées. Par contre, en comparaison des attaques SPA qui tirent l'information d'un nombre restreint de traces, les attaques DPA nécessitent un plus grand nombre de traces et ce, afin de trouver des différences de consommation relatives à un bit. En effet, l'attaquant cherche à récupérer la clé de chiffrement en utilisant des informations relatives à un bit et en se basant sur un sous-ensemble de la clé et du texte (chiffré ou clair). Plus précisément, voici les quatre étapes utiles pour mettre en application une attaque DPA :

1. **Mesurer la consommation de puissance** : l'attaquant doit mesurer et enregistrer des traces de puissance capturées sur le device cryptographique lorsque celui-ci est en cours de chiffrement. **Une trace** représente l'ensemble des opérations exécutées par un algorithme pour **un texte clair envoyé**. Étant donné qu'il faut un grand nombre (N) de traces pour que l'attaque DPA réussisse, il faudra par conséquent envoyer un grand nombre (N) de textes clairs.
2. **Modèle logiciel** : un modèle logiciel reproduit les opérations de l'algorithme implémenté et donne un résultat sur 1 bit pour chaque ensemble de clé testé. Prenons l'exemple de l'algorithme AES-128 pour mieux comprendre. Pour rappel, la clé utilisée par l'AES-128 a une taille de 128 bits. Si l'attaquant tente de retrouver un seul octet de la clé, c'est-à-dire 8 bits, il existe au total 2^8 soit 256 valeurs possibles de clé. L'objectif de l'attaquant consistera à utiliser un modèle logiciel simulant l'algorithme AES-128 et testant les 256 clés possibles. L'attaque DPA fonctionne plus rapidement (moins de traces nécessaires) à la sortie de l'opération *SubBytes* (par rapport aux trois autres opérations). Pour cette raison, le modèle logiciel reproduit les deux premières étapes de l'AES et donne un résultat sur 1 bit pour chacune des 256 sous-clés possibles. Ce résultat est un des bits en sortie de l'étape *SubBytes*, le premier est couramment utilisé.
3. **Classement** : nous avons d'une part un ensemble de N traces (1000) et d'autre part un ensemble de bits de sélection (256, soit pour 1 byte de la clé). Le but de l'attaquant est de classer les traces mesurées en deux groupes G_0 et G_1 . La prise de décision se fait de la manière suivante : pour un texte clair T_i avec $i \in (0 ; 1000)$, si le bit de sélection vaut 0 pour une clé K_j avec $j \in (0 ; 256)$, la trace de puissance TP_i avec $i \in (0 ; 1000)$ correspondant à ce texte clair T_i est mémorisée dans un groupe G_0 . À l'inverse, si le bit vaut 1, elle est mémorisée dans un groupe G_1 .
4. **Mesurer la courbe DPA** : la différence de la moyenne temporelle des traces du groupe G_1 avec la moyenne temporelle des traces du groupe G_0 pour la sous-clé K_j donne la courbe DPA associée à cette sous-clé. Il existe donc 256 courbes DPA (dans le cas où on souhaite retrouver un byte de la clé). Parmi celles-ci, la courbe dont la valeur moyenne est la plus élevée correspond à la clé de chiffrement.

La figure 4.3 ci-dessous présente les quatre étapes utiles pour mettre en oeuvre une attaque DPA.

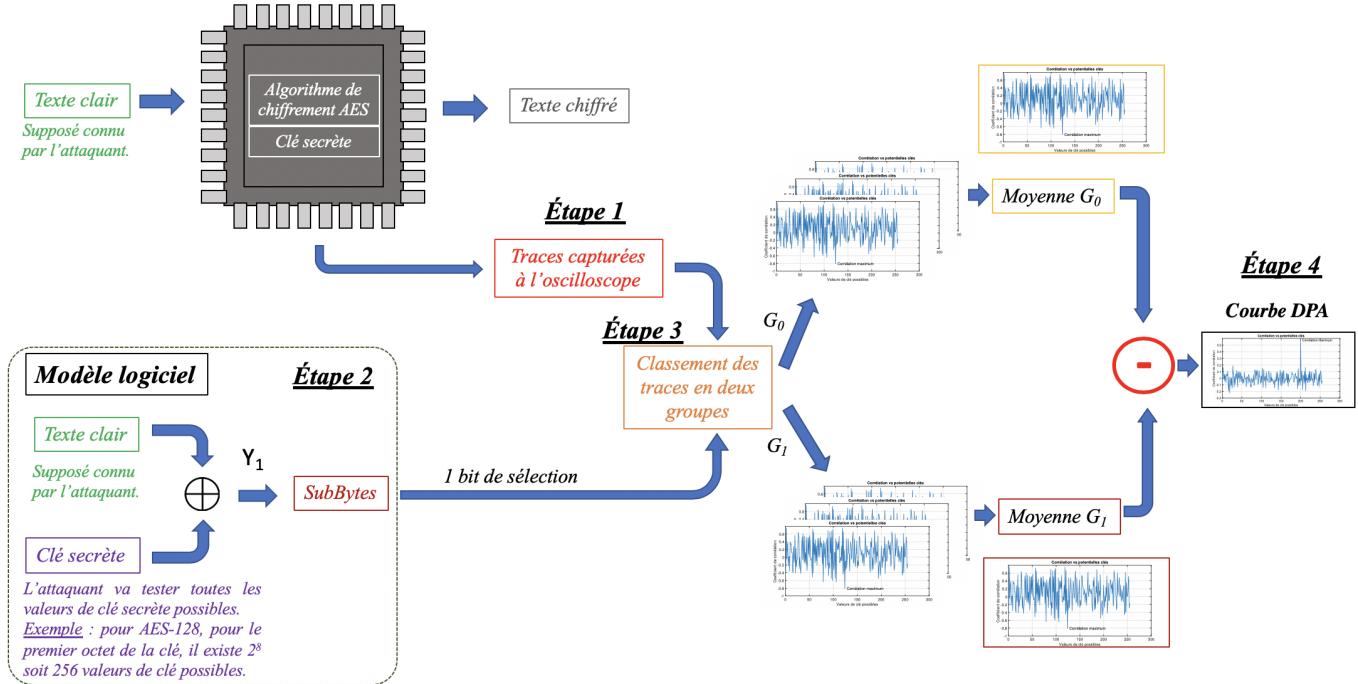


Figure 4.3 : Principe de fonctionnement d'une attaque DPA.

4.2.3 Analyse de la consommation par corrélation

L'attaque par analyse de la consommation par corrélation (CPA en anglais pour *Correlation Power Analysis*) est une attaque qui dérive de l'attaque DPA. En réalité, il s'agit d'une amélioration de l'attaque DPA. En effet, en 2004, de nouvelles recherches eurent pour objectif de directement calculer la corrélation entre la consommation de puissance du device cryptographique (les traces) et un modèle de prédiction de puissance (décris à la section 3.2). La faille utilisée est la corrélation, dans les circuits électroniques de technologie CMOS, entre la consommation de puissance dynamique et le nombre de transistors qui commutent (de $0 \rightarrow 1$ et de $1 \rightarrow 0$) (voir chapitre 3). Pour bien comprendre le fonctionnement de cette attaque, il convient donc de définir le terme *corrélation* et plus précisément le terme *coefficient de corrélation*.

Définition 4.2.1. Coefficient de corrélation : Par définition, le coefficient de corrélation est un coefficient statistique permettant de mettre en évidence une liaison entre deux types de séries de données statistiques. La valeur du coefficient de corrélation est toujours comprise entre -1 et 1. Cette valeur se calcule de la façon suivante (4.1) :

$$r(X; Y) = \frac{\sum(X - \bar{X}).(Y - \bar{Y})}{\sqrt{\sum(X - \bar{X})^2} \cdot \sqrt{\sum(Y - \bar{Y})^2}} \quad (4.1)$$

Où

- X et Y sont deux séries de données statistiques.
- \bar{X} et \bar{Y} sont les moyennes respectives des variables X et Y .

Sur base de la valeur du coefficient de corrélation, on peut conclure que :

- Si la valeur absolue du coefficient de corrélation est élevée (proche de 1), il y a une forte liaison entre les deux séries analysées.
- Si la valeur absolue du coefficient de corrélation est faible (proche de 0), il y a une très faible liaison, voire aucun lien entre les deux séries analysées.

Pour réaliser une attaque CPA, l'attaquant a principalement besoin de deux outils :

- **Un oscilloscope** : L'oscilloscope est utilisé pour enregistrer des traces de puissance sur le device attaqué. C'est-à-dire des mesures de tension en fonction du temps représentant la consommation de puissance du device cryptographique lorsque celui-ci chiffre un texte clair. N textes clairs mènent à N traces.
- **Un ordinateur** : L'ordinateur fournit la puissance de calculs nécessaire à la réalisation de l'attaque. Plus précisément, il réalise trois grands types de calculs : calculs d'opérations de l'algorithme de chiffrement (l'AES en l'occurrence), calculs de prédiction de puissance (selon le modèle de puissance établi) et calculs des coefficients de corrélation (sur base des traces de puissance).

Comme précisé ci-dessus, l'oscilloscope est utilisé afin de capturer et d'enregistrer des données, appelées *traces*, mesurées à partir des canaux auxiliaires du circuit électrique (dans notre cas, un FPGA). Pour réaliser la mesure à l'oscilloscope, une résistance est placée en série avec le canal (la PIN) connecté à la tension d'alimentation V_{DD} du device cryptographique. L'oscilloscope est alors en mesure d'enregistrer une différence de potentiel $V(t)$ aux bornes de la résistance. Étant donné que le courant $I_{DD}(t)$ circulant dans le device cryptographique est de très faible valeur (μA), la tension $V(t)$ est également très faible. Un amplificateur est généralement utilisé afin d'amplifier cette différence de potentiel. Comme il sera précisé au chapitre 6, le circuit de test employé (*SAKURA-G*) utilise également un amplificateur pour mesurer une tension raisonnable. La figure 4.4 ci-dessous présente le principe de mesure à l'oscilloscope.

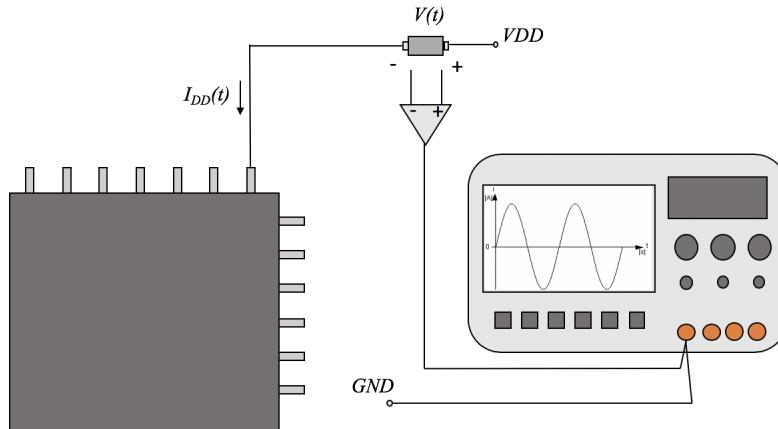


Figure 4.4 : Principe de mesure à l'oscilloscope.

On parle d'attaque par analyse de la consommation de puissance or avec un oscilloscope, on mesure une tension et non une puissance. Cependant, comme le démontre l'équation 4.3, la puissance consommée $p(t)$ est proportionnelle à la tension consommée $V(t)$. Il ne s'agit donc pas d'une erreur de parler de consommation de puissance. En effet, en supposant que la tension d'alimentation V_{DD} est constante et par simple application de la loi d'Ohm, on a :

$$\begin{cases} u(t) = V_{DD} \\ i(t) = \frac{V(t)}{R} \end{cases} \quad (4.2)$$

En reprenant la définition de la puissance consommée et en y remplaçant les termes $u(t)$ et $i(t)$, on a :

$$p(t) = u(t).i(t) = V_{DD} \cdot \frac{V(t)}{R} \quad (4.3)$$

Quatre étapes (similaires à l'attaque DPA) sont nécessaires à l'attaquant pour mettre en place une attaque CPA :

1. **Mesurer la consommation de puissance** : l'attaquant doit mesurer et enregistrer des traces de puissance capturées sur le device cryptographique lorsque celui-ci est en cours de chiffrement. Une trace représente l'ensemble des opérations exécutées par un algorithme pour un texte clair envoyé. Étant donné qu'il faut un grand nombre (N) de traces pour que l'attaque CPA réussisse, il faudra par conséquent envoyer un grand nombre (N) de textes clairs.
2. **Calculer les valeurs intermédiaires hypothétiques** : comme nous l'avons vu au chapitre 2 section 2.2, lorsque l'algorithme AES chiffre des données, des valeurs intermédiaires sont calculées en fonction des opérations exécutées et des données manipulées. Prenons l'exemple de l'algorithme AES-128. La clé utilisée a une taille de 128 bits. Si on tente de retrouver un seul byte de la clé, c'est-à-dire 8 bits, il existe au total 2^8 soit 256 valeurs de clé possibles. L'objectif de l'attaquant va alors être de simuler l'algorithme AES-128 en testant les 256 clés possibles. Comme pour l'attaque DPA, l'attaque CPA fonctionne plus rapidement (moins de traces nécessaires) à la sortie de l'opération *SubBytes* (par rapport aux trois autres opérations). Pour cette raison, la simulation sur ordinateur reproduit les deux premières étapes de l'AES et donne un résultat sur 8 bits pour chacune des 256 sous-clés possibles. Nous noterons qu'il est envisageable de réaliser l'attaque en sortie d'une des trois autres opérations de l'AES (*AddRoundKey*, *ShiftRows* ou *MixColumns*). Cependant cela nécessiterait d'envoyer plus de messages clairs, d'enregistrer plus de traces et par conséquent, cela prendrait plus de temps. Il s'agirait donc d'une attaque non optimisée.
3. **Utiliser un modèle de puissance** : Une fois que les valeurs intermédiaires hypothétiques sont calculées, l'attaquant doit utiliser un modèle de puissance pour faire correspondre à ces valeurs intermédiaires hypothétiques des valeurs de consommation de puissance simulées. Les modèles de puissance généralement utilisés sont le poids de Hamming ou la distance de Hamming.
4. **Mesurer la similarité** : Enfin, la dernière étape consiste à mesurer la similarité entre les vraies traces de puissance et les traces de puissance simulées. Pour ce faire, l'attaquant utilise le coefficient de corrélation. En effet, pour chaque texte clair envoyé, 256 valeurs de clés sont testées (si on ne s'intéresse qu'à un octet de la clé) conduisant ainsi à 256 valeurs simulées de consommation de puissance. Dans ce cas, si N textes clairs sont envoyés, $N * 256$ valeurs simulées de puissance sont calculées. Ces $N * 256$ valeurs simulées de puissance sont alors corrélées selon l'équation 4.1 avec les N traces de puissance capturées à l'oscilloscope. Le résultat le plus élevé (en valeur absolue) de cette corrélation permettra de retrouver la valeur du byte de la clé (voir figures 4.6 et 4.7 plus loin). En effet, comme décrit précédemment, le coefficient de corrélation met en évidence une liaison entre deux types de séries de données. Dans ce cas-ci, les deux types de séries de données sont d'une part les traces de puissance et d'autre part les valeurs simulées de puissance. Ainsi, si une forte valeur de corrélation est obtenue pour une clé testée, cela signifie qu'il y a une forte liaison entre les deux séries. À l'inverse, si une valeur de corrélation est faible alors cela signifie qu'il n'y a pas de liaison entre les deux séries de données.

Remarque : Pour la réalisation de l'attaque CPA, nous avons posé l'hypothèse que l'attaquant connaît les textes clairs envoyés au device cryptographique. Ceci est envisageable à condition que ce dernier soit en possession du device cryptographique. Il est aussi possible de réaliser une attaque CPA à partir des textes chiffrés cependant, cette méthode ne sera pas abordée dans ce travail.

La figure 4.5 présente le principe de fonctionnement d'une attaque CPA selon les quatre étapes définies précédemment.

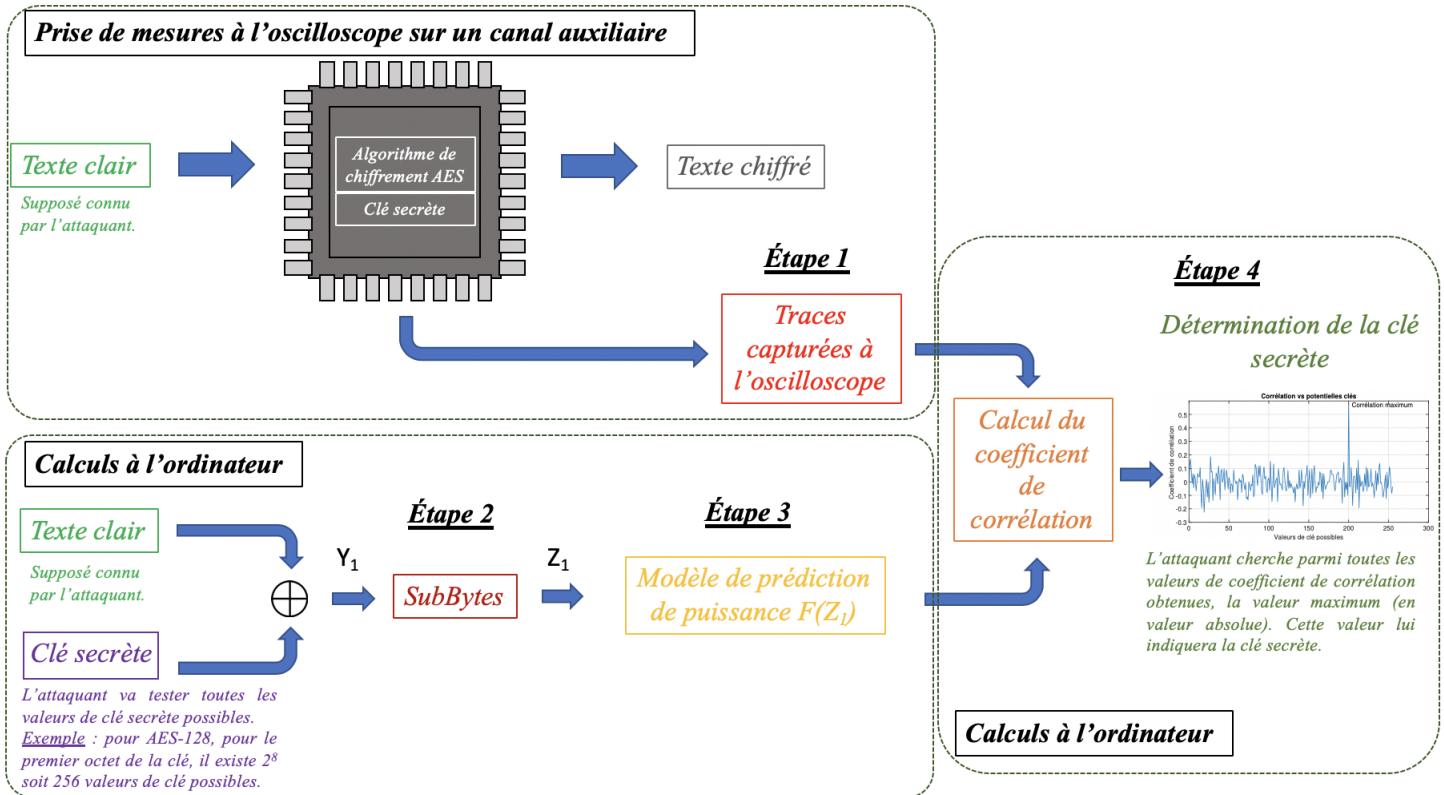


Figure 4.5 : Principe de fonctionnement d'une attaque CPA.

Les figures 4.6 et 4.7 présentent de manière générale l'évolution du coefficient de corrélation pour le huitième byte d'une clé de chiffrement. En effet, dans cet exemple, nous considérons que l'attaquant a pour tâche de retrouver le huitième byte d'une clé utilisée par l'AES-256 (pour rappel, une clé utilisée avec l'AES-256 possède une taille totale de trente-deux bytes). La première figure (4.6) représente l'évolution du coefficient de corrélation au cours du temps pour quatre valeurs de clés différentes. On remarque un pic de corrélation plus élevé dans le graphe dédié à la clé 40. Il s'agit en effet de la vraie valeur du huitième byte de la clé employée par l'algorithme AES pour cet exemple. Pour encore mieux comprendre le principe de corrélation, la figure 4.7 présente la corrélation obtenue en fonction des 256 valeurs de clés testées (toujours pour le huitième byte de la clé). Sur ce graphe encore, on constate qu'il y a un pic de corrélation. Cette valeur maximale du coefficient de corrélation correspond à la clé 40. Ainsi, si l'attaquant souhaite retrouver les trente-deux bytes de la clé employée pour le chiffrement, il lui suffit de retrouver les trente-deux valeurs de clé correspondantes aux trente-deux valeurs maximales du coefficient de corrélation. Les deux figures (4.6) et (4.7) sont présentées à la page suivante.

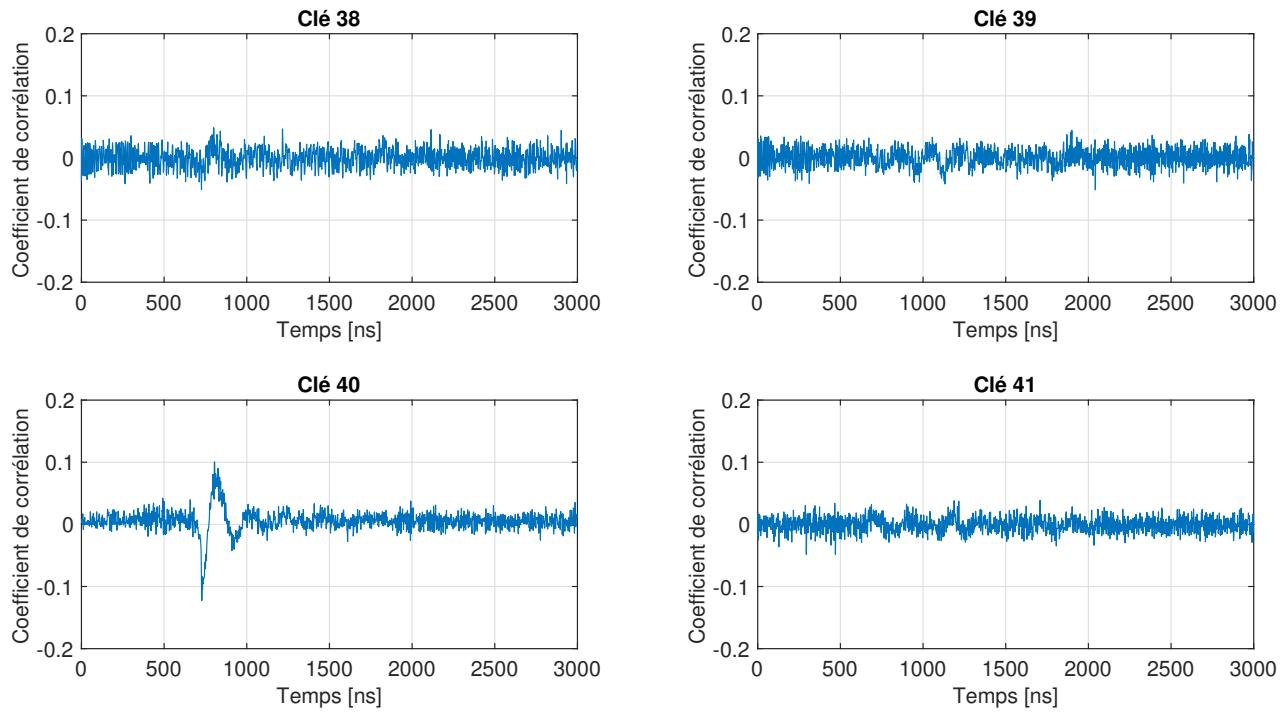


Figure 4.6 : Différents résultats de corrélation en fonction de 4 clés testées.

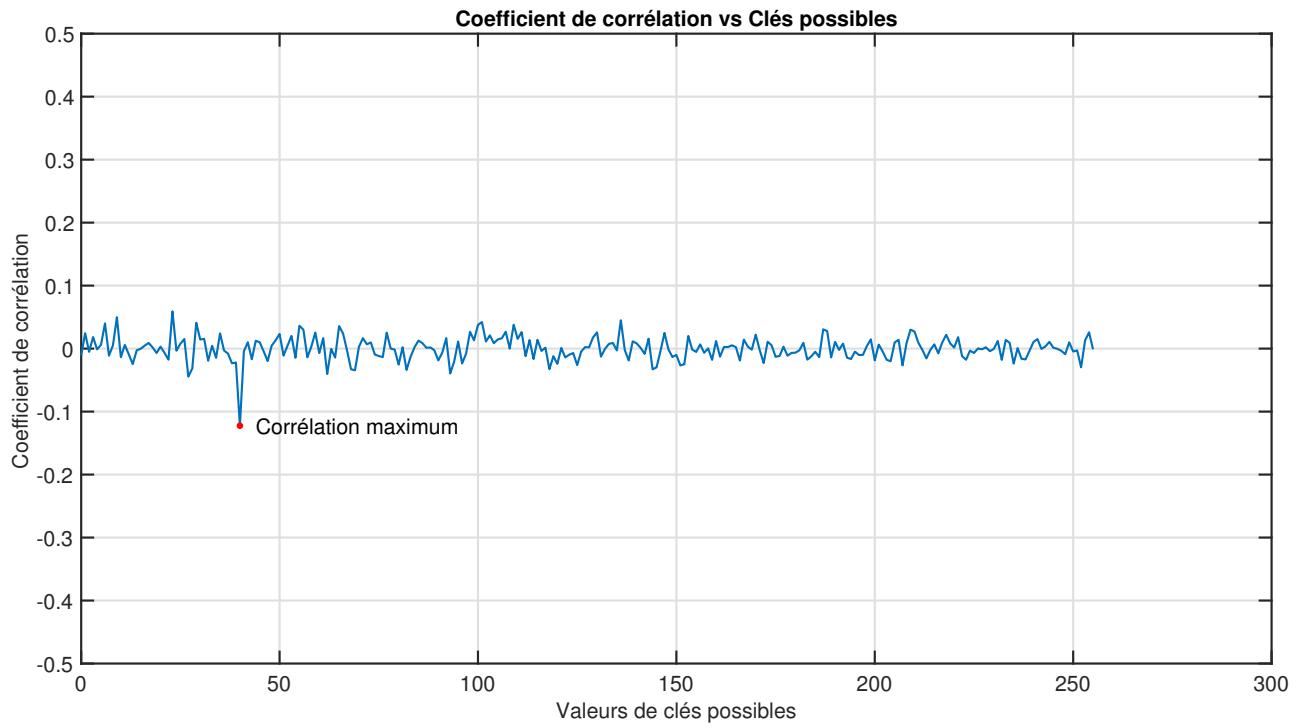


Figure 4.7 : Corrélation en fonction des 256 clés testées.

Chapitre 5

Contre-mesures

Ce cinquième et dernier chapitre de la partie *théorie* introduit les différents types de contre-mesures existantes permettant de neutraliser les attaques par analyse de la consommation de puissance. Tout d'abord, une introduction servant de préliminaires est présentée. Ensuite, deux grands types de contre-mesures sont détaillés. Il s'agit des contre-mesures de type *masking* et de type *hiding*. Enfin, une contre-mesure particulière est définie. Il s'agit de la contre-mesure de type *faking*, développée pour la réalisation de ce travail.

5.1 Introduction

Dès que les attaques par analyse de la consommation de puissance ont été reconnues fonctionnelles, une série de contre-mesures fut développée afin d'empêcher la réussite de celles-ci. Les attaques CPA, au même titre que toute autre attaque utilisant l'analyse de la consommation de puissance (attaque SPA, attaque DPA, etc.), fonctionnent du fait que la consommation de puissance du device cryptographique dépend des valeurs intermédiaires exécutées par l'algorithme de chiffrement. Ainsi, l'objectif d'une contre-mesure est de casser cette relation qu'il existe entre la consommation de puissance et les données sensibles manipulées et entre la consommation de puissance et les opérations exécutées. On distingue deux grandes catégories de contre-mesures :

1. **Les contre-mesures de type *Hiding*** : Le principe des contre-mesures de type Hiding est de rendre la consommation de puissance du device cryptographique indépendante des opérations exécutées et des données manipulées. Pour ce faire, il existe deux approches différentes possibles. Ces deux approches sont détaillées à la section 5.2.
2. **Les contre-mesures de type *Masking*** : Le principe des contre-mesures de type Masking est de générer des valeurs intermédiaires aléatoires. Ainsi, on accepte que la consommation de puissance du device cryptographique dépende des données manipulées. Cependant, on modifie (on masque) volontairement ces valeurs intermédiaires afin de fausser les traces de puissance obtenues à l'oscilloscope.

Il est donc important de bien comprendre que les contre-mesures développées pour les attaques par analyse de la consommation de puissance ont pour objectif général de modifier l'allure des traces de puissances afin de compliquer la tâche de l'attaquant. Il existe deux façons de modifier une trace de puissance :

- En agissant sur *l'amplitude* de la trace, on parlera *d'intensité du leakage*.
- En agissant sur *la position dans le temps* de la trace, on parlera *d'instant du leakage*.

Les figures 5.1 et 5.2 respectivement montrent comment modifier des traces de puissances en agissant d'une part sur l'amplitude (figure 5.1) et d'autre part sur la position dans le temps (figure 5.2). Pour ces deux figures, nous affichons la consommation de puissance d'un FPGA en cours de chiffrement sur lequel est implémenté l'algorithme AES-256. Concernant la figure 5.1, le graphe de gauche affiche la consommation normale tandis que le graphe de droite affiche la consommation modifiée en amplitude. En effet, nous avons fait en sorte que la consommation de puissance soit plus uniforme en amplitude. Nous sommes passés d'une consommation qui variait entre $[-18,1 \text{ V}; 9,7 \text{ V}]$ à une consommation qui varie entre $[-13,1 \text{ V}; 4,9 \text{ V}]$. Concernant la figure 5.2, nous avons inversé l'ordre d'exécution des opérations. Dans cet exemple, nous avons imaginés deux opérations (ce n'est pas le cas en réalité), représentées par les cercles rouge et vert. Le graphe de gauche affiche la consommation normale tandis que le graphe de droite affiche la consommation *retournée* dans le temps. En effet, certaines opérations sont parfois mélangées dans un algorithme cryptographique afin de nuire à l'attaque. Pour conclure, on se rend bien compte que la modification de ces traces de puissance, que ce soit en amplitude ou dans le temps, complexifie la tâche de l'attaquant étant donné qu'il ne possède plus les mêmes repères.



Figure 5.1 : Modification de l'amplitude d'une trace de puissance.



Figure 5.2 : Modification de la position dans le temps d'une trace de puissance.

Avant de détailler chacune des deux catégories de contre-mesures, rappelons la définition de rapport signal à bruit (SNR : *Signal to Noise Ratio* en anglais). Sa définition est reprise ci-dessous.

Définition 5.1.1. Rapport Signal à Bruit (SNR) : *Le rapport signal à bruit ou SNR est un indicateur de performance. Son objectif est de mesurer la qualité de la transmission d'une information. Sa formulation mathématique est reprise à l'équation (5.1). Elle peut également être exprimée en dB (5.2) :*

$$SNR = \frac{P_{signal}}{P_{noise}} \quad (5.1)$$

$$SNR_{dB} = 10 \cdot \log_{10} \frac{P_{signal}}{P_{noise}} \quad (5.2)$$

Où :

- SNR (SNR_{dB}) représente l'indicateur de performance de la transmission de l'information sans unité (en dB).
- P_{signal} représente la puissance du signal (en Watt).
- P_{noise} représente la puissance du bruit (en Watt).

Ainsi, le calcul du SNR permet d'évaluer si le signal de transmission que l'on étudie est fortement bruité ou non. Selon l'équation (5.1), cet indicateur SNR est d'autant plus élevé que la puissance du signal est élevée ou est d'autant plus élevé que la puissance du bruit est faible. Dans une transmission idéale, on désirera toujours un SNR très grand, signifiant que la trace obtenue représente majoritairement le signal et minoritairement le bruit (qui pour rappel est un élément parasite aléatoire).

Pour une trace de puissance, si le SNR d'une opération est élevé, cela signifie que la puissance du signal est plus élevée que la puissance du bruit, c'est-à-dire qu'il est plus facile de détecter des fuites d'information. Idéalement, il faut donc que le SNR soit proche de 0. De cette façon, le bruit recouvre tellement le signal qu'il est impossible pour l'attaquant de détecter du *leakage* (fuite d'information). En pratique, cela peut être réalisé en diminuant la variance du signal vers 0 ou en augmentant la variance du bruit vers l'infini.

- Pour réduire la variance du signal, la consommation de puissance a besoin d'être exactement égale pour toutes les opérations exécutées et données manipulées. En pratique, cela se traduira par de petites valeurs de variances pour le signal.
- Pour augmenter la variance du bruit, l'amplitude du bruit a besoin d'être augmentée de façon significative laissant croire à l'attaquant l'existence de commutations sur les cellules du device. En pratique, cela se traduira par de grandes valeurs de variances pour le bruit.

La figure 5.3 ci-dessous présente une fonction sinusoïdale sans bruit (gauche) et avec bruit (droite).

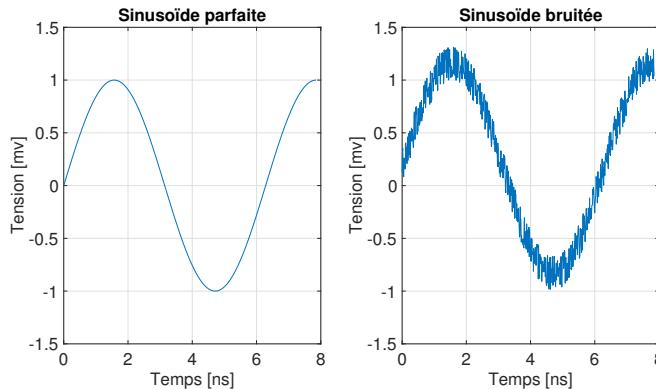


Figure 5.3 : Importance du bruit pour un signal de transmission.

5.2 Contre-mesures Hiding

Comme précisé ci-avant, le principe des contre-mesures de type Hiding est de rendre la consommation de puissance du device cryptographique indépendante des opérations exécutées et des données manipulées. Pour ce faire, deux approches sont possibles :

1. Faire en sorte que la consommation de puissance du device cryptographique soit **aléatoire**. Cela signifie qu'à chaque coup de clock, une certaine quantité aléatoire de puissance est consommée. Le but est donc de **modifier l'instant du leakage** ou bien de **modifier l'intensité du leakage** dans la trace de puissance.
 - *La modification de l'instant du leakage peut être réalisée par un désalignement des traces.* Cela va compliquer la tâche de l'attaquant. En effet, celui-ci doit, dans un premier temps, procéder à l'alignement de ses traces pour pouvoir les analyser correctement. Si celles-ci ne sont pas alignées, il ne pourra rien en tirer de concret. Ce désalignement des traces peut s'obtenir de différentes façons : utiliser des horloges de fréquences différentes, utiliser des interruptions aléatoires lors de l'exécution du programme, changer l'ordre des instructions, etc.
 - *La modification de l'intensité du leakage peut être réalisée par une modification du rapport signal à bruit.* Nous avons vu que pour diminuer le SNR, il suffisait d'augmenter le bruit ou de diminuer le signal. Dans ce cas-ci, nous allons augmenter la variance du bruit. En effet, en ajoutant du bruit de façon indépendante à l'exécution de l'algorithme, on va diminuer le SNR, ce qui va avoir pour conséquence de diminuer le leakage d'une opération : L'attaquant aura dès lors plus de difficultés à retrouver la clé secrète. On peut, pour ce faire, utiliser un filtre afin de ne laisser passer que certaines composantes de puissance ou encore utiliser ce qu'on appelle des "*noise engines*", systèmes fonctionnant en parallèle du device cryptographique et ayant pour but de générer du bruit.
2. Faire en sorte que la consommation de puissance du device cryptographique soit **identique**. Cela signifie qu'à chaque coup de clock, une quantité égale de puissance est consommée pour toutes les opérations exécutées et pour toutes les données manipulées. Le but est donc de **modifier l'intensité du leakage**. Autrement dit, on souhaite uniformiser l'amplitude de la trace. *Pour ce faire, on va à nouveau modifier le rapport signal à bruit.* L'idéal étant d'avoir un SNR proche de 0, si on n'augmente pas la variance du bruit, la deuxième approche consiste à diminuer la variance du signal. Pour ce faire, on va modifier de façon physique les cellules CMOS. En effet, le but est de faire en sorte que chaque cellule logique consomme la même consommation de puissance pour toute opération demandée.

Le tableau 5.1 ci-dessous présente les paramètres influencés pour la réalisation d'une contre-mesure selon le type d'approche suivi (puissance équivalente ou aléatoire) tandis que l'annexe B.1 reprend un ensemble de contre-mesures de type hiding. Pour rappel, des exemples de modification de l'intensité du leakage et de l'instant du leakage pour une trace de puissance ont été donnés en introduction de ce chapitre (voir figures 5.1 et 5.2).

<i>Consommation de puissance équivalente</i>	<i>Consommation de puissance aléatoire</i>
Intensité du leakage	Instant du leakage Intensité du leakage

Table 5.1 : Les contre-mesures de type hiding sont utilisées pour rendre aléatoire ou égale la consommation de puissance du device cryptographique..

Sur base de l'annexe B.1, nous pouvons expliquer le principe de fonctionnement de certains exemples de contre-mesures :

- Concernant la manipulation d'horloge, nous pouvons :
 - Ignorer certains coups de clock : ce qui aura pour effet de retarder l'exécution du prochain cycle, ce qui engendrera donc un désalignement des traces.
 - Changer aléatoirement la fréquence de clock : à l'aide d'un oscillateur et de nombres aléatoires, la fréquence d'horloge est changée à intervalles fréquents, provoquant des vitesses d'exécution différentes et par conséquent un désalignement des traces.

- Concernant les interruptions : on va générer aléatoirement certaines interruptions lors de l'exécution de l'algorithme, ce qui va *casser* (couper) les traces provoquant ainsi un désalignement.
- Concernant la manipulation d'instructions :
 - Le mélange des instructions : on va mélanger l'ordre des opérations (seules les opérations qui peuvent être déplacées ou interverties). Ainsi, pour AES par exemple, il est possible de réaliser l'opération *subBytes* sur les 16 bytes de la matrice, dans n'importe quel ordre, sans nuire au déroulement de l'algorithme.
 - Les instructions inutiles : en insérant des opérations inutiles, on va injecter des quantités d'informations inutiles dans chaque trace. Cela provoquera un désalignement des traces.
 - Le choix des instructions : toutes les instructions ne "fuitent" pas la même quantité d'information sur les opérandes. Il pourrait être bon de choisir les instructions qui révèlent le moins d'informations utiles à un attaquant ou de remplacer certaines instructions par d'autres instructions équivalentes pour que la fuite d'information ne soit pas toujours la même.
- Concernant la diminution du SNR :
 - On peut diminuer le SNR en augmentant la variance du bruit à l'aide de *noise engines*. Il s'agit de composants hardware qui travaillent en parallèle à l'exécution cryptographique. Autrement dit, ces composants hardware vont générer du bruit, ce qui va avoir pour conséquence de diminuer le SNR. En effet, la consommation mesurée sera la somme de la consommation de l'exécution de l'algorithme cryptographique et des composants hardware.
 - On peut diminuer le SNR en diminuant la variance du signal. En pratique, il existe deux possibilités pour diminuer la variance du signal :
 - Une première approche pourrait concerner la cellule logique CMOS en elle-même. On sait (section 3.1.1) que la consommation de puissance totale d'un device cryptographique est la somme des puissances consommées par chaque cellule. Ainsi, si chaque cellule consomme une puissance constante, la puissance totale est constante. Il faut donc construire des cellules qui consomment des quantités de puissance constantes.
 - Une seconde approche plus complexe concerne le filtrage. Il s'agit de filtrer la puissance consommée par le device cryptographique. Le but est alors de supprimer, via ce filtre, toutes les composantes de la trace de puissance qui dépendent des données manipulées et des opérations exécutées.

En conclusion, le développement d'une contre-mesure de type hiding se concrétise soit en modifiant le *design* du circuit électronique lors de sa conception (fabrication) soit en manipulant les coups de clocks et/ou les instructions devant être opérées par le circuit électronique. Par conséquent, ce type de contre-mesure prend du temps à être développé et est généralement assez compliqué à mettre en oeuvre.

5.3 Contre-mesures Masking

Le principe des contre-mesures de type masking est de générer des valeurs intermédiaires aléatoires. Autrement dit, lors de l'exécution d'un algorithme de chiffrement, différentes opérations sont exécutées conduisant à différentes valeurs intermédiaires calculées. Si aucune protection n'est mise en place, la consommation de puissance du device cryptographique dépendra des données intermédiaires qui sont manipulées par le device cryptographique (par l'algorithme précisément). L'attaquant peut alors potentiellement retrouver la clé secrète en enregistrant la consommation de puissance du device cryptographique et en réalisant une attaque de type CPA par exemple. Par contre, si chaque valeur intermédiaire est dissimulée sous une nouvelle valeur intermédiaire aléatoire dite **masquée**, alors l'attaquant pourra toujours analyser la consommation de puissance, les traces qu'il obtiendra fourniront des informations faussées (par le masque). En d'autres mots, ce type de contre-mesure accepte que la consommation de puissance du device cryptographique dépend des données manipulées et des opérations exécutées. Cependant, on va modifier les valeurs intermédiaires de façon aléatoire pour que les traces mesurées à l'oscilloscope n'aient plus de sens. Un avantage de cette approche est qu'elle peut être implantée au niveau de l'algorithme, c'est-à-dire sans changer les caractéristiques de consommation de puissance du device cryptographique (ce qui est plus

complexe) comme c'est le cas pour les contre-mesures de type *Hiding*. La définition 5.3 ci-dessous permet de comprendre le principe de masquage.

Définition 5.3.1. Masque : Une valeur intermédiaire masquée v_m est une valeur intermédiaire v cachée par une valeur aléatoire m . On obtient donc la relation suivante (5.3) :

$$v_m = v * m \quad (5.3)$$

Ne connaissant pas la valeur aléatoire m , l'attaquant ne peut pas retrouver la valeur intermédiaire v . Ainsi, la consommation de puissance du device cryptographique dépend des valeurs intermédiaires masquées v_m qui ne fournissent aucune information sur les valeurs intermédiaires v . Autrement dit, un masque cache les valeurs intermédiaires et il n'est donc plus possible de retrouver la clé de chiffrement. Bien évidemment, les masques ont besoin d'être supprimés à la fin des opérations algorithmiques afin d'obtenir *in fine* le *vrai* message chiffré. Autrement, le message chiffré obtenu serait également masqué et n'aurait donc aucun sens.

Le symbole $*$ dans (5.3) représente le type d'opérateur utilisé pour appliquer le masque sur les valeurs intermédiaires. Il peut s'agir d'une fonction booléenne ou d'une fonction arithmétique. Ainsi, on définit :

Définition 5.3.2. Masque booléen : l'opération $*$ est remplacée par la fonction booléenne *XOR* (\oplus). Dans ce cas, le masque devient (5.4) :

$$v_m = v \oplus m \quad (5.4)$$

Définition 5.3.3. Masque arithmétique : l'opération $*$ est remplacée par une addition modulaire ($+$) ou par une multiplication modulaire (\times). Dans ce cas, le masque devient (5.5 ou 5.6) :

$$v_m = v + m \pmod{n} \quad (5.5)$$

ou

$$v_m = v \times m \pmod{n} \quad (5.6)$$

où *modulo n* est défini en fonction de l'algorithme de chiffrement.

En conclusion, le développement d'une contre-mesure de type masking est plus facilement réalisable que le développement d'une contre-mesure de type hiding. En effet, ce type de contre-mesure consiste à s'intéresser à l'exécution des opérations d'un algorithme cryptographique (à ses valeurs intermédiaires) et d'ensuite modifier ces valeurs intermédiaires au travers de fonctions de masquage. Même si le principe semble facile en théorie, attention tout de même à la mise en application. En effet, certaines contre-mesures proposées sont considérablement compliquées à comprendre et à traduire en langage machine.

5.4 Contre-mesures Faking

Le choix de la contre-mesure à développer dans le cadre de mon travail ne m'ayant pas été imposé, après diverses recherches dans les livres de références et la littérature, j'ai trouvé un nouveau principe de contre-mesure. Ce type de contre-mesure se dénomme **Faking**. Cette contre-mesure s'appuie sur une partie du concept de contre-mesure par *Masking* mais avec quelques subtilités. C'est donc ce type de contre-mesure que j'ai décidé de développer dans le cadre de mon travail. Les résultats seront présentés au chapitre 8 (Partie 2).

5.4.1 Définition

L'objectif de la contre-mesure est le suivant : Dès le départ, on va venir masquer la vraie valeur de clé (Key_{Real}) en opérant un XOR avec un masque (Key_{Mask}) produisant ainsi une fausse clé (Key_{Fake}). On a donc la relation (5.7) :

$$Key_{Fake} = Key_{Real} \oplus Key_{Mask} \quad (5.7)$$

À partir de cette fausse clé, on va exécuter l'algorithme AES de façon ordinaire (tel que décrit à la section 2.2.3) : c'est-à-dire qu'on va tout d'abord appliquer l'opération *KeySchedule* sur la fausse clé générant ainsi d'autres fausses clés et ensuite on va exécuter les quatre opérations élémentaires (*AddRoundKey*, *SubBytes*, *ShiftRows*, *MixColumns*) afin de chiffrer les données. L'implémentation de cette contre-mesure est présentée à la figure 5.4. On constate ainsi que c'est la fausse clé qui est utilisée pour chiffrer les données et non la vraie clé. De cette manière, en supposant que l'attaquant ne connaisse pas la valeur du masque (*KeyMask*) appliquée sur la vraie clé, si ce dernier décide d'opérer une attaque de type CPA (ou autre) sur le device, il sera capable de retrouver une valeur de clé. Cependant, il s'agira de la fausse clé générée dès le départ. Par conséquent, l'attaquant sera incapable de déchiffrer les données étant donné qu'il possède une fausse clé.

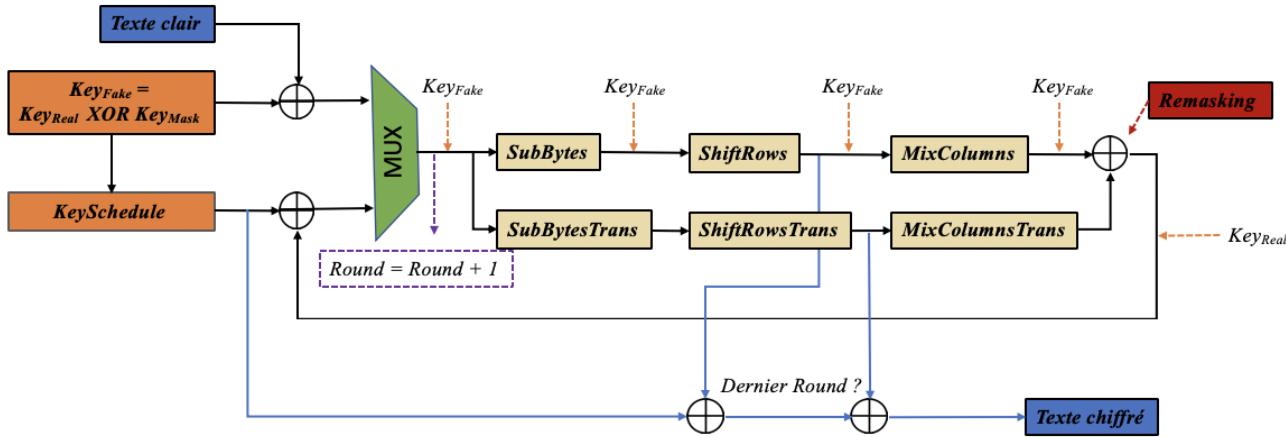


Figure 5.4 : Implémentation de la contre-mesure *faking* pour l'algorithme de chiffrement AES.

Dans les contre-mesures de type *Masking*, lorsqu'on applique un masque sur les données intermédiaires, il faut ensuite le retirer de façon à obtenir *in fine* le *vrai* texte chiffré. Pour le cas des contre-mesures de type *Faking*, le principe est identique. En effet, si aucune action supplémentaire n'est entreprise, le texte chiffré serait crypté avec la fausse clé plutôt qu'avec la vraie clé. Pour ce faire, on va ajouter trois nouvelles opérations dont le rôle est d'annuler l'emploi du masque sur la clé. Ces trois opérations sont *SubBytesTrans*, *ShiftRowsTrans* et *MixColumnsTrans*. Enfin, afin de définitivement couper tout lien des résultats intermédiaires avec la fausse clé, un XOR (opération *remasking*) est opéré entre les résultats intermédiaires obtenus après *MixColumns* et ceux obtenus après *MixColumnsTrans*. Le résultat de ce XOR permet de retirer l'emploi du masque, on retrouve alors le résultat que l'on aurait obtenu en sortie de l'opération *MixColumns* si on avait utilisé la vraie clé initialement (voir équation 5.13). Ceci est visible sur la figure 5.4. Ensuite, l'opération *AddRoundKey* emploie à nouveau une fausse clé, dérivant de l'opération *KeySchedule*.

Définissons la nouvelle opération *SubBytesTrans*. Pour ce faire, notons $KeyFake(i, j)$ (avec $i \in [0 : 3]$ et $j \in [0 : 3]$) la fausse clé utilisée, $KeyReal(i, j)$ la vraie clé utilisée et $P(i, j)$ l'ensemble de textes clairs (*plaintext* en anglais) envoyés au device cryptographique. Notons également $a_F(i, j)$ un byte particulier à la sortie de l'opération *AddRoundKey* chiffré avec la fausse clé *KeyFake* (voir figure 5.4) et notons $a_R(i, j)$ un byte particulier à la sortie de l'opération *AddRoundKey* mais cette fois-ci chiffré avec la vraie clé *KeyReal*. Ces deux notations sont reprises à l'équation 5.8.

$$\begin{cases} a_F(i, j) = KeyFake(i, j) \oplus P(i, j) \\ a_R(i, j) = KeyReal(i, j) \oplus P(i, j) \end{cases} \quad (5.8)$$

Étant donné que c'est la fausse clé qui est utilisée pour l'implémentation de la contre-mesure, si nous souhaitons obtenir le résultat intermédiaire à la sortie de l'opération *SubBytes*, nous avons la relation 5.9 :

$$Sbox(a_F(i, j)) = Sbox(KeyFake(i, j) \oplus P(i, j)) \quad (5.9)$$

Connaissant la relation 5.7, nous pouvons ré-écrire l'équation 5.9 sous la forme suivante :

$$Sbox(a_F(i, j)) = Sbox(Key_{Real}(i, j) \oplus Key_{Mask}(i, j) \oplus P(i, j)) \quad (5.10)$$

Ainsi, une façon simple de retrouver le byte original $a_R(i, j)$ utilisé avec la clé réelle à la sortie de l'opération $SubBytes$ est de définir une nouvelle fonction, appelée $SubBytesTrans$. Cette fonction est donc définie sur base de l'équation 5.10 de la manière suivante :

$$Sbox(a_F(i, j)) = Sbox(Key_{Real}(i, j) \oplus P(i, j)) \oplus SubBytesTrans(a_F(i, j)) \quad (5.11)$$

En effet, selon l'équation 5.8, on peut simplifier l'équation précédente 5.11 par l'équation suivante 5.12 où l'on retrouve bien le byte original $a_R(i, j)$.

$$Sbox(a_F(i, j)) = Sbox(a_R(i, j)) \oplus SubBytesTrans(a_F(i, j)) \quad (5.12)$$

Autrement dit, la fonction $SubBytesTrans$ se traduit par :

$$SubBytesTrans(a_F(i, j)) = Sbox(a_R(i, j)) \oplus Sbox(a_F(i, j)) \quad (5.13)$$

Concernant les opérations $ShiftRowsTrans$ et $MixColumnsTrans$, il ne s'agit en réalité pas de nouvelles opérations étant donné que ces deux opérations réalisent respectivement la même fonction que les opérations $ShiftRows$ et $MixColumns$ (tel que décrit à la section 2.2.3). Cependant, nous précisons à la section suivante (5.4.2) que le fait d'exécuter ces deux opérations permet d'éviter une faille dans la contre-mesure. C'est donc la raison pour laquelle, le *remasking* (opération XOR) ne se fait pas avant cette opération.

5.4.2 Points faibles et améliorations

Deux points faibles de l'implémentation de la contre-mesure faking proposés à la figure 5.4 sont repris ci-dessous et améliorés afin de diminuer grandement les chances de réussite de l'attaquant.

1. Le but de la contre-mesure faking est d'autoriser l'attaquant à trouver une fausse valeur de clé. Cependant, connaissant la valeur de cette fausse clé, si l'attaquant trouve la valeur du masque, il peut retrouver la valeur de la vraie clé selon l'équation 5.7. Ainsi, pour améliorer ce problème, il faut que chaque byte du masque utilisé (Key_{mask}) soit différent. Ceci complique la tâche de l'attaquant.
2. Il existe une attaque utilisant les résultats intermédiaires à la sortie des opérations $SubBytes$ et $SubBytesTrans$ et qui permet de retrouver la clé réelle. En effet, si l'attaquant récupère ces deux valeurs intermédiaires et qu'il réalise un XOR entre elles, il obtient la simplification suivante :

$$\begin{aligned} Sbox(a_F(i, j)) \oplus SubBytesTrans(a_F(i, j)) &= Sbox(a_R(i, j)) \\ &= Sbox(Key_{Real}(i, j) \oplus State(i, j)) \end{aligned}$$

En considérant le premier round de l'AES-256, l'élément $State(i, j)$ représente un texte clair ($P(i, j)$). Dans ce cas, connaissant le fonctionnement de l'opérations $SubBytes$, l'attaquant peut retrouver la valeur de la vraie clé Key_{Real} . Pour se prémunir de ce problème, une amélioration de la contre-mesure faking consiste à masquer les valeurs en sortie de l'opération $SubBytesTrans$. En effet, à chaque round, un masque aléatoire $M_h(i, j)$ est appliqué sur les résultats intermédiaires en sortie de l'opération $SubBytesTrans$ de sorte à obtenir l'équation suivante 5.14 :

$$SubBytesTrans(a_F(i, j)) = Sbox(a_R(i, j)) \oplus Sbox(a_F(i, j)) \oplus M_h(i, j) \quad (5.14)$$

Notons que le masque $M_h(i, j)$ est renouvelé de façon aléatoire à chaque round pour complexifier le chiffrement. Ce masque $M_h(i, j)$ devient le masque $M_s(i, j)$ après l'exécution de l'opération $ShiftRowsTrans$ et devient ensuite le masque $M_k(i, j)$ après l'exécution de l'opération $MixColumnsTrans$.

La figure 5.5 présente le principe de fonctionnement de la contre-mesure faking avec les améliorations apportées.

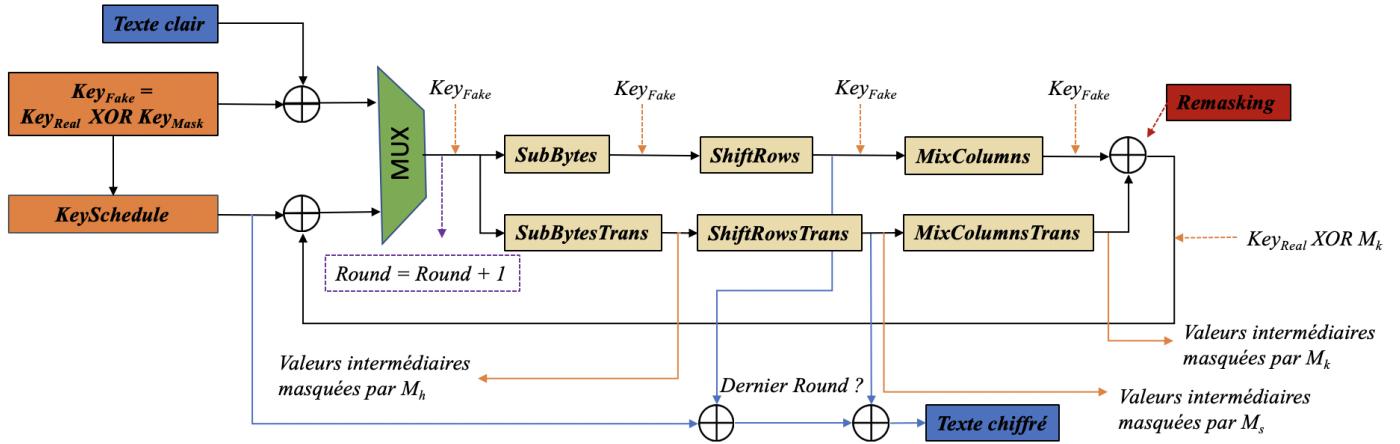


Figure 5.5 : Implémentation de la contre-mesure *faking* pour l'algorithme de chiffrement AES avec prise en compte des améliorations.

Reprendons l'exemple du chapitre précédent (chapitre 4) où l'on présentait les résultats (voir figures 4.6 et 4.7) d'une attaque CPA sur un FPGA implémentant l'algorithme AES-256. Pour rappel, l'attaque est réalisée sur le huitième byte de la clé dont la valeur attendue est 40. Considérons maintenant avoir implémenté une contre-mesure de type *faking* sur le device cryptographique (il s'agit d'une simulation réalisée sur MATLAB). Dans ce cas, si l'attaquant réalise une nouvelle attaque CPA sur le device, la clé qu'il obtient doit être différente de la vraie clé utilisée. Ceci se vérifie sur la figure 5.6. La vraie clé utilisée pour le chiffrement des données (40) ne laisse fuiter aucune information. Par contre, la clé laissant fuiter le plus d'information est bien la fausse clé (185). En effet, la valeur du masque étant 145, selon l'équation 5.7, nous trouvons :

$$\begin{aligned} KeyFake &= KeyReal \oplus KeyMask \\ &= 40 \oplus 145 \\ &= 185 \end{aligned}$$

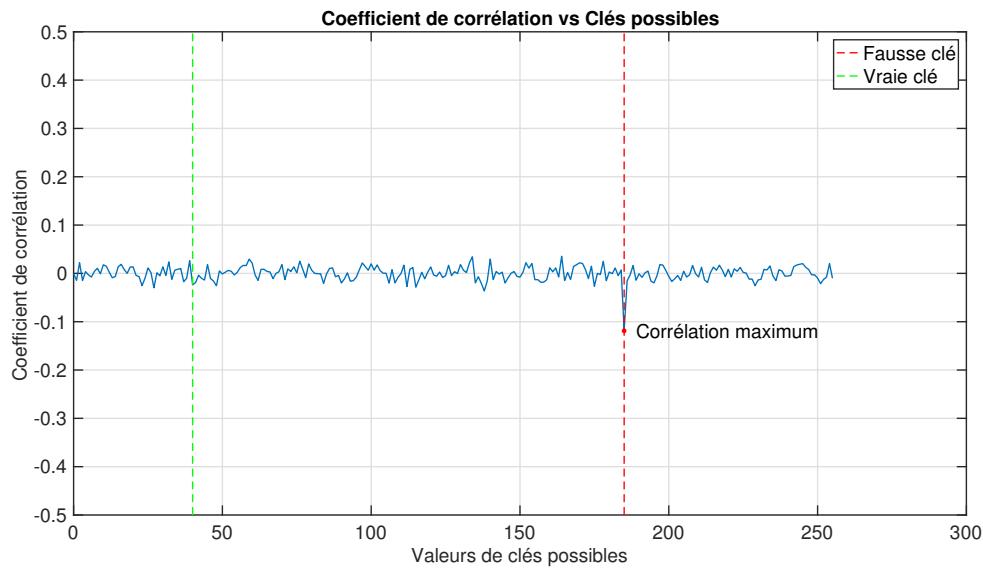


Figure 5.6 : Valeur de la clé obtenue avec implémentation d'une contre-mesure *faking*.

Deuxième partie

Mise en application

Mise en oeuvre de l'attaque CPA

Ce chapitre décrit la mise en oeuvre de l'attaque CPA. Une description des différents langages de programmation utilisés est tout d'abord abordée. Ensuite, deux sections sont respectivement réservées à la description de la configuration de l'oscilloscope et à la procédure mise en place pour le chiffrement des données ainsi qu'une description de la configuration de l'attaque CPA.

6.1 Langages de programmation

Trois langages de programmation ont été utiles à la réalisation de ce travail :

- **PYTHON** pour la configuration de la communication entre l'ordinateur et le FPGA et entre l'ordinateur et l'oscilloscope ;
- **VHDL** (*VHSIC Hardware Description Language*) pour l'implémentation de l'algorithme AES-256 et de la contre-mesure *Faking* sur le FPGA ;
- **MATLAB** (*MATrix LABoratory*) pour d'une part la mise en application de l'attaque CPA et d'autre part les simulations et réalisations de graphes utiles à la compréhension des notions théoriques.

6.2 Procédure de chiffrement et configuration de l'oscilloscope

La liste du matériel nécessaire pour le chiffrement de messages clairs est présentée ci-dessous :

1) Device cryptographique : le device cryptographique employé pour implémenter l'algorithme AES-256 est un FPGA. Plus précisément, il s'agit d'une carte de test spécialement conçue pour la recherche et le développement dans le domaine de la sécurité matérielle. Cette carte, visible sur la figure 6.1 et portant le nom *SAKURA-G*, est effectivement utilisée pour réaliser des attaques par canaux cachés mais également d'autres types d'attaques physiques comme les attaques par injections de fautes, etc. Cette carte intègre deux FPGAs de la famille *Spartan-6* : l'un est utilisé comme contrôleur (*Controller FPGA*) et l'autre comme circuit principal de sécurité (*Main FPGA*). Pour la réalisation de ce travail, le FPGA principal est configuré comme bloc de chiffrement (il implémente l'algorithme AES-256) tandis que le contrôleur est utilisé pour gérer la communication entre le FPGA principal et l'ordinateur. La fréquence d'horloge pour ces deux FPGAs est de 48 MHz. Trois points de mesures sont prévus sur la carte SAKURA-G pour analyser la consommation de puissance du FPGA principal. Ces trois connecteurs SMA (*SubMiniature version A*) sont visibles sur la figure 6.1 et plus précisément sur le schématique présenté à la figure 6.2. À l'inverse des connecteurs J_1 et J_2 , le connecteur J_3 amplifie la consommation de puissance du FPGA principal afin de mieux analyser la tension (voir section 4.2.3). Enfin, la carte SAKURA-G peut être alimentée de deux façons différentes : soit par l'interface du port USB soit par une alimentation externe.



Figure 6.1 : Carte SAKURA-G.

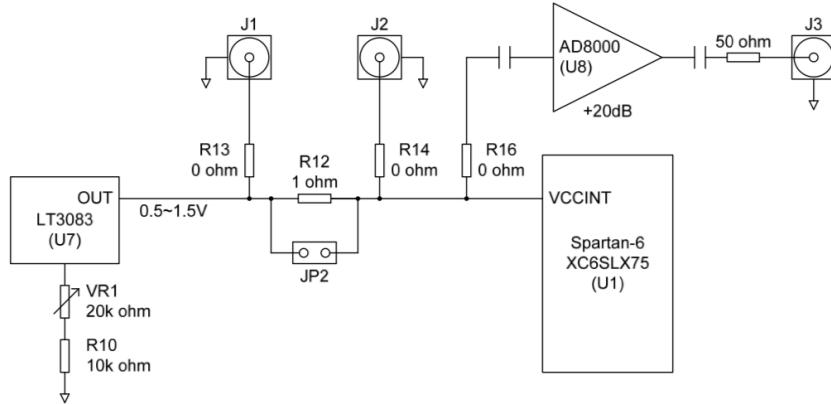


Figure 6.2 : Les trois points de mesures prévus pour analyser la consommation de puissance (la tension pour être exact) du FPGA principal. Seul le connecteur SMA J_3 est utilisé.

2) Alimentation de laboratoire : Comme précisé ci-avant, il existe deux façons d'alimenter la carte SAKURA-G : soit par l'interface du port USB, soit par une alimentation externe. Sur base du retour d'expérience d'un ancien étudiant stagiaire, il s'avère que l'alimentation via le port USB est instable, ce qui rend l'analyse de la consommation de puissance plus compliquée. Pour cette raison, une alimentation de laboratoire est utilisée et alimente la carte en $5V \pm 5\%$ (avec un courant limité à 3A).

3) Oscilloscope : L'oscilloscope est l'instrument de mesure utilisé pour analyser la consommation de puissance du device cryptographique (FPGA) lorsque celui-ci est en cours de chiffrement. Plus précisément, l'oscilloscope mesure la tension sur un pin du FPGA (voir section 4.2.3). L'oscilloscope utilisé est un Picoscope de la série 5000D possédant deux canaux de mesures (A et B), un trigger externe (permettant la synchronisation de signaux sur base d'un signal "déclencheur") et un canal prévu pour générer certaines formes de signaux (voir figure 6.3). Ce Picoscope possède une bande passante de 200 MHz et un échantillonnage pouvant aller jusqu'à 1 Gé/s. Sa résolution peut être configurée sur 8, 12, 14, 15 ou 16 bits. La

particularité de cet oscilloscope est que l'affichage, les commandes et l'alimentation (USB) s'effectuent exclusivement à partir de l'ordinateur auquel il est branché. Il existe une interface logicielle (voir figure 6.4) permettant de le configurer. Cependant pour la réalisation de ce travail, l'oscilloscope sera configuré et utilisé uniquement via un script python (utilisant une librairie en référence à ce Picoscope).



Figure 6.3 : Le Picoscope est l'instrument de mesure utilisé pour analyser la consommation de puissance (la tension pour être exact) du FPGA.



Figure 6.4 : Interface logicielle utilisée pour le Picoscope. La trace de puissance affichée correspond à l'exécution de l'algorithme AES-256 sur le FPGA.

Le tableau 6.1 présente les paramètres de configuration du Picoscope pour ce travail. Seul le canal A est utilisé pour la mesure de la tension. Le canal Ext est utilisé pour détecter le *trigger* généré par le FPGA. Ce *trigger* permet d'aligner (synchroniser) les traces entre elles, ce qui est nécessaire pour analyser correctement la corrélation entre les traces de puissance et les prédictions de puissance (voir section 4.2.3). Le canal B a initialement été utilisé pour détecter le *trigger* mais cela avait pour désavantage de réduire la résolution des traces et par conséquent le taux de succès de l'attaque. Pour cette raison, le canal Ext constitue une alternative intéressante. Concernant les paramètres de configuration du canal A, le nombre d'échantillons peut être calculé grâce à l'équation 6.1. Concernant le canal Ext, le délai représente le temps d'attente du programme si aucun trigger n'est détecté. Autrement dit, si le canal Ext ne détecte pas à 40 mv un flanc montant, il va patienter durant 3 secondes. Une fois ce délai dépassé, Le FPGA génère

un nouveau trigger. Enfin, le paramètre *post trigger* du canal Ext détermine la période de temps durant laquelle le Picoscope analyse le signal mesuré sur le canal A après que le trigger a été détecté. Autrement dit, ce paramètre définit la largeur de la fenêtre de mesure (dans ce cas, elle vaut 1510 ns).

Canal	Couplage	Plage de tension (mV)	Résolution	Nombre d'échantillons*
Canal A	AC	±200	12 bits	755
Canal B	/	/	/	/
Canal	Seuil de déclenchement (mV)	Sur flanc	Délai (ms)	Post trigger (ns)
Canal Ext	40	montant	3000	1500

Table 6.1 : Paramètres de configuration pour les canaux A, B et Ext du Picoscope.

*Pour déterminer le nombre d'échantillons capturés dans une trace, il faut se référer à la datasheet du Picoscope. Ce nombre dépend effectivement de différents facteurs tels que la résolution, le *timebase* ou encore le nombre de canaux activés. La figure 6.5 présente pour des résolutions sur 8 bits et 12 bits, différentes formules permettant de déterminer l'intervalle de temps entre deux échantillons successifs dans une trace. L'équation 6.1 permet de calculer le nombre d'échantillons capturés sur une trace pour une résolution de 12 bits et un *timebase* de 1 (configuration utilisée pour ce travail).

8-bit mode			
Timebase (n)	Sampling interval formula	Sampling interval	Notes
0		1 ns	Only one channel enabled
1	$2^n / 1,000,000,000$	2 ns	
2		4 ns	
3		8 ns	
...		...	
$2^{32}-1$	$(n-2) / 125,000,000$	~ 34.36 s	

12-bit mode			
Timebase (n)**	Sampling interval formula	Sampling interval	Notes
1		2 ns	Only one channel enabled
2	$2^{(n-1)} / 500,000,000$	4 ns	
3		8 ns	
4		16 ns	
...		...	
$2^{32}-2$	$(n-3) / 62,500,000$	~ 68.72 s	

Figure 6.5 : Formules permettant de déterminer l'intervalle d'échantillonnage (tableaux repris de la *datasheet* du Picoscope).

Selon la figure 6.5, pour une résolution sur 12 bits et pour un *timebase* de 1, l'intervalle de temps entre chaque échantillon dans une trace vaut 2 ns. De plus, si on considère que la largeur de la fenêtre de mesure vaut 1500 ns, alors on obtient le résultat suivant (6.1) :

$$\text{Nombre d'échantillons} = \frac{\text{Valeur Post trigger}}{\text{Intervalle de temps entre 2 échantillons}} = \frac{1500}{2} = 755 \quad (6.1)$$

3) Ordinateur : D'une part, l'ordinateur est utilisé pour configurer et sauvegarder toutes les données provenant de l'oscilloscope (les traces) mais aussi toutes les données provenant du FPGA (les textes chiffrés). D'autre part, l'ordinateur fournit la puissance de calcul nécessaire à l'exécution des différents scripts (python, matlab) utiles pour l'analyse de résultats. Ces différents scripts écrits en Python et sous Matlab sont mentionnés ci-après.

La figure 6.6 regroupe les différents matériels/instruments de mesure cités précédemment et utilisés d'une part pour chiffrer des données claires et récupérer les résultats s'y rapportant (textes chiffrés) et d'autre part pour mesurer la consommation de puissance du device cryptographique lorsque celui-ci est en cours de chiffrement.

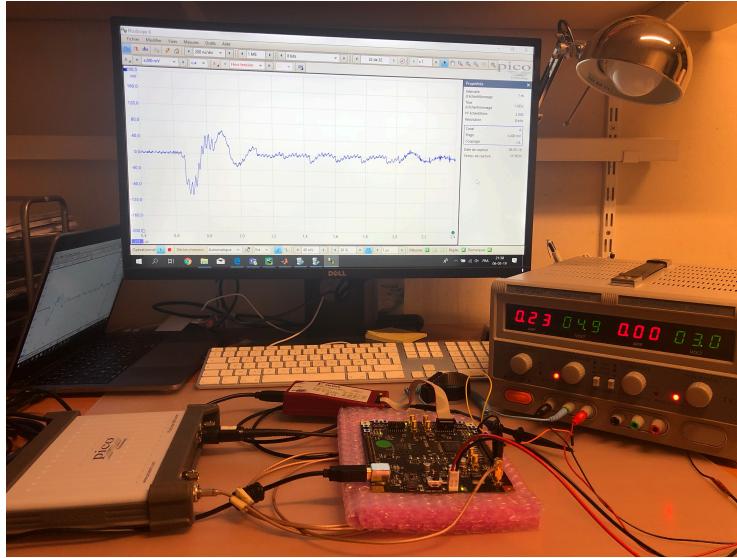


Figure 6.6 : Banc de mesures.

La figure 6.7 décrit la procédure utilisée pour configurer les différents systèmes informatiques employés (FPGA, oscilloscope, ordinateur) ainsi que pour chiffrer différents messages clairs. Cette procédure se compose de six étapes décrites ci-après. Pour cette procédure, on considère que le code VHDL de l'algorithme AES-256 est déjà implémenté sur le FPGA.

1. Configuration du protocole de communication entre l'oscilloscope et l'ordinateur (Python) ;
2. Configuration du protocole de communication entre le FPGA et l'ordinateur (Python) ;
3. Envoi de la clé de chiffrement (K) et d'un texte clair (P) au FPGA depuis l'ordinateur (Python) ;
4. Chiffrement du message clair sur le FPGA et génération d'un trigger par le FPGA pour aligner les traces sur l'oscilloscope (VHDL) ;
5. Une fois le chiffrement du message clair terminé :
 - (a) Sauvegarde sur ordinateur de la trace de puissances capturée à l'oscilloscope (Python) ;
 - (b) Sauvegarde sur ordinateur du message chiffré par le FPGA (Python) ;
6. Étant donné qu'il faut envoyer un grand nombre (N) de messages clairs au FPGA pour que l'attaque CPA fonctionne aisément (voir section 4.2.3) : répétition des étapes 3 à 5 jusqu'à ce que le nombre de messages clairs à envoyer (N) soit atteint.

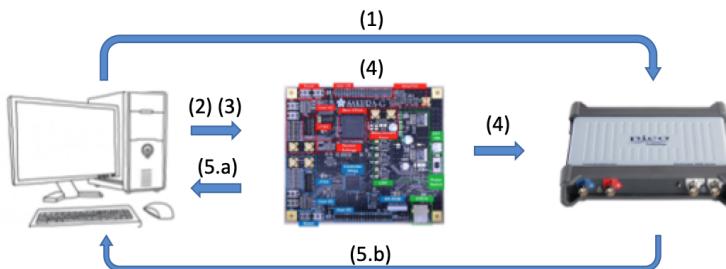


Figure 6.7 : Procédure mise en place pour le chiffrement de textes clairs et l'enregistrement des traces de puissance capturées à l'oscilloscope.

6.3 Procédure appliquée pour l'attaque CPA

L'attaque CPA s'exécute uniquement via un script MATLAB. Ce script va d'une part reprendre les traces de puissance enregistrées à l'oscilloscope (voir section 6.2) et d'autre part calculer des prédictions de puissance sur base des textes clairs envoyés au FPGA. La figure 6.8 ci-dessous décrit la procédure utilisée pour appliquer une attaque CPA sur l'implémentation de l'algorithme AES-256. Cette procédure se compose de six étapes décrites ci-après. Pour cette procédure, on considère que l'enregistrement des traces a déjà été réalisé (voir procédure précédente, section 6.2).

1. Récupération et chargement des N traces de puissance capturées à l'oscilloscope ;
2. Récupération et chargement des N messages clairs envoyés au FPGA pour être chiffrés ;
3. Simulation des opérations *AddRoundKey* et *SubBytes* pour tester toutes les valeurs de clés possibles ;
4. Utilisation d'un modèle de puissance pour prédire la consommation du FPGA (Poids de *Hamming*) ;
5. Calcul de la corrélation entre les traces de puissance et les prédictions de consommation de puissance ;
6. Détermination de la clé secrète sur base de la corrélation maximum ;



Figure 6.8 : Procédure mise en place pour réaliser une attaque par analyse de la consommation de puissance de type CPA.

Attaque CPA sur l'implémentation de l'AES-256

Ce chapitre présente les résultats de l'attaque CPA obtenus d'une part par simulation et d'autre part par une attaque réelle sur l'implémentation de l'algorithme AES-256. Tout d'abord, une description de l'implémentation de l'algorithme AES-256 et du contrôleur est abordée. Ensuite, les résultats de simulation de l'attaque CPA sont décrits. Ces simulations ont été réalisées à l'aide du logiciel MATLAB afin de bien apprêhender les notions théoriques (voir partie 1). Enfin, les résultats expérimentaux de l'attaque CPA sur le circuit sont présentés et discutés.

7.1 Implémentation de l'algorithme AES-256 et du contrôleur

Les codes *VHDL* pour l'implémentation de l'algorithme AES-256 et pour l'implémentation du contrôleur ont été repris du travail d'un ancien étudiant stagiaire. La compréhension de ces codes est une tâche nécessaire pour pouvoir implémenter par la suite la contre-mesure de type *faking* (chapitre 8). Comme présenté au chapitre 6, la carte SAKURA-G est composée de deux FPGAs :

- Un FPGA principal utilisé comme bloc de chiffrement (algorithme AES-256) ;
- Un FPGA secondaire utilisé comme contrôleur.

Les deux sous-sections suivantes décrivent brièvement l'implémentation de l'AES-256 et du contrôleur.

7.1.1 Implémentation AES-256

La figure 7.1 donne une représentation haut niveau du bloc de chiffrement (AES-256). Ce bloc de chiffrement prend en entrée une clé sur 256 bits et un message clair sur 128 bits. Le port *start* en entrée permet de débuter le chiffrement alors que le port *done* en sortie indique lorsque le chiffrement est terminé. Une *clock* est utilisée à la fréquence de 48 MHz et un bouton *reset* permet d'effacer les données contenues dans les différents registres utilisés durant le chiffrement. Enfin, précisons que le port de sortie *Ciphertext* renvoie le message chiffré sur 128 bits.

La figure 7.2 présente l'architecture implémentée pour l'algorithme AES-256. Dans cette architecture, l'opération *KeySchedule* est exécutée en premier lieu afin de fournir les nouvelles clés qui seront utilisées pour les *rounds* suivants. Les quatre autres opérations principales de l'AES (*AddRoundKey*, *SubBytes*, *ShiftRows* et *MixColumns*) sont chacune exécutées en un seul coup d'horloge. L'opération *Key Schedule* compte quant à elle X coups d'horloge. Sachant que l'AES-256 comprend 14 *rounds* (voir section 2.2.3), cette implémentation requiert donc un total de 56+X, soit X coups d'horloge pour chiffrer un bloc de 128 bits.

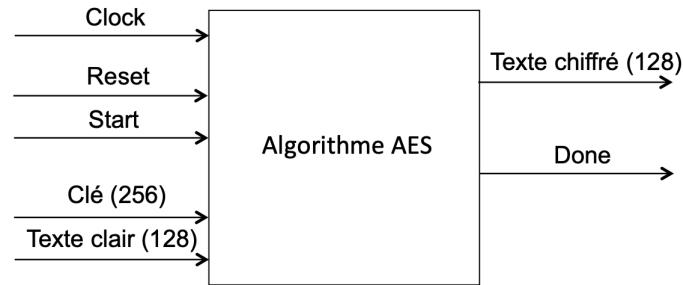


Figure 7.1 : Représentation haut niveau du bloc de chiffrement.

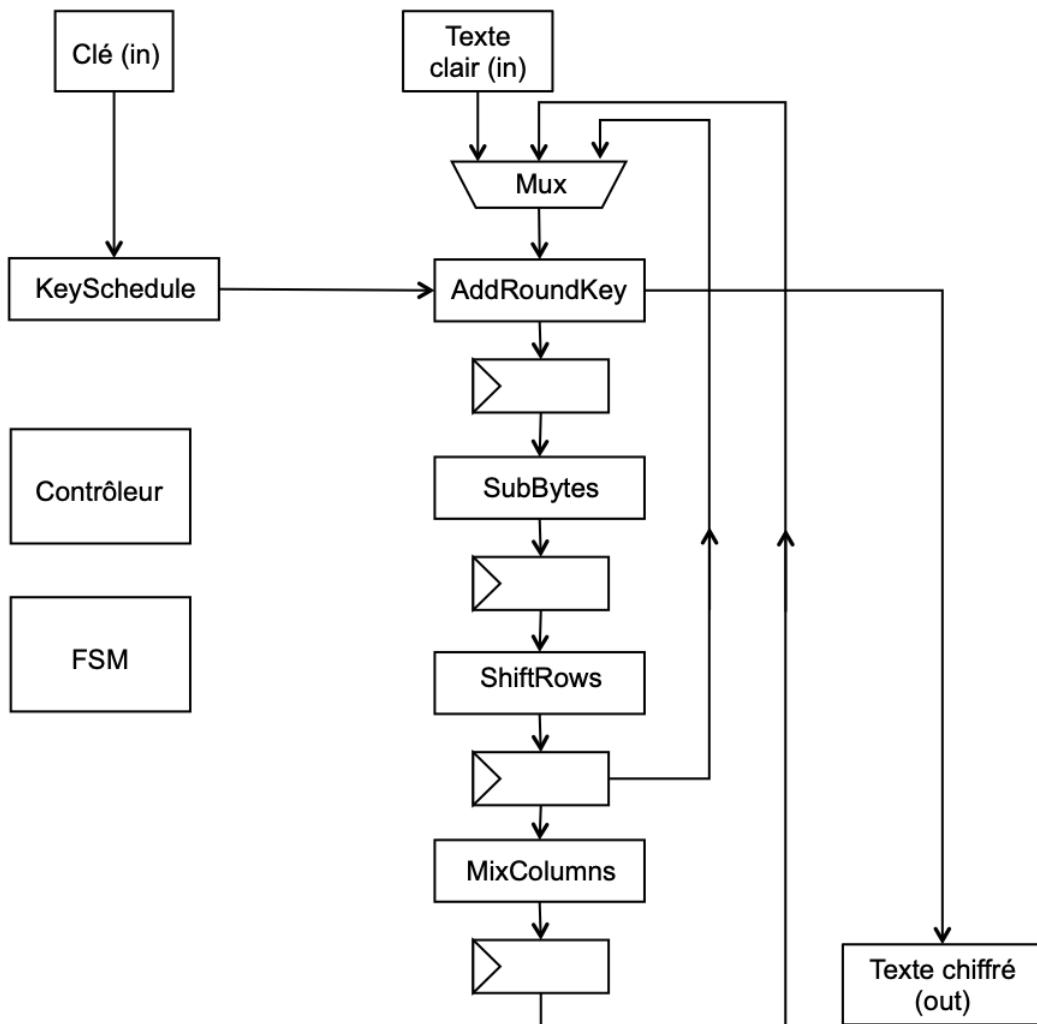


Figure 7.2 : Architecture de l'algorithme AES-256 implémentée sur le FPGA principal.

7.1.2 Implémentation contrôleur

Un bloc de chiffrement (AES-256) est implémenté sur le FPGA principal. L'étape suivante consiste à trouver une façon d'envoyer les données claires à ce bloc de chiffrement et à recevoir en retour les données chiffrées. La solution est d'utiliser le second FPGA comme contrôleur. Ainsi, cette sous-section décrit brièvement la chaîne de communication entre l'ordinateur et le bloc de chiffrement via ce contrôleur. La figure 7.3 présente le schéma-bloc de cette chaîne de communication. Quatre grands blocs sont parfaitement identifiables sur la figure : l'USB, la puce FT2232H, le contrôleur et le FPGA principal. La puce FT2232H se trouve sur la carte SAKURA-G et permet de réaliser une communication série entre le FPGA et l'ordinateur via le port USB. Cette puce peut être configurée dans une grande variété d'interfaces standards. Voici quelques exemples de protocoles possibles avec cette puce : UART, 245 FIFO, SPI, JTAG, etc. Le protocole conseillé dans la *datasheet* de la carte SAKURA-G et mis en place par l'étudiant stagiaire précédent est le protocole 245 FIFO Sync [??]. Ce protocole ne sera pas plus détaillé. Cependant, sachez que pour transférer les données depuis l'ordinateur vers le bloc de chiffrement (et inversement), trois interfaces sont nécessaires, chacune permettant de transférer 1 byte de données à la fois. Enfin, pour que le bloc de chiffrement puisse distinguer les données qu'il reçoit (distinguer la clé du message clair), un byte de tête est utilisé pour indiquer l'identité de la donnée qui va être envoyée par l'ordinateur. En effet, les données envoyées depuis le script Python sont transmises par paquet de 17 bytes. Le premier byte sert d'entête et indique le type de donnée qui va suivre sur les 16 bytes suivants. Le tableau 7.1 présente les trois différentes entêtes possibles concernant l'AES-256 ("000", "001", "100") et concernant également la contre-mesure *faking* ("010", "011"). Pour rappel, l'algorithme AES-256 chiffre les données par bloc de 128 bits (16 bytes) et la clé de chiffrement a une taille de 32 bytes.

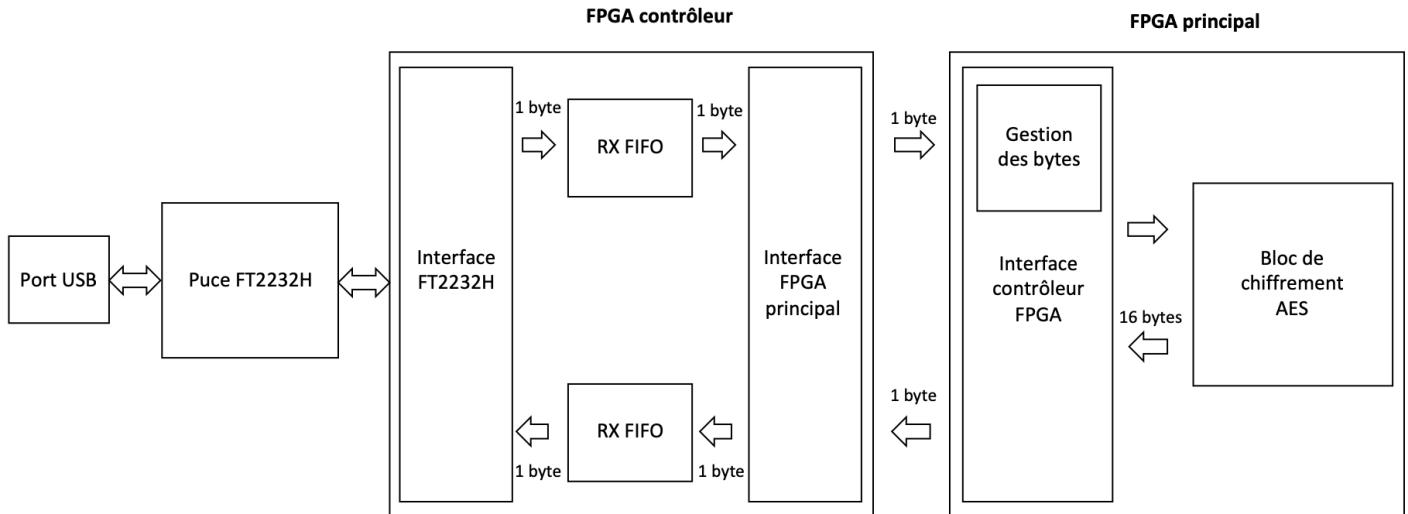


Figure 7.3 : Chaîne de communication entre l'ordinateur et le FPGA principal.

Bit [2:0]	Type de donnée
000	Clé [255:128]
001	Clé [127:0]
010	Masque [255:128]
011	Masque [127:0]
100	Message clair[127:0]

Table 7.1 : Identification des données utiles au bloc de chiffrement par l'entremise d'une entête sur un byte.

7.2 Résultats de simulations

Avant d'attaquer une réelle implémentation de l'AES-256 sur un FPGA, différentes simulations ont été mises en oeuvre à l'aide du logiciel MATLAB. Ces simulations ont permis de mieux appréhender le concept de l'attaque CPA. Les sections 7.2.1 et 7.2.2 décrivent respectivement les résultats obtenus sur base de traces simulées sans bruit et avec bruit. L'importance du bruit dans une trace est ainsi mise en évidence au travers de ces simulations. L'attaque CPA s'opère sur les deux premiers *rounds* de l'AES. En effet, sachant que la taille de la clé de chiffrement pour l'AES-256 est de 32 bytes et sachant que l'algorithme ne chiffre des messages que par bloc de 128 bits, deux *rounds* sont nécessaires pour retrouver les 32 bytes de la clé. Le premier *round* permet de retrouver les 16 premiers bytes de la clé tandis que le second *round* permet de retrouver les 16 derniers bytes de la clé. Le poids de Hamming (voir section 3.2.1) est utilisé en tant que modèle de prédiction de puissance pour ces simulations.

7.2.1 Traces simulées idéales

Dans ce premier exemple, nous considérons le cas idéal où les traces ne sont pas bruitées. La figure 7.4 présente le résultat de l'attaque CPA (telle que décrite à la section 4.2.3) sur un byte de l'algorithme AES-256. Plus précisément, l'attaque cible le dixième byte de la clé de chiffrement et a utilisé 1000 traces. Ce graphe présente la valeur du coefficient de corrélation pour chacune des 256 valeurs de clé testées par l'attaquant. Nous remarquons que la valeur maximum des 256 coefficients de corrélation calculés vaut 1 et correspond à la clé 200. C'est effectivement cette valeur décimale qui est utilisée en tant que dixième byte de la clé secrète. Il est normal que la corrélation vaille 1 étant donné qu'il s'agit d'une trace idéale sans bruit (voir section 4.2.3 pour le calcul du coefficient de corrélation). Enfin, notons que la valeur maximale de corrélation (1) se distingue clairement des 255 autres valeurs obtenues. Cependant, nous allons voir par la suite que cette différence n'est pas toujours aussi marquée.



Figure 7.4 : Résultat de l'attaque CPA sur le dixième byte de la clé. Ce graphe présente la valeur du coefficient de corrélation pour chacune des 256 valeurs de clé testées. La valeur de corrélation maximum vaut 1 et correspond à la clé 200. 1000 traces sont utilisées pour obtenir ce résultat.

Comme indiqué à la section 4.2.3, le nombre de traces employées pour réaliser l'attaque CPA influence le résultat obtenu. Pour rappel, lorsqu'un message clair est envoyé au bloc de chiffrement, une trace est enregistrée grâce à l'oscilloscope. Par conséquent, lorsqu'on augmente le nombre de messages clairs envoyés au bloc de chiffrement, le nombre de traces augmente proportionnellement. En théorie, plus le nombre de traces employées est grand, plus la corrélation entre cet ensemble de traces et l'ensemble des valeurs prédictes de consommation doit être :

- Élevée lorsque la bonne clé est testée ;
- Faible lorsqu'une mauvaise clé est testée ;

À titre d'exemple, la figure 7.5 présente la corrélation obtenue pour les 256 valeurs différentes de clé testées lorsque 10, 100 et 1000 traces sont employées. Il s'agit du même exemple que celui cité précédemment (attaque CPA sur le dixième byte de la clé). La figure 7.5 montre clairement que plus le nombre de traces augmente, plus la corrélation obtenue pour la bonne clé est prédominante comparativement aux 255 autres valeurs de clé testées.

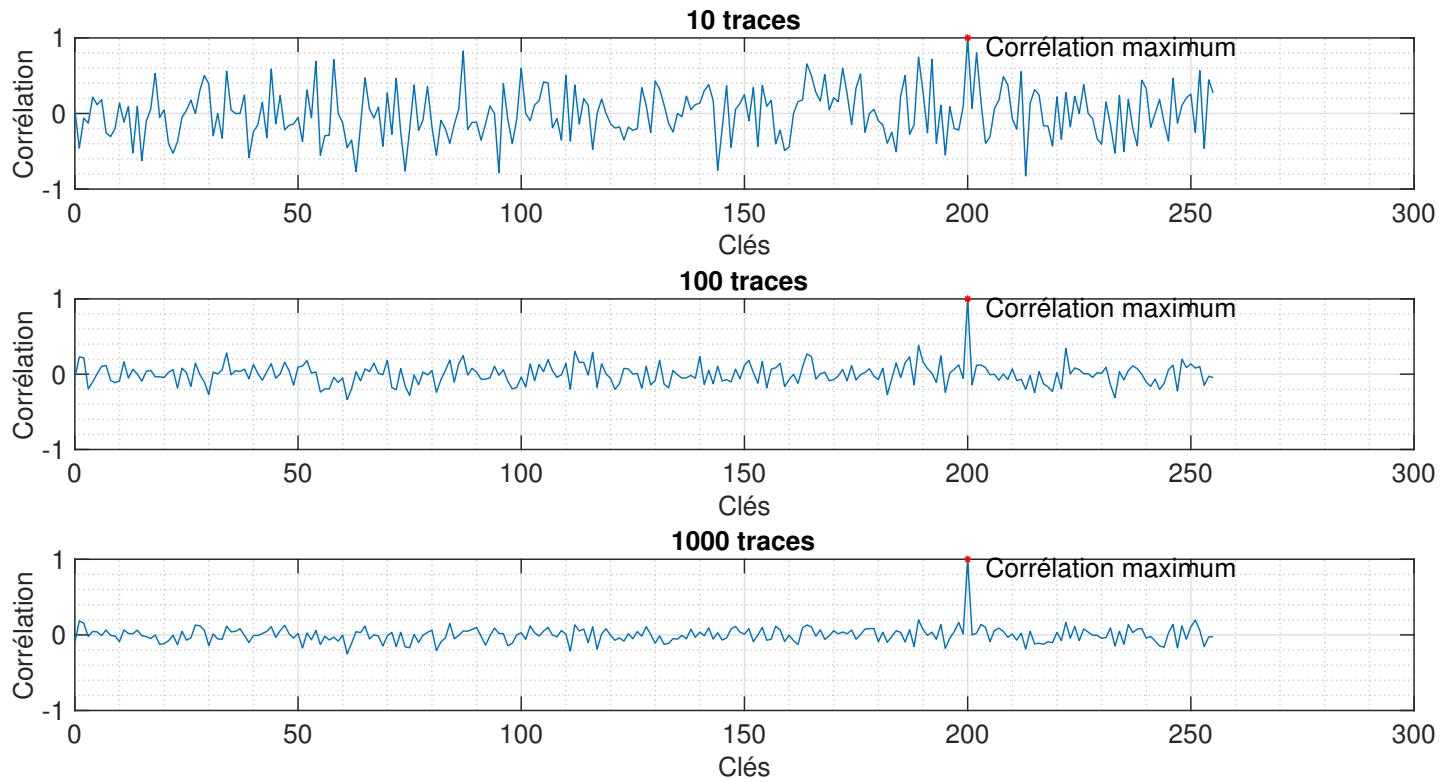


Figure 7.5 : Influence du nombre de traces employées pour la réalisation de l'attaque CPA. Plus le nombre de traces augmente, plus la corrélation obtenue pour la bonne clé testée est prédominante par rapport aux 255 autres.

Pour montrer que l'augmentation du nombre de traces permet de révéler plus facilement la valeur de la clé secrète, on utilise un graphe comme celui présenté à la figure 7.6. Pour une attaque sur le dixième byte de la clé, ce graphe présente la corrélation obtenue en fonction du nombre de traces utilisées. Plus le nombre de traces employées augmente, plus la bonne clé testée (200) se distingue des 255 autres clés testées. Sur ce graphe, on remarque que la bonne clé est révélée à partir d'une seule trace. Ceci est impossible en réalité à cause du bruit. La section suivante ajoute ainsi dans les traces un bruit *gaussien* de moyenne μ et d'écart-type σ afin que les résultats obtenus par simulations soient plus réalistes.



Figure 7.6 : Influence du nombre de traces employées pour la réalisation de l'attaque CPA. Plus le nombre de traces augmente, plus il est facile de retrouver la valeur de la clé secrète.

7.2.2 Traces simulées avec addition de bruit

Le bruit étant systématiquement présent dans les circuits électroniques, cette section met en évidence l'impact du bruit dans le résultat de l'attaque CPA. En théorie, plus le bruit est présent, plus il est difficile de retrouver la valeur exacte de la clé secrète. Ceci est dû au fait que la trace capturée à l'oscilloscope ne représente plus uniquement la consommation de puissance du FPGA mais elle représente cette consommation de puissance additionnée d'un bruit parasite. Ainsi, plus le bruit devient important, moins les traces représentent la puissance consommée par le FPGA et donc plus il est difficile de retrouver la clé secrète. Le bruit ajouté sur les traces simulées est un bruit *gaussien*. Un bruit *gaussien* suit une loi normale de moyenne μ et d'écart-type σ .

La figure 7.7 présente le résultat d'une attaque CPA sur le dixième byte de la clé de chiffrement. Les mêmes ensembles de messages clairs ont été repris de l'exemple précédent (voir section 7.2.1) afin de comparer efficacement l'impact du bruit sur les résultats. Le bruit additionné pour cet exemple possède une moyenne égale à 0 et une écart-type égal à 15. Nous pouvons remarquer que la valeur de corrélation pour la clé secrète est désormais nettement inférieure à 1 alors qu'elle était égal à 1 sans bruit (voir figure 7.6). Nous remarquons également que pour cet exemple, la clé secrète est toujours retrouvée à partir d'approximativement 800 traces.

Pour mieux montrer l'impact du bruit, une métrique est utilisée afin de mesurer le taux de succès de l'attaque en fonction du nombre de traces utilisées. Ce taux de succès est également fonction de la quantité de bruit ajoutée dans les traces simulées (et qui dépend du paramètre σ). La figure 7.8 présente le taux de succès de l'attaque CPA sur le dixième byte de la clé secrète. Trois bruits différents sont ajoutés dans les traces. Ces trois bruits ont chacun une moyenne nulle mais un écart-type différent : $\sigma = 10$, $\sigma = 15$ et $\sigma = 20$. Nous remarquons que la clé secrète peut, en moyenne, être révélée à partir de 1000 traces lorsque les traces sont bruitées avec un σ égal à 15, ce qui correspond avec le graphe présenté à la figure 7.7. On se rend ainsi mieux compte de l'impact du bruit dans le résultat de l'attaque CPA.



Figure 7.7 : Attaque sur le dixième byte de la clé avec pour objectif la mise en évidence de l'influence du bruit sur le résultat de l'attaque CPA. La présence du bruit complexifie la tâche de l'attaquant.



Figure 7.8 : Taux de succès de l'attaque CPA en fonction du nombre de traces utilisées. Les 3 courbes présentées sur le graphe correspondent à des valeurs différentes de σ (pour le bruit ajouté). La courbe orange présente le résultat pour $\sigma = 15$, qui correspond à l'exemple présenté à la figure 7.7.

7.3 Résultats expérimentaux

Cette section présente les résultats expérimentaux obtenus lors de la mise en application de l'attaque CPA (voir section 6.3) sur l'implémentation de l'AES-256 (voir 7.1.1).

7.3.1 Traces réelles brutes

La figure 7.9 présente une trace brute capturée à l'oscilloscope. La tension consommée par le FPGA varie entre -104 mV et 56 mV et l'algorithme met approximativement 1500 ns pour chiffrer un message clair. Cette trace ne sera pas plus analysée en profondeur. Cependant, nous pouvons affirmer que la première forme représentée sur la trace, située entre 100 ns et 300 ns (et qui représente grossièrement un "V"), correspond à la première opération exécutée par l'AES-256, c'est-à-dire l'opération *KeySchedule*.

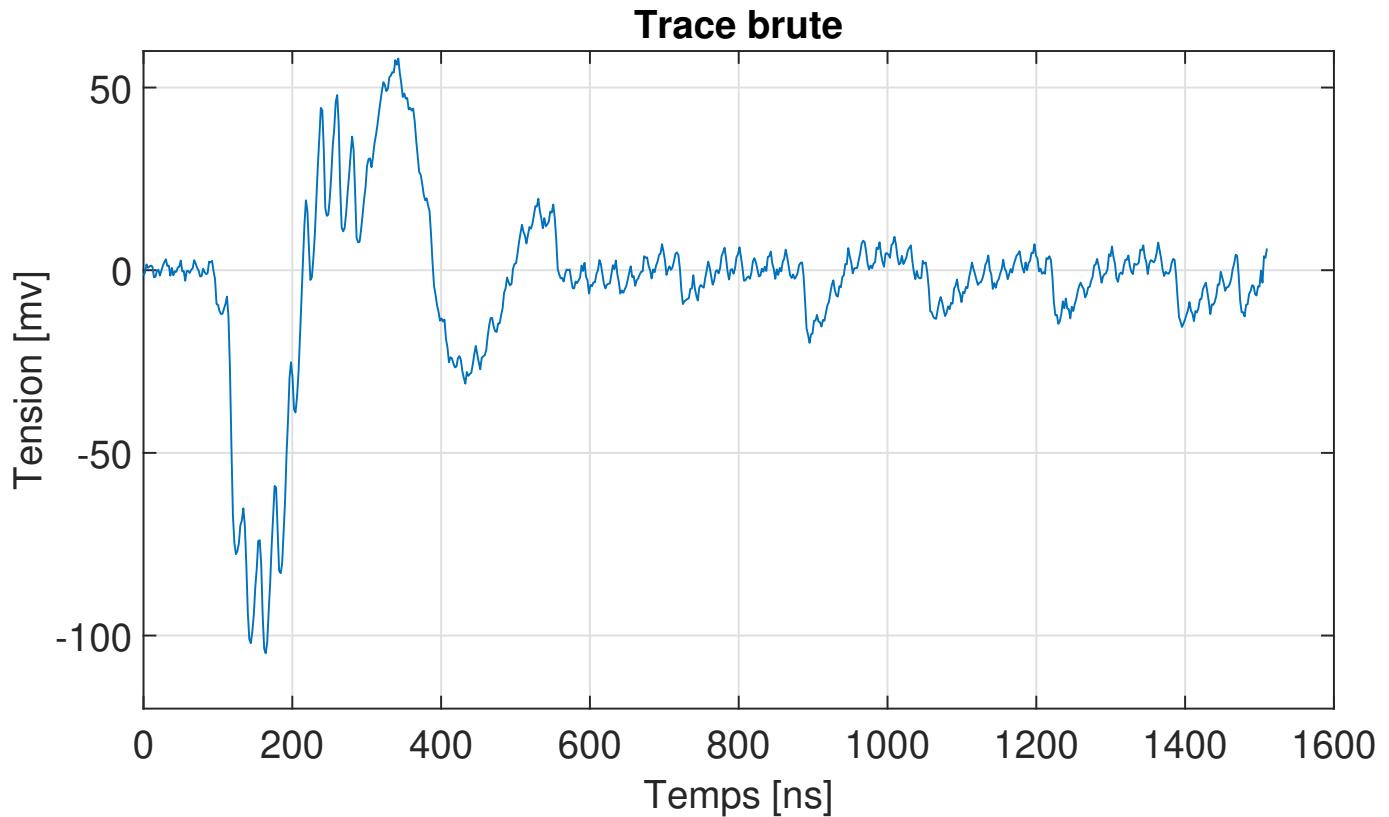


Figure 7.9 : Trace brute capturée à l'oscilloscope. Cette trace présente la tension consommée par le FPGA lorsque celui-ci est en cours de chiffrement.

La figure 7.10 présente le résultat de l'attaque CPA sur un byte de l'algorithme AES-256. Plus précisément, l'attaque cible le quatorzième byte de la clé secrète. Pour réaliser cette attaque, 5000 messages clairs ont été envoyés au FPGA, ce qui correspond donc à 5000 traces de puissance enregistrées à l'oscilloscope. Le résultat de cette attaque montre que les 256 valeurs du coefficient de corrélation sont extrêmement faibles. La valeur maximale de corrélation vaut -0,085 et révèle la valeur décimale 13 pour le quatorzième byte de la clé secrète (ce qui est correct). Les 255 autres valeurs de corrélation sont comprises dans l'intervalle [-0,5 : 0,4]. Cela montre que la différence entre la valeur maximale de corrélation et les autres valeurs de corrélation reste faible (0,035 de différence). Il est donc nécessaire d'utiliser un grand nombre de traces pour retrouver la bonne clé de chiffrement. Le nombre de traces minimum nécessaire à l'attaquant pour réussir une attaque CPA est présenté ci-après.

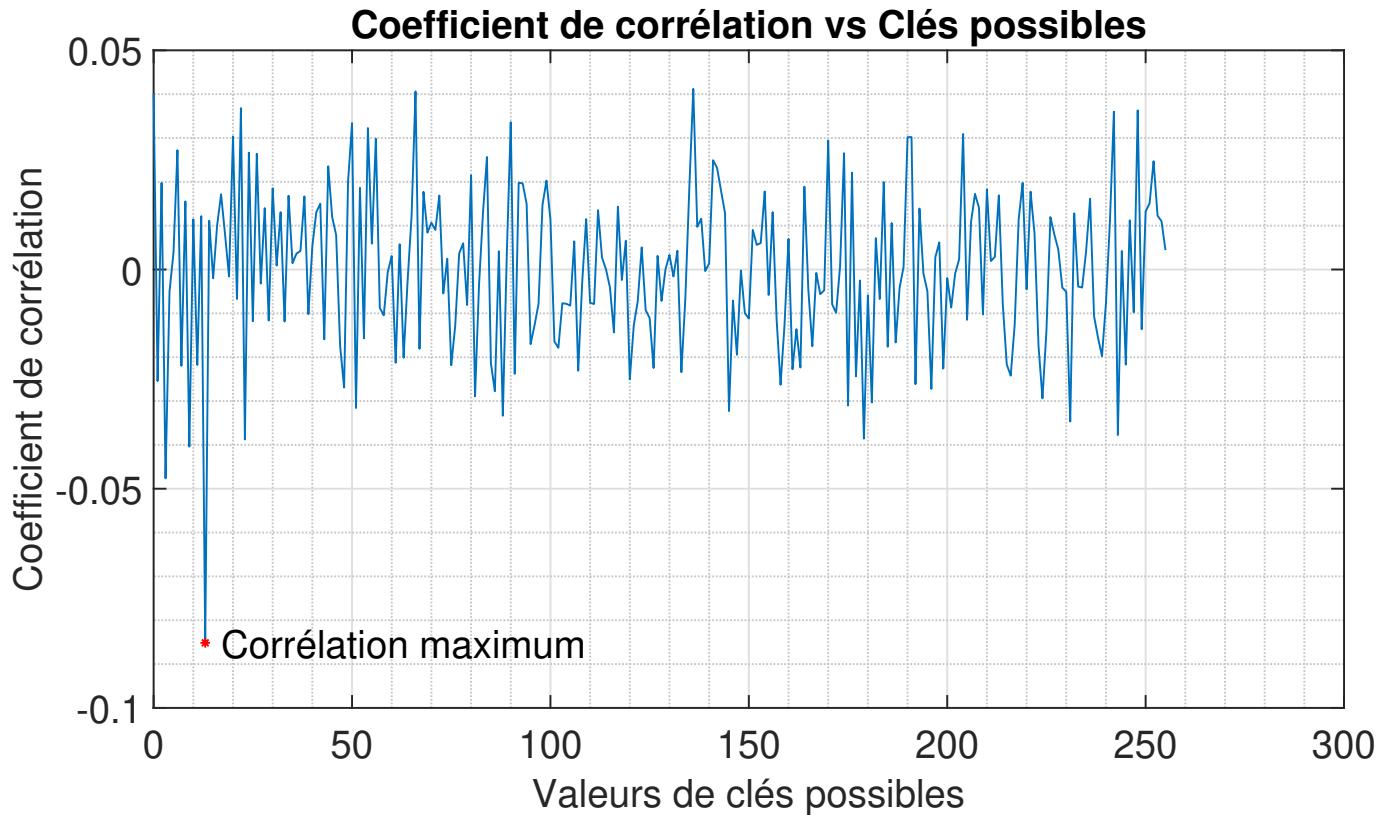


Figure 7.10 : Résultat de l'attaque CPA sur le quatorzième byte de la clé. Ce graphe présente la valeur du coefficient de corrélation pour chacune des 256 valeurs de clé testées. La valeur de corrélation maximum vaut -0,085 et révèle la valeur décimale 13 pour le quatorzième byte de la clé.

La figure 7.11 indique le nombre de traces minimum qu'un attaquant doit utiliser pour que l'attaque CPA puisse toujours permettre de révéler la clé de chiffrement. Ce graphe présente les résultats de l'attaque CPA pour le quatorzième byte de la clé. On constate qu'au-dessus d'approximativement 1400 traces, l'attaque CPA révèle toujours la valeur de la clé secrète. De plus, au plus le nombre de traces augmente, au plus la clé secrète se distingue des 255 autres clés testées par l'attaquant. À noter que le graphe peut être totalement différent pour les 31 autres bytes de la clé.

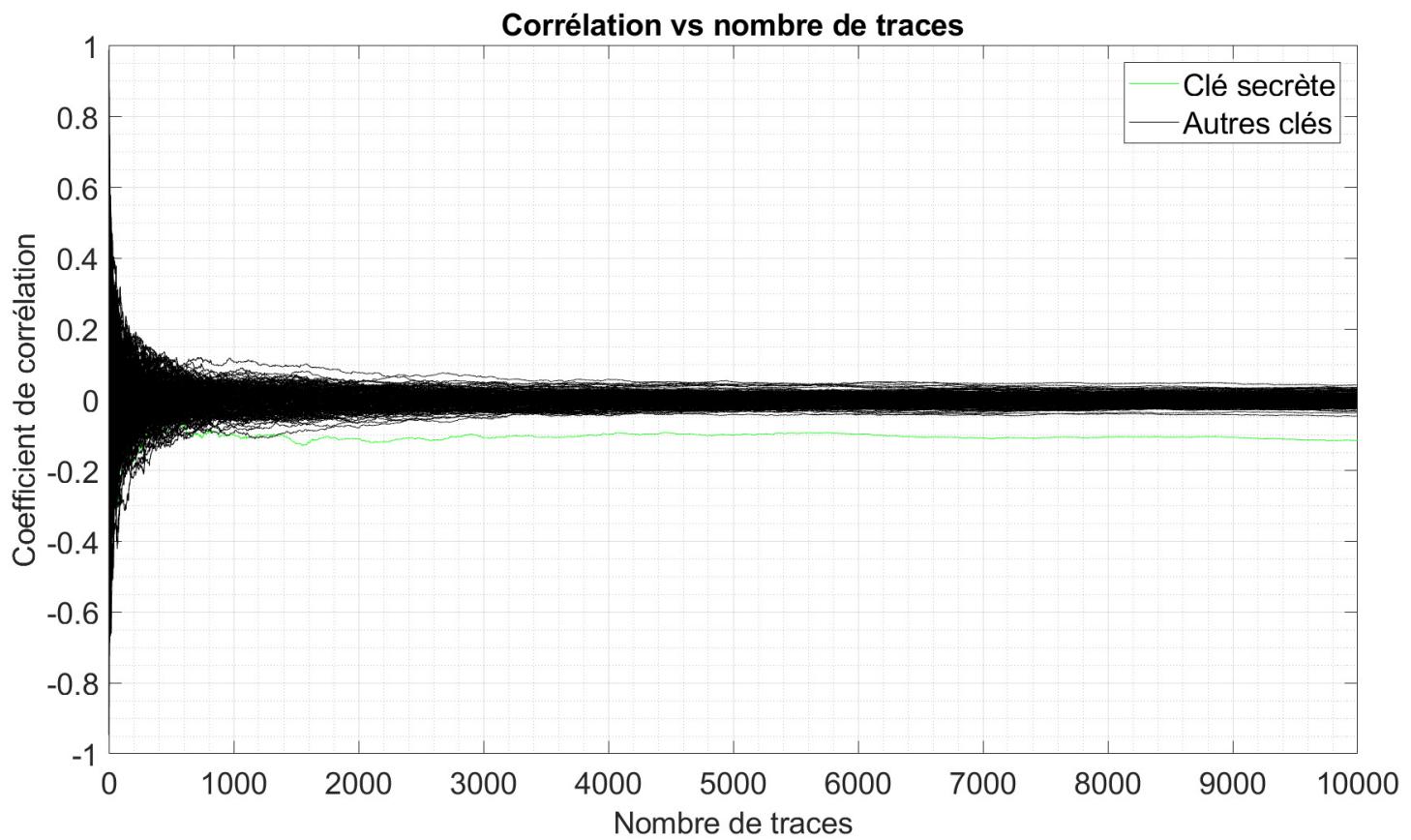


Figure 7.11 : Attaque CPA sur le quatorzième byte de la clé de chiffrement. Ce graphe indique qu'il faut un minimum de 1400 traces à l'attaquant pour que l'attaque CPA fonctionne tout le temps.

7.3.2 Traces réelles filtrées

À la section 7.2, l'influence du bruit dans les traces analysées a été mise en évidence. De ce fait, un pré-traitement des traces peut être opéré afin de retirer le bruit parasite inutile. Une façon efficace d'enlever ce bruit parasite est d'utiliser un filtre. L'utilisation de ce filtre a pour objectif d'augmenter le taux de succès de l'attaque. Sachant que le Picoscope possède une bande passante de 200 MHz et qu'il échantillonne à 500 Mé/s (pour une résolution sur 12 bits) et sachant que la fréquence d'horloge utilisée par le FPGA est de 48 MHz (voir section 6.2 pour la description du matériel utilisé), l'idée consiste à utiliser un filtre numérique de type passe-bas. Plus précisément, le filtre employé est un filtre à réponse impulsionnelle finie (*FIR*) d'ordre 100 et dont la fréquence de coupure est définie à 48 MHz. La fenêtre employée est une fenêtre de *Hamming* [??] dont les caractéristiques sont reprises à l'annexe C.1. La figure 7.12 compare une trace brute à une trace filtrée. L'application du filtre a pour effet de *lisser* la trace. Cependant, quand il s'agit d'analyser le traitement d'un signal, il est plus commode (plus facile) d'observer ce signal dans le domaine fréquentiel. Ainsi, les figures 7.13 et 7.14 présentent respectivement le résultat de la transformée de Fourier discrète sur une trace brute et sur une trace filtrée. L'algorithme *FFT* (*Fast Fourier Transform*) a été utilisé pour calculer cette transformée de Fourier discrète. La figure 7.14 permet de conclure que le bruit parasite au-dessus de la fréquence de 48 MHz a bien été enlevé suite à l'application du filtre passe-bas.



Figure 7.12 : Comparaison dans le domaine temporel entre une trace brute et une trace filtrée. Le filtre FIR de type passe-bas a pour effet de *lisser* le signal.

La figure 7.15 indique le nombre de traces filtrées qu'un attaquant doit utiliser pour que l'attaque CPA révèle la valeur du quatorzième byte de la clé de chiffrement. On constate qu'au-dessus de 1300 traces, l'attaque révèle toujours la valeur de la clé secrète. On remarque également que la corrélation pour la clé secrète est plus élevée lorsque le filtrage est appliqué (valeur de 0,169 pour 10000 traces) que lorsqu'il ne l'est pas (valeur de 0,118 pour 10000 traces). Ainsi, comparativement aux résultats obtenus sans pré-traitement (voir figure 7.11), le filtrage a permis d'améliorer l'attaque. La section 7.3.3 présente le taux de succès de l'attaque selon que le pré-traitement a été appliqué ou non.



Figure 7.13 : Transformée de Fourier discrète pour une trace brute.



Figure 7.14 : Transformée de Fourier discrète pour une trace filtrée.



Figure 7.15 : Attaque CPA sur le quatorzième byte de la clé de chiffrement. Ce graphe indique qu'il faut un minimum de 1300 traces à l'attaquant pour que l'attaque CPA fonctionne tout le temps.

7.3.3 Taux de succès de l'attaque

Comme pour les simulations, nous mesurons le taux de succès de l'attaque CPA avec et sans filtrage. Le résultat est présenté à la figure 7.16. Il est ainsi mis en évidence l'avantage qu'apporte le filtrage sur les traces mesurées.

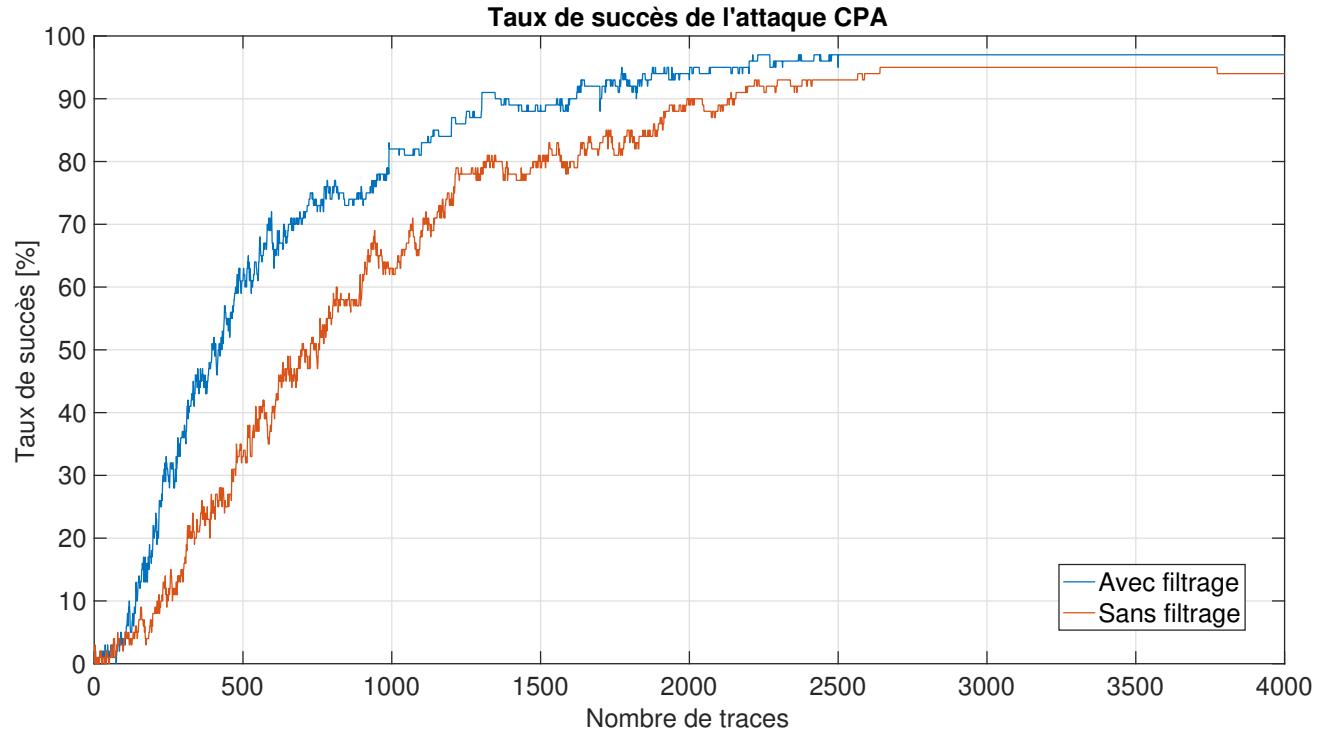


Figure 7.16 : Taux de succès de l'attaque CPA selon que les traces soient préalablement filtrées (courbe bleue) ou non (courbe orange). L'attaque CPA fonctionne mieux avec filtrage.

Chapitre 8

Implémentation de la contre-mesure de type *faking*

Ce chapitre se scinde en trois parties. Premièrement, il décrit l'implémentation de la contre-mesure *faking*. Deuxièmement, il présente les résultats de l'attaque CPA obtenus par simulation. Enfin, il présente les résultats de l'attaque CPA sur l'implémentation de la contre-mesure *faking*.

8.1 Implémentation de la contre-mesure

L'implémentation du protocole de communication entre l'ordinateur et le bloc de chiffrement reste inchangé (voir section 7.1.2). Seul le bloc de chiffrement est adapté afin d'ajouter l'implémentation de la contre-mesure *faking*. La figure 8.1 donne une représentation haut niveau du bloc de chiffrement lorsque la contre-mesure est implémentée. Seul un port est ajouté au bloc de chiffrement défini à la section 7.1.1. Il s'agit du port d'entrée *Mask*, sur 256 bits, représentant la valeur du masque utilisé pour cacher la vraie clé (voir section 5.4.1). Deux architectures différentes ont été mises en oeuvre pour la contre-mesure *faking*. Dans la première architecture visible à la figure 8.2, les opérations principales sont exécutées en parallèle. Dans la seconde architecture visible à la figure 8.3, les opérations principales sont exécutées de manière séquentielle.

Pour l'architecture en parallèle, les opérations *AddRoundKey*, *SubBytes*, *ShiftRows* et *MixColumns* s'exécutent en un seul coup de clock comme décrit à la section 7.1.1. Lorsque la contre-mesure est implémentée, quatre nouvelles opérations sont ajoutées : *SubBytesTrans*, *ShiftRowsTrans*, *MixColumnsTrans* et *Remasking* (voir section 5.4). En réalité, les opérations *ShiftRowsTrans* et *MixColumnsTrans* réalisent exactement les mêmes fonctions que les opérations *ShiftRows* et *MixColumns* respectivement. Ainsi, seules les opérations *SubBytesTrans* et *Remasking* doivent être définies (voir section 5.4). L'opération *SubBytesTrans* s'exécute en deux coups d'horloge tandis que l'opération *Remasking* s'exécute en un seul coup d'horloge. Par conséquent, cette implémentation requiert un total de X coups d'horloge.

Pour l'architecture en séquentielle, le nombre de coups d'horloge pour chaque opération est identique à celui pour l'architecture en parallèle. Ainsi, après l'opération *Keyschedule*, les 4 opérations principales de l'AES sont exécutées (*AddRoundKey*, *SubBytes*, *ShiftRows* et *MixColumns*). Ensuite, ce sont les cinq opérations définies pour la contre-mesure *faking* qui sont exécutées (*SubBytesTrans*, *ShiftRowsTrans*, *MixColumnsTrans* et *Remasking*). Cette implémentation requiert un total de X coups d'horloge.

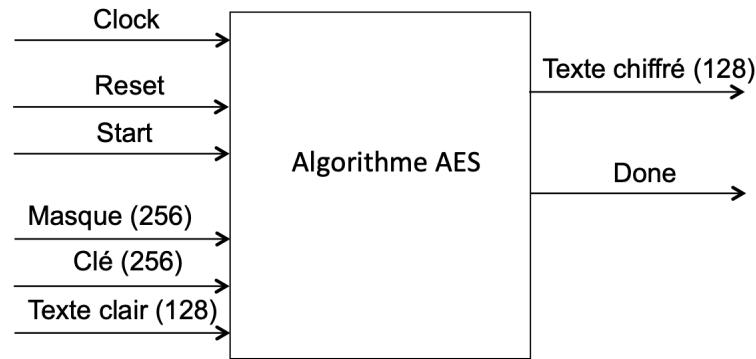


Figure 8.1 : Représentation haut niveau du bloc de chiffrement lorsque la contre-mesure *faking* est implémentée.

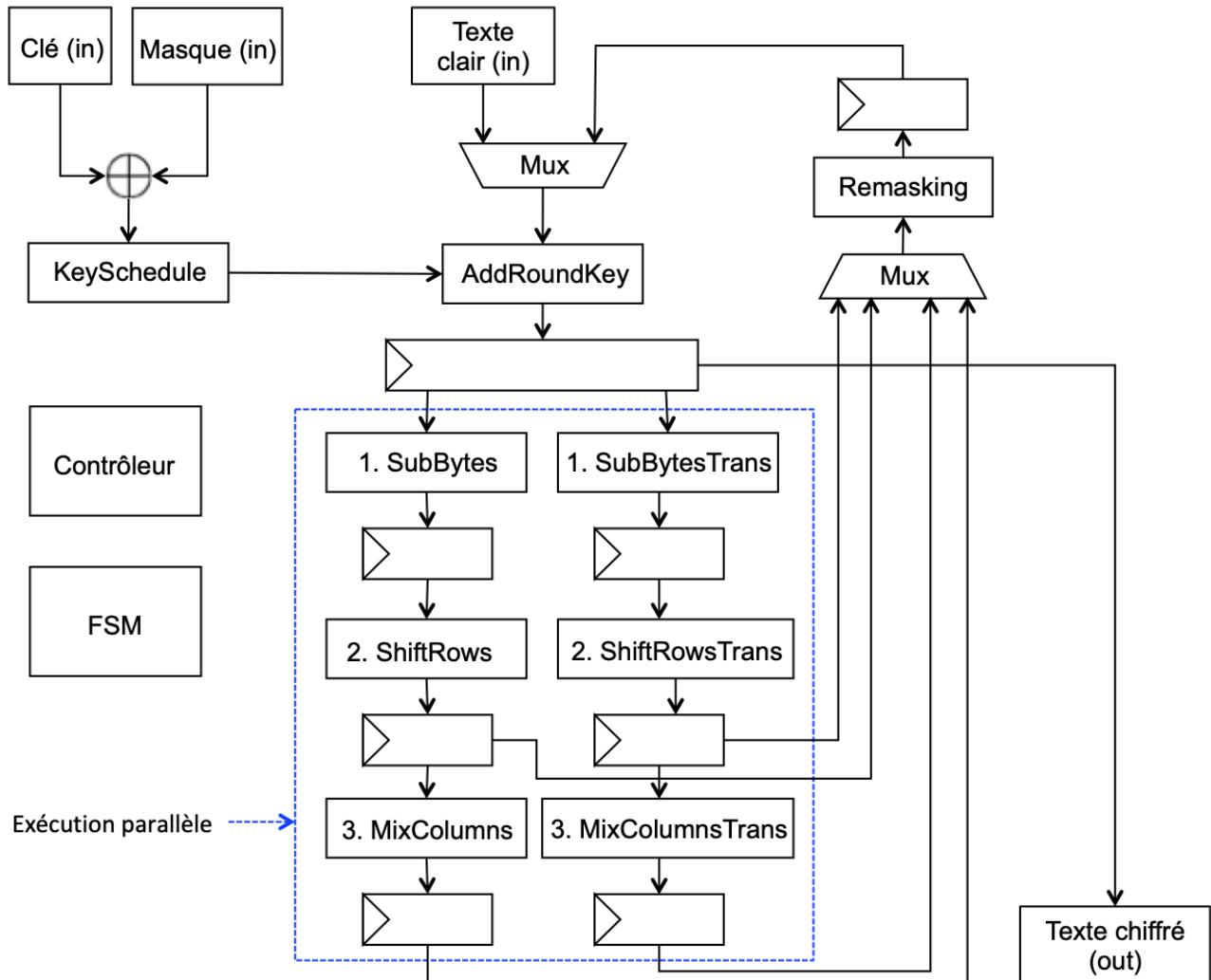


Figure 8.2 : Architecture de l'algorithme AES-256 avec la contre-mesure *faking* exécutée en mode parallèle.

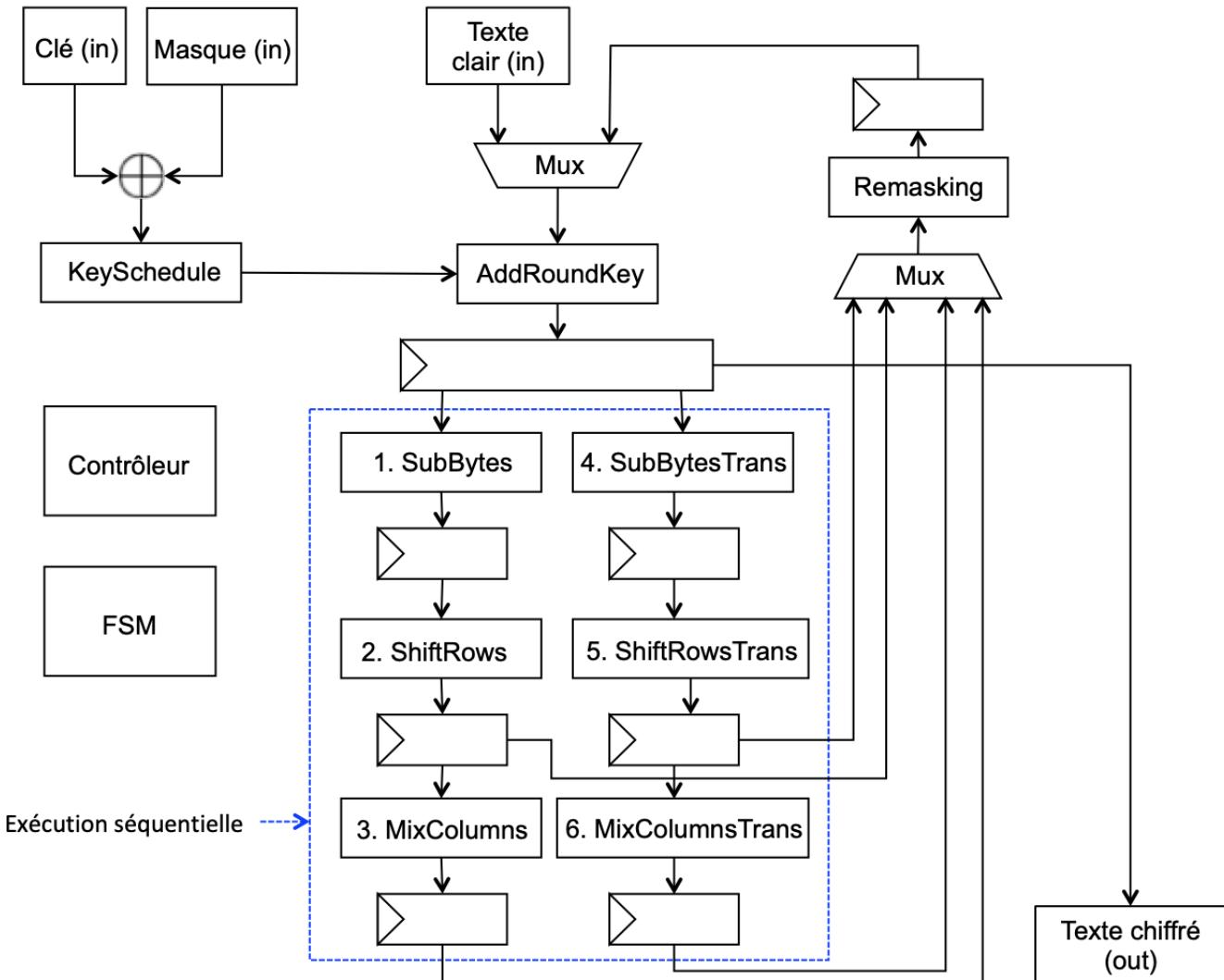


Figure 8.3 : Architecture de l'algorithme AES-256 avec la contre-mesure *faking* exécutée en mode séquentiel.

8.2 Résultats de simulations

Le bruit ayant été mis en évidence au chapitre précédent (chapitre 7 section 7.2), les traces simulées pour cet exemple sont directement additionnées d'un bruit *gaussien* de moyenne μ égale à 0 et d'écart-type σ égal à 10. La figure 8.4 présente le résultat de l'attaque CPA pour le cinquième byte de la clé de chiffrement. 2000 traces ont été utilisées pour réaliser l'attaque. La vraie clé employée pour chiffrer les données vaut 25. Sachant que le masque vaut 100, la fausse clé vaut 125 (voir sous-section 5.4.1, équation 5.7). La figure 8.5 présente, pour le même exemple, le résultat obtenu selon le nombre de traces utilisées. La clé secrète ne laisse fuiter aucune information, sa corrélation est semblable à celle des 254 autres mauvaises clés testées. La fausse clé est quant à elle directement révélée à partir de 600 traces.

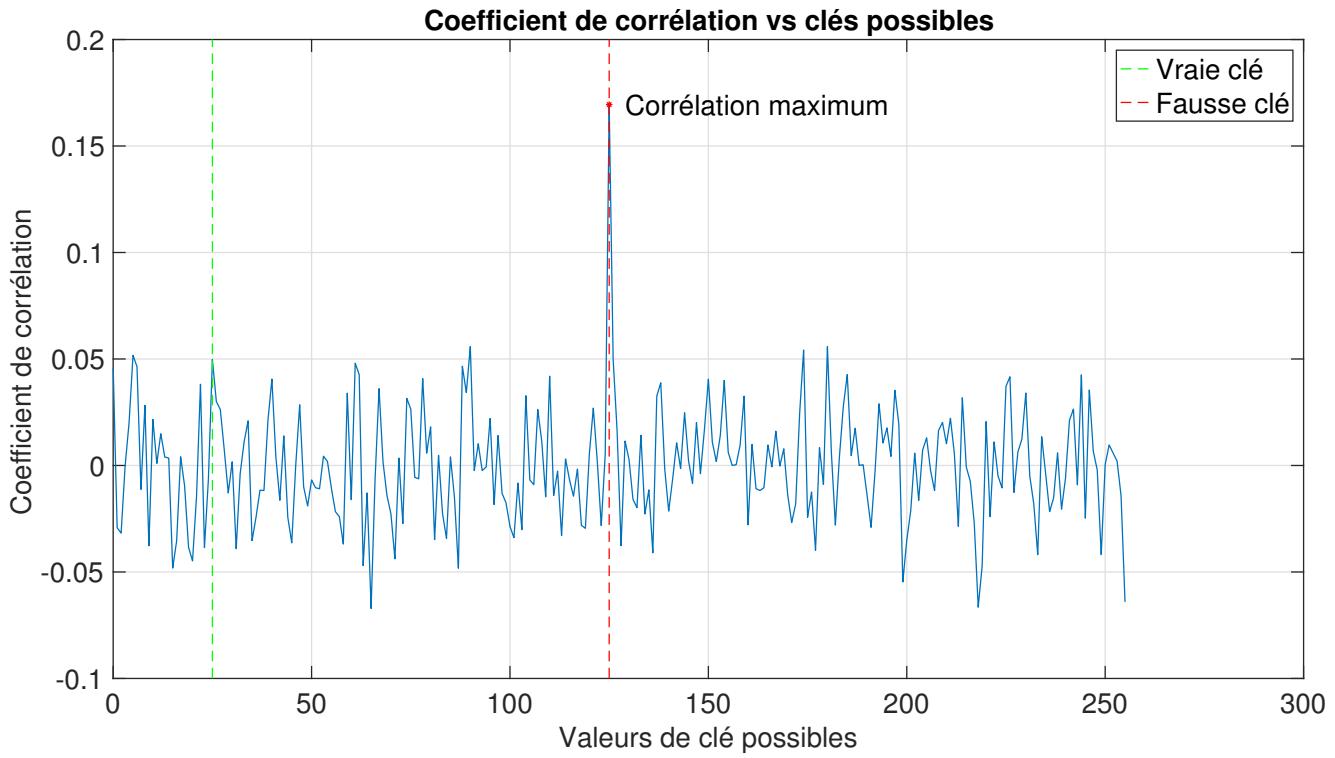


Figure 8.4 : Valeurs de corrélation pour chacune des 256 valeurs de clé testées (attaque CPA sur un byte de la clé). La fausse clé (125) est révélée tandis que le vraie clé (25) ne laisse fuiter aucune information.

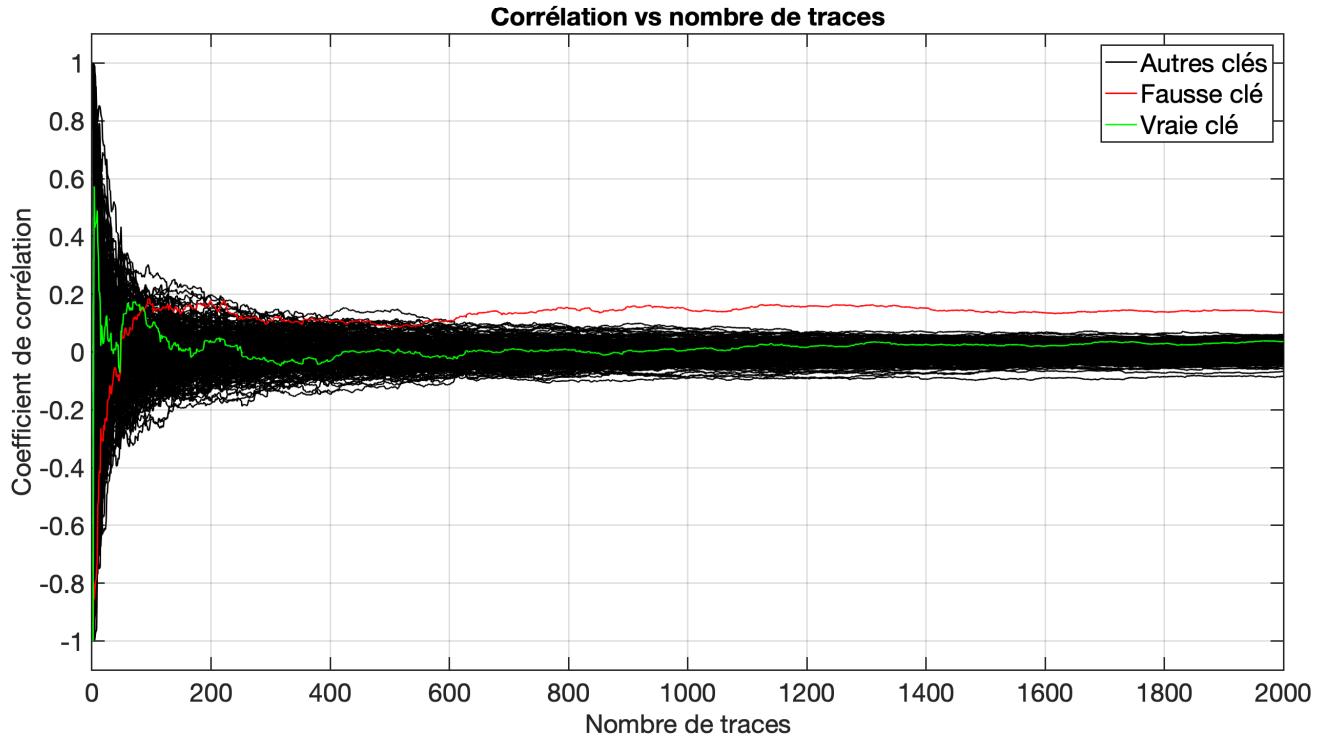


Figure 8.5 : Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La fausse clé (courbe rouge) est retrouvée à partir de 600 traces alors que le vraie clé ne laisse fuiter aucune information.

8.3 Résultats expérimentaux

Cette section présente les résultats de l'attaque CPA obtenus sur l'implémentation de la contre-mesure *faking* (avec filtrage appliqué). Plus précisément, les résultats sont scindés en deux sous-sections. La sous-section 8.3.1 présente les résultats obtenus pour l'architecture parallèle alors que le sous-section 8.3.2 présente les résultats obtenus pour l'architecture séquentielle.

8.3.1 Architecture parallèle

La figure 8.6 présente la trace obtenue lors de l'exécution de l'algorithme AES avec implémentation de la contre-mesure *faking* en mode parallèle. Comparativement à l'exécution de l'AES sans contre-mesure où la durée de chiffrement d'un texte clair prend 1500 ns (voir sous-section 7.3.1), la durée de chiffrement lorsque la contre-mesure est implantée s'élargit à 2400 ns.

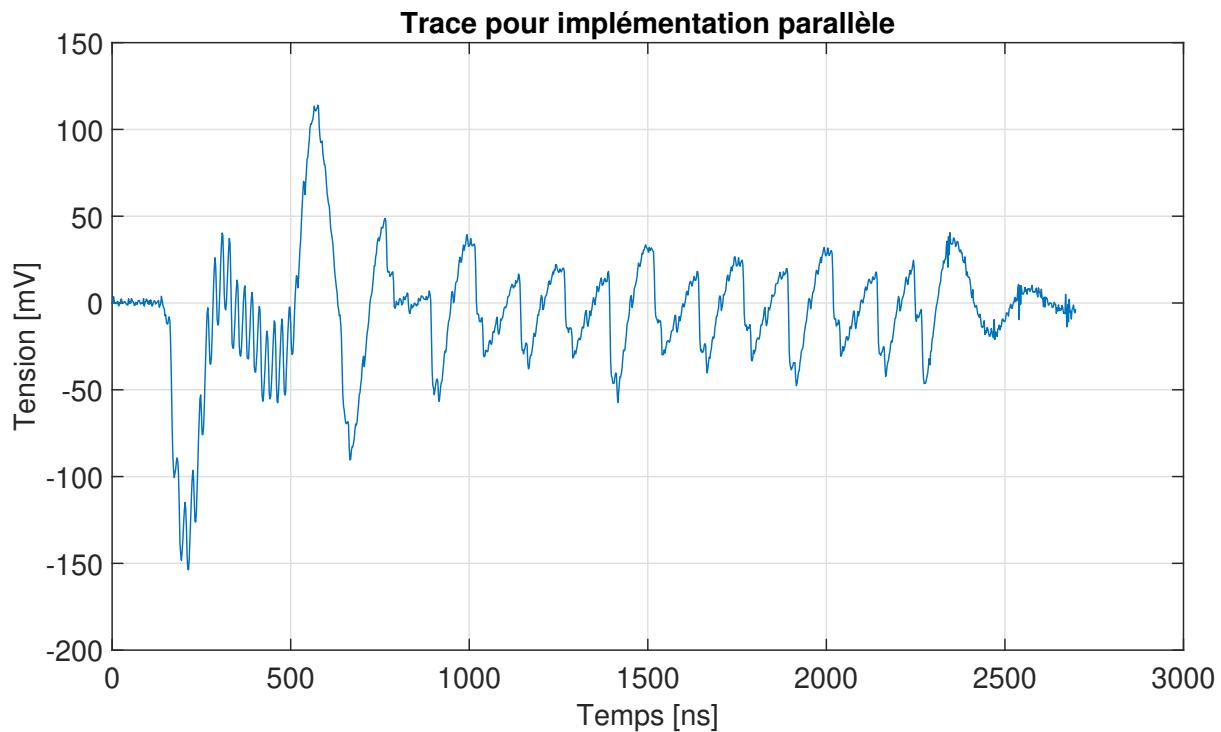


Figure 8.6 : Trace obtenue lorsque la contre-mesure *faking* est implantée de façon parallèle.

Les figures 8.7 et 8.8 présentent le résultat de l'attaque CPA pour le cinquième byte de la clé de chiffrement. La clé révélée (138) ne correspond pas à la vraie clé (4), ce qui est normal puisque la vraie clé est masquée. Cela montre bien que la contre-mesure joue correctement son rôle afin d'empêcher l'attaquant de retrouver la valeur de la clé secrète. Cependant, le résultat obtenu est inattendu. La clé retrouvée ne correspond pas non plus à la fausse clé utilisée (84). Or, en théorie, cette fausse clé devrait être retrouvée par l'attaquant. De nouvelles attaques plus ciblées ont été mises en oeuvre. Cependant, le résultat reste identique, maximum six bytes de la fausse clé (sur les 32 bytes) peuvent être retrouvés. L'une des raisons pouvant justifier le non-fonctionnement de l'attaque concerne l'exécution en parallèle des opérations. Comme détaillé à la section 5.2, il existe un type de contre-mesure *hiding*, appelé *noise engines*, dont le but est d'ajouter volontairement du bruit dans les traces afin d'empêcher l'attaquant de retrouver la clé de chiffrement. Pour ce faire, on utilise des composants *hardware* qui travaillent en parallèle de l'exécution de l'algorithme de chiffrement et dont l'objectif est de générer du bruit. Étant donné l'architecture parallèle mise en place (voir section 8.1) et le fait que seulement certains bytes de la fausse clé soient révélés, l'idée de modifier l'architecture de la contre-mesure a été établie. Voici la raison pour laquelle une architecture séquentielle de la contre-mesure *faking* a également été mise en place. Les résultats sont présentés à la sous-section 8.3.2.

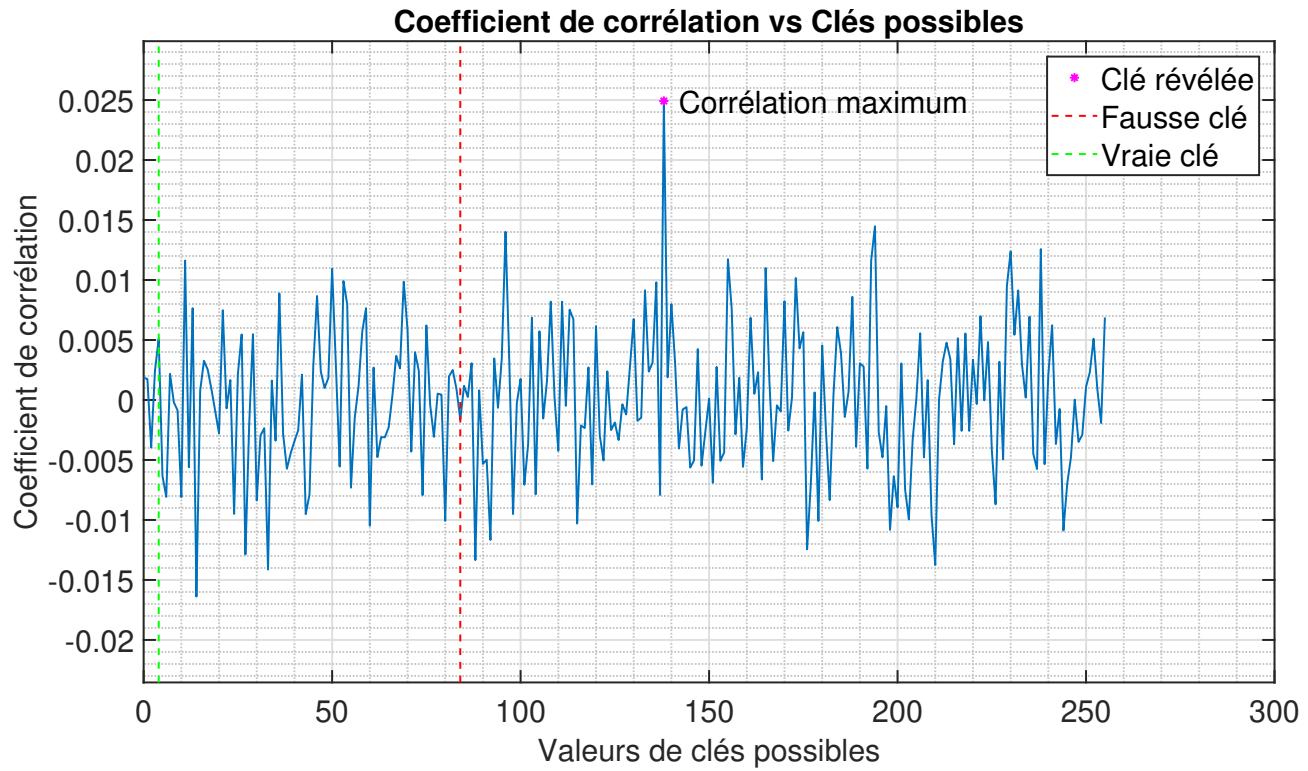


Figure 8.7 : Valeurs de corrélation pour chacune des 256 valeurs de clé testées. La clé révélée (138) ne correspond ni à la vraie clé (4) ni à la fausse clé (84).

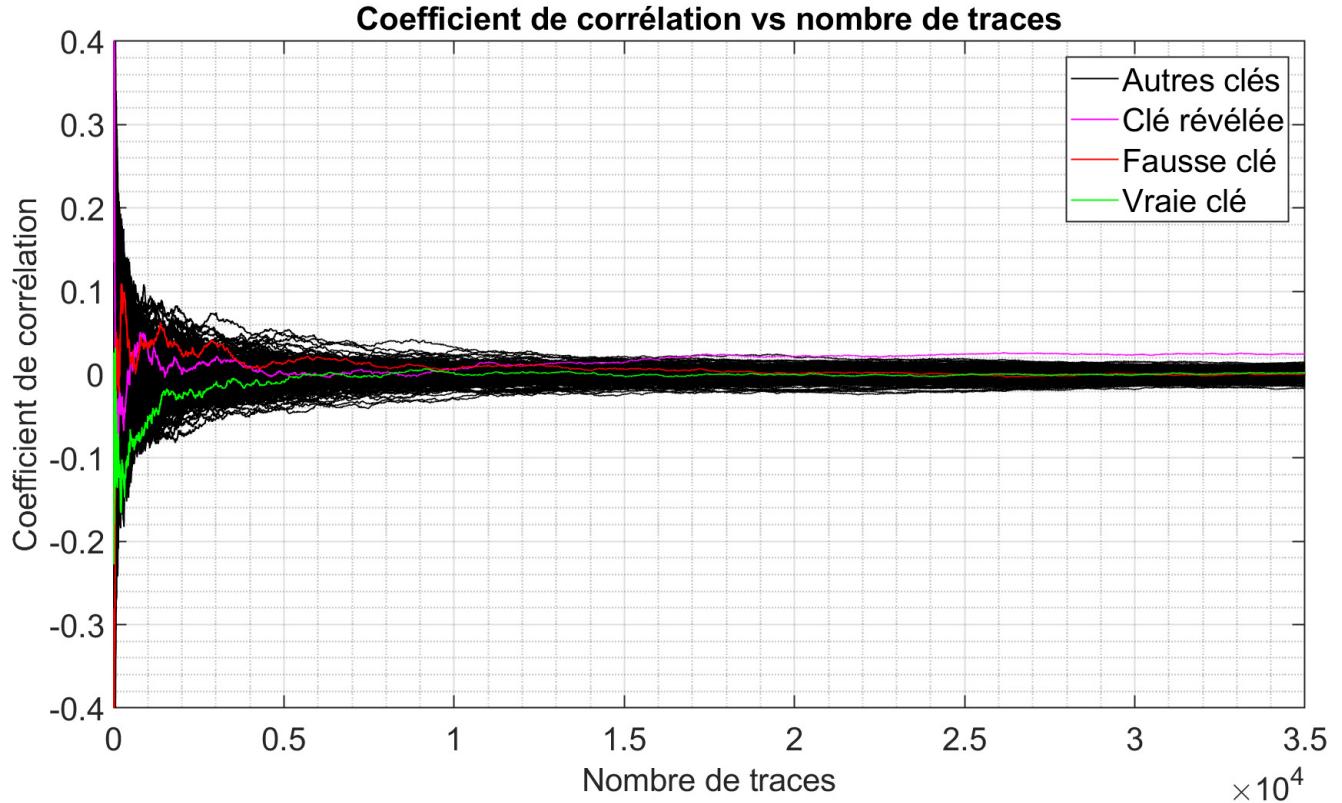


Figure 8.8 : Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La vraie clé (courbe verte) n'est pas révélée. La fausse clé (courbe rouge) n'est pas non plus révélée. La clé révélée (courbe magenta) est un résultat inattendu.

8.3.2 Architecture séquentielle

La figure 8.9 présente la trace obtenue lors de l'exécution de l'algorithme AES avec implémentation de la contre-mesure *faking* en mode séquentiel. La durée de chiffrement s'élargit ici à 3250 ns.

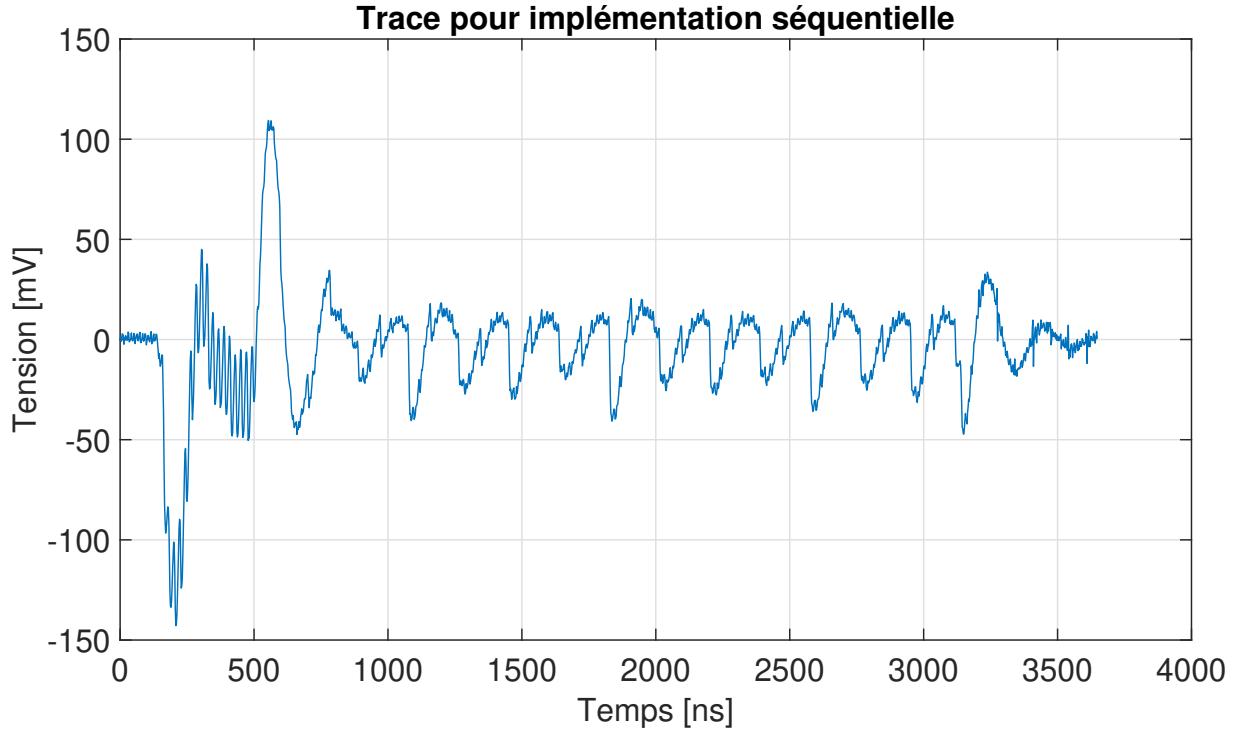


Figure 8.9 : Trace obtenue lorsque la contre-mesure *faking* est implantée de façon séquentielle.

Les figures 8.10 et 8.11 présentent le résultat de l'attaque CPA sur le cinquième byte de la clé de chiffrement. Comme pour l'architecture parallèle, la clé retrouvée (201) ne correspond pas à la vraie clé (4), ce qui est normal puisque la vraie clé est masquée. Cela montre à nouveau que la contre-mesure joue correctement son rôle afin d'empêcher l'attaquant de retrouver la valeur de la clé secrète. Cependant, le résultat obtenu n'est pas plus satisfaisant. La clé retrouvée ne correspond pas non plus à la fausse clé utilisée (84). Plus incompréhensible, après plusieurs attaques, le résultat n'est pas meilleur que celui obtenu lorsque l'architecture est configurée en parallèle : maximum six bytes de la fausse clé (sur les 32 bytes) peuvent être retrouvés.

Dans un article [] détaillant la contre-mesure *faking*, il est précisé que les opérations exécutées spécifiquement pour la contre-mesure *faking* (*SubBytesTrans*, *ShiftRowsTrans*, *MixColumnsTrans* et *Remasking*) sont implémentées sur un second FPGA. Par manque de temps, je n'ai pu implémenter cette contre-mesure sur le second FPGA. Cela aurait demandé de modifier l'implémentation du contrôleur et vu le temps imparié, cela n'aurait pas été envisageable. Cependant, il s'agit d'une piste d'amélioration à ne pas négliger. Une question reste cependant en suspens : *pourquoi retrouve-t-on dans 75% des attaques CPA réalisées, quatre bytes de la fausse clé de chiffrement ?* Sur base de cette question, deux pistes peuvent être envisagées :

- Mieux cibler l'attaque : lorsque l'attaque CPA est réalisée, elle se concentre sur un certain nombre d'échantillons dans la trace. La mise en oeuvre d'une métrique permettant de déduire dans les traces, les échantillons à cibler pour améliorer le taux de succès de l'attaque semble être une première piste afin de savoir où l'information peut être retrouvée.
- Utilisation d'un filtre performant : certes, un filtre *FIR* passe-bas a été mis en place et permet d'obtenir de meilleurs résultats (voir sous-section 7.3.2). La bande passante après application de ce filtre s'étend de 1 Hz à 50 MHz. Cette bande passante peut être analysée afin de savoir sur quelles fréquences les informations relatives à la clé secrète peuvent être révélées. En effet, les informations relatives à la clé sont révélées à basse fréquence.

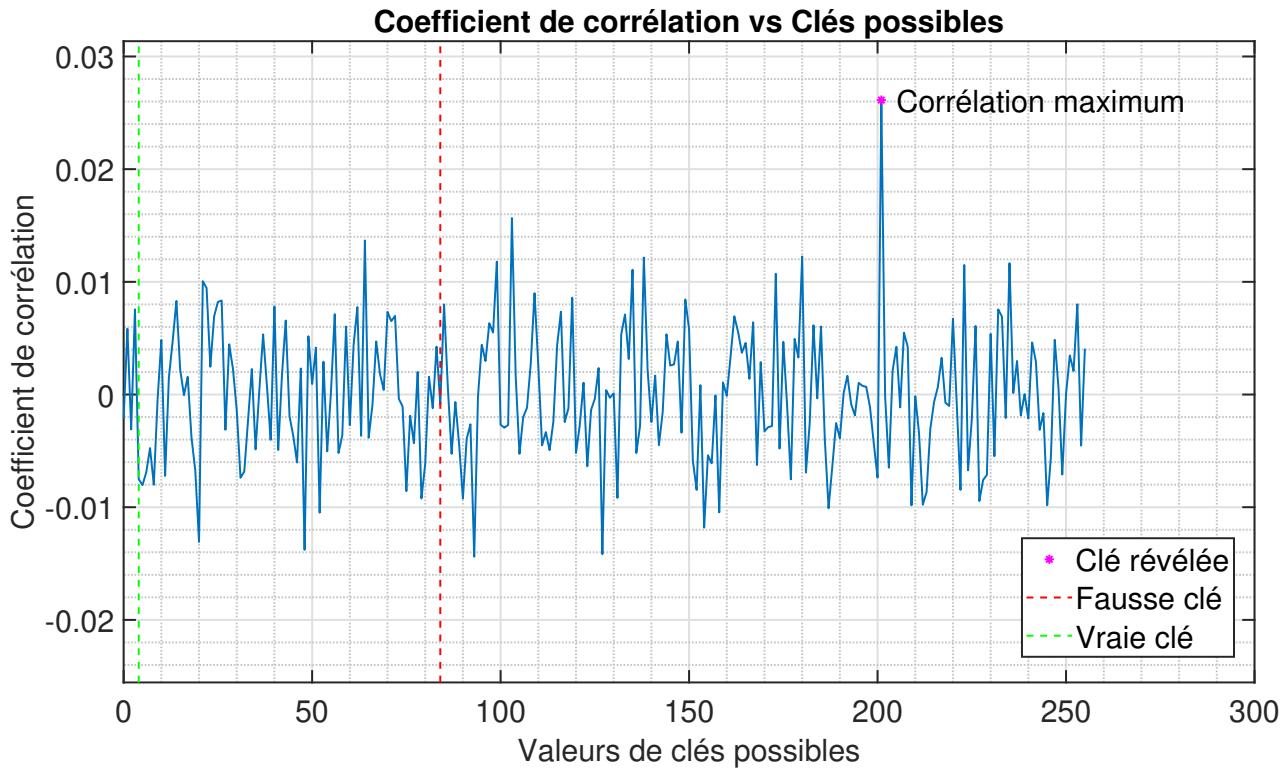


Figure 8.10 : Valeurs de corrélation pour chacune des 256 valeurs de clé testées. La clé révélée (201) ne correspond ni à la vraie clé (4) ni à la fausse clé (84).

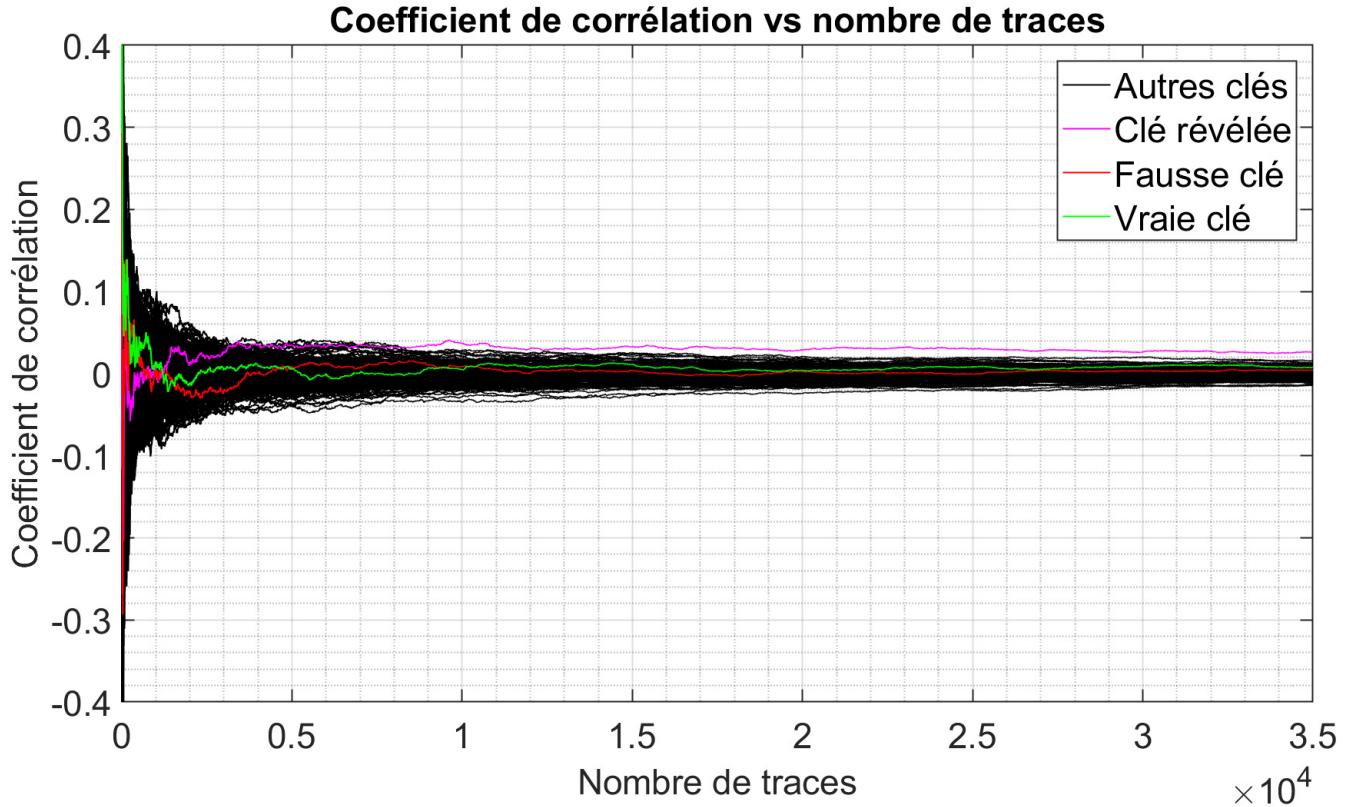


Figure 8.11 : Valeurs de corrélation pour les 256 clés testées en fonction du nombre de traces mesurées. La vraie clé (courbe verte) n'est pas révélée. La fausse clé (courbe rouge) n'est pas non plus révélée. La clé révélée (courbe magenta) est un résultat inattendu.

Chapitre 9

Évaluation de la contre-mesure

Ce dernier chapitre se scinde en deux parties. Dans un premier temps, il décrit les moyens mis en place pour évaluer et valider le fonctionnement de l'algorithme AES lorsque la contre-mesure *faking* est implémentée. Ensuite, il compare les performances de l'AES avec et sans contre-mesure.

9.1 Simulations (*test benches*)

Certes, la contre-mesure implémentée ne fonctionne pas comme décrit en théorie lorsqu'on applique une attaque CPA (voir section 5.4) : l'attaque ne révèle pas la fausse clé utilisée. Cependant, cette contre-mesure constitue tout de même une réussite étant donné qu'elle ne révèle pas le vraie clé de chiffrement. L'attaquant est donc incapable de déchiffrer les données confidentielles à partir de la clé révélée par l'attaque. Par ailleurs, la contre-mesure permet de chiffrer correctement les textes clairs. Pour s'assurer au préalable de son bon fonctionnement, une série de tests ont été mis en place et sont décrits ci-après.

Avant d'écrire le code VHDL nécessaire pour implémenter cette contre-mesure, une simulation à l'aide du logiciel MATLAB a été réalisée. Cette simulation a permis de bien comprendre l'utilité de chaque opération dans l'exécution de la contre-mesure. Une fois la simulation maîtrisée, la mise en place de la contre-mesure a pu être élaborée.

En langage de programmation *hardware*, il est de coutume de réaliser plusieurs tests (*test benches* en anglais) sur les nouveaux modules *hardware* définis. Ceci, dans le but de valider leur bon fonctionnement avant leur synthèse et leur implémentation sur le FPGA. Dans ce travail, outre la vérification et la validation des modules un à un via de simples *test benches*, un test automatisé général a été mis en place afin de certifier que les données soient correctement chiffrées. Un tel environnement de tests est complexe à mettre en oeuvre et demande des connaissances et du temps pour être complet. Cependant, une fois terminé, ce test automatisé permet de valider rapidement et aisément le bon fonctionnement du module en réalisant une centaine de tests. Ceci a l'avantage d'amener plus de garantie quant au bon fonctionnement du code écrit (un test vs une centaine de tests). En effet, avec un simple *test bench*, la vérification n'est pas toujours aisée : d'une part, il faut aller vérifier les réponses obtenues pour chaque signaux analysés sur un graphe (ce qui prend du temps) et d'autre part, la certitude d'un fonctionnement sans défaut du module n'est pas garantie (étant donné que seul un test est réalisé). Dès lors, la mise en place d'un test automatisé permettant de traiter une centaines d'épreuves différentes s'est révélé être un outil intéressant et très pratique. La figure 9.1 présente le principe de fonctionnement de ce test automatisé.

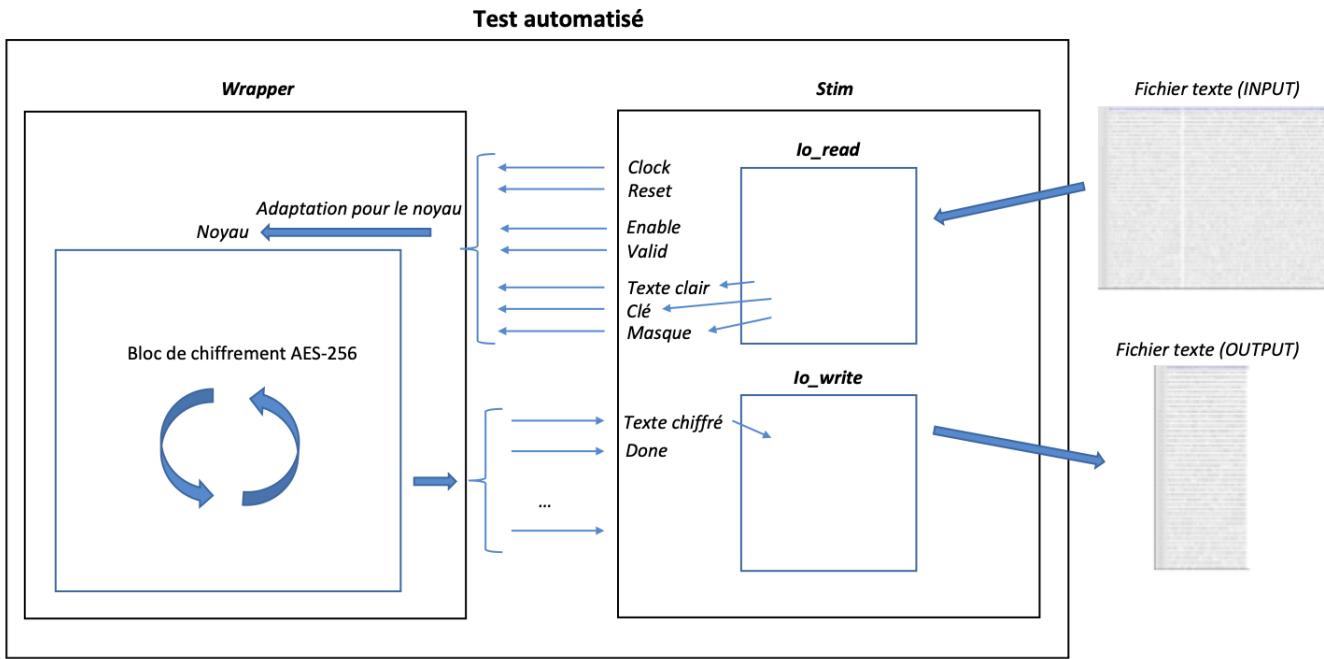


Figure 9.1 : Schéma-bloc du principe de fonctionnement du test automatisé mis en place.

Le test automatisé est généré à partir d'une invite de commande. Plus précisément, c'est un script python qui est en charge de lancer le test automatisé selon un paramètre que l'on écrit à la suite du fichier python appelé. Pour lancer le test, il suffit de taper la commande : "python flow.py sim". "python flow.py" indique que l'on souhaite lancer un script python, nommé *flow.py*. Le paramètre "*sim*" indique que l'on souhaite lancer le test automatisé. Ce test est évidemment écrit en langage VHDL afin de pouvoir tester les modules du bloc de chiffrement AES et de la contre-mesure *faking*. Plus précisément, ce test se compose de deux grands modules (voir figure 9.1) :

- Module *Stim* : ce module a pour objectif d'une part d'envoyer des stimulus au bloc de chiffrement et d'autre part de réceptionner les sorties de ce bloc de chiffrement, fonction des stimulus envoyés. Ce module *Stim* est lui-même composée de deux sous-modules :
 - Sous-module *Io_read* : l'objectif de ce sous-module est d'aller chercher dans un fichier texte les informations utiles aux ports d'entrées du bloc de chiffrement. Ces informations utiles sont les textes clairs, la clé et le masque (lorsque la contre-mesure est mise en place).
 - Sous-module *Io_write* : l'objectif de ce sous-module est de réceptionner les informations utiles aux ports de sorties du bloc de chiffrement. Ces informations utiles sont les textes chiffrés. Une fois réceptionnées, ces informations sont écrites dans un fichier texte.
- Module *Wrapper* : ce module a pour objectif d'adapter (largeur de bus, fréquence, etc.) les signaux envoyés du module *Stim* afin qu'ils puissent être interprétés par le noyau. Le noyau représente le module à tester. Pour ce travail, le noyau représente dans un premier temps le bloc de chiffrement AES-256 et dans un second temps le bloc de chiffrement AES-256 avec la contre-mesure *faking*.

Ainsi, vu de l'extérieur, ce test automatisé contient un port d'entrée et un port de sortie. Le port d'entrée prend un fichier texte contenant toutes les épreuves, c'est-à-dire contenant différents messages clairs, une clé et une masque. Le port de sortie renvoie un fichier texte avec les réponses aux différentes épreuves, c'est-à-dire contenant les messages chiffrés. Enfin, une fois le fichier texte de sortie généré, un script python compare les résultats obtenus dans ce fichier texte avec les vrais résultats attendus (enregistrés préalablement dans un autre fichier texte). Si une erreur est apparue lors du chiffrement d'un message clair, le test automatisé la renseignera directement. De cette façon, une centaine d'épreuves sont utilisées pour valider le bon fonctionnement de l'AES-256 lorsque la contre-mesure *faking* est implémentée.

Par ailleurs, outre lancer le test automatisé, le script python *flow.py* permet également d'afficher sur le logiciel VIVADO les graphes des différents signaux générés. Les figures 9.2 et 9.3 présentent respectivement

quelques signaux analysés lorsque la contre-mesure est implémentée en mode parallèle et en mode séquentiel. Pour ce faire, il suffit de taper la commande "`python flow.py wave`". Le paramètre "`wave`" indique simplement que l'on souhaite afficher les signaux sur un graphe. Ceci est utile pour comprendre le fonctionnement du code VHDL ou bien pour trouver la source d'une éventuelle erreur. Durant ce travail, lorsqu'un erreur était détectée par le test automatisé, la génération des signaux était couramment utilisée pour cibler les erreurs. Au final, toutes les erreurs ont été débogées, ce qui a permis d'obtenir une version fonctionnelle de la contre-mesure. De cette façon, le *bitstream* a ainsi pu être généré et le programme chargé sur le FPGA. Aucune erreur de chiffrement n'a été détectée lors de l'utilisation du FPGA pour chiffrer des textes clairs. La contre-mesure est donc bien fonctionnelle.

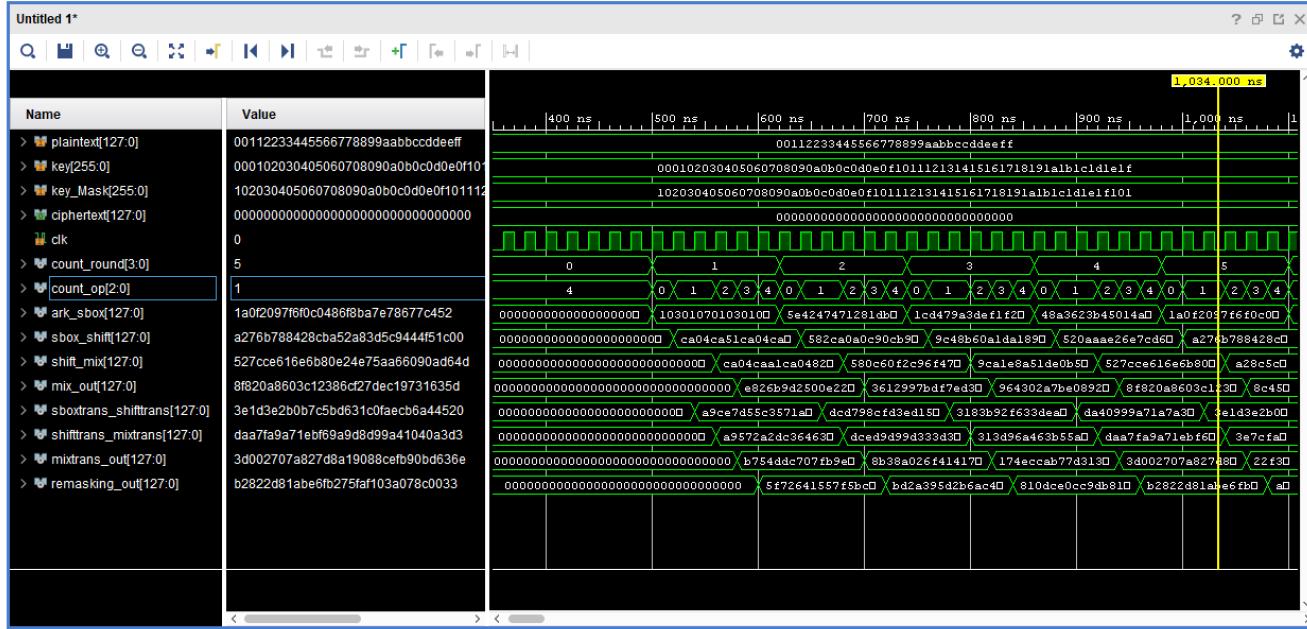


Figure 9.2 : Affichage via la logiciel VIVADO des différents ports/signaux utilisés dans l'implémentation de la contre-mesure en mode parallèle. Le graphe montre bien que les opérations principales sont exécutées en parallèle.

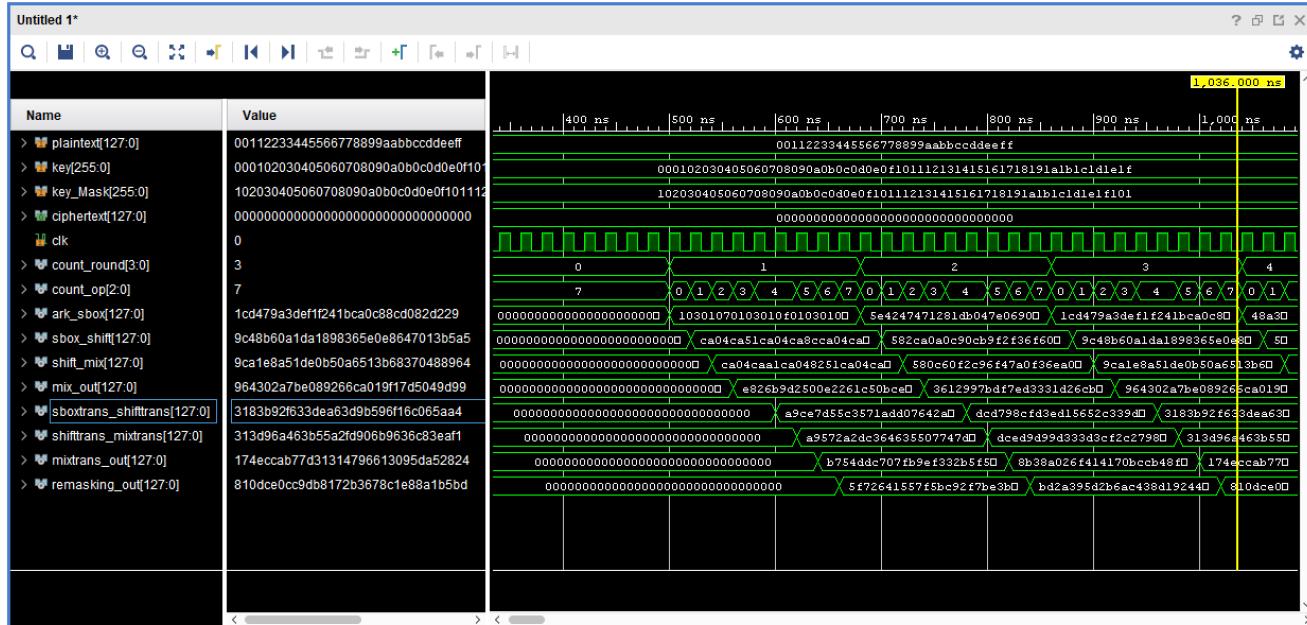


Figure 9.3 : Affichage via la logiciel VIVADO des différents ports/signaux utilisés dans l'implémentation de la contre-mesure en mode séquentiel. Le graphe montre bien que les opérations sont exécutées en séquentiel.

9.2 Évaluation par critères de performances

Une fois les tests correctement validés, l’élaboration de la synthèse du code VHDL via un logiciel (*ISE Design Suite*) peut être lancée. Cette étape de synthèse terminée, les étapes de placement et de routage peuvent ensuite être exécutées. Enfin, la génération d’un fichier *Bitstream* clôturera le flot de conception et permettra de programmer le FPGA selon les spécifications demandées. Dans cette section, une comparaison des performances de l’AES sans et avec contre-mesure est abordée. Le tableau 9.1 présente les principales caractéristiques analysées et discutées pour cette section. Les annexes D.1, D.2 et D.3 reprennent respectivement les rapports complets de *design* pour l’implémentation de l’AES tout seul, l’implémentation de la contre-mesure *faking* en mode parallèle et l’implémentation de la contre-mesure *faking* en mode séquentiel.

Afin de bien comprendre les caractéristiques reprises dans le tableau 9.1, une description des principaux concepts *hardware* est définie ci-dessous. Par définition, un FPGA est un circuit logique programmable. Cela signifie que le circuit intégré peut être reprogrammé après sa fabrication. La figure 9.4 présente un échantillon très simplifié et généraliste de l’architecture d’un FPGA. Plus précisément, cette figure présente la relation entre les blocs d’entrée/sortie (Input/Output Block - IOB), les blocs logiques (Configurable Logic Block - CLB) et les blocs de mémoire (Block RAM - BRAM).

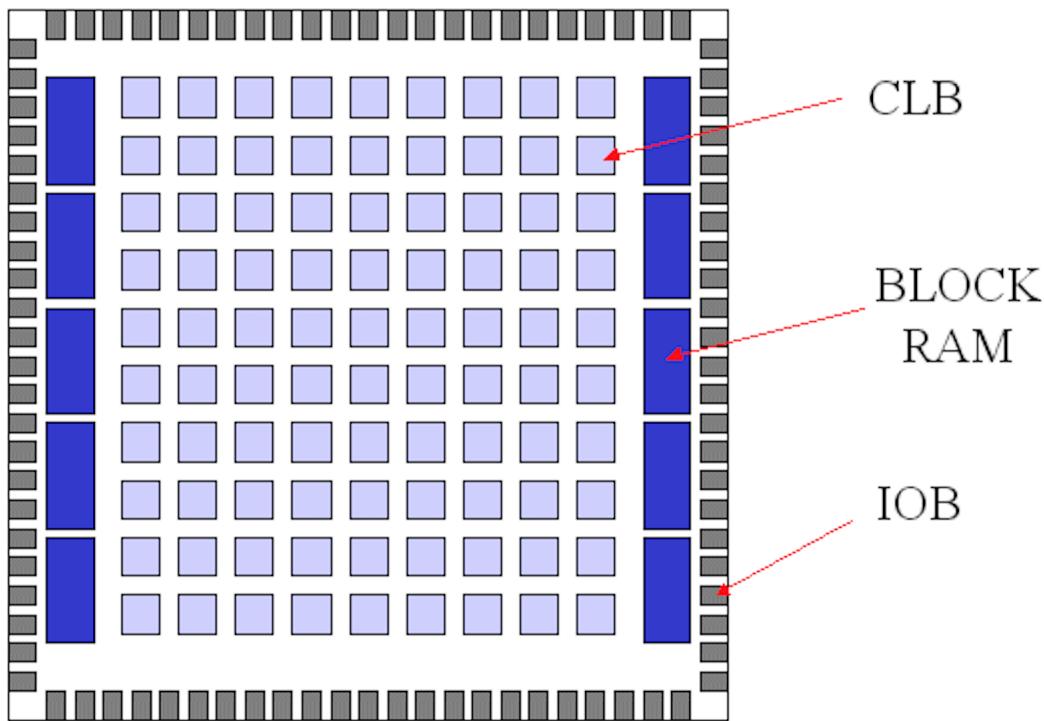


Figure 9.4 : Architecture simpliste d’un FPGA.

Le bloc logique est constitué de deux éléments principaux : des Look-Up Tables (LUT’s) et des Flip-Flops. Les LUT’s servent à implémenter des fonctions logiques (OR, AND, XOR, etc.) et les flip-flops représentent des registres d’un bit permettant le stockage d’une information binaire. Ces deux éléments simples sont regroupés dans ce qu’on appelle un *slice*. Pour le FPGA utilisé dans ce travail (famille Xilinx Spartan 6), deux slices forment un bloc logique. À partir d’un bloc logique, un développeur d’application FPGA est capable de générer n’importe quel circuit numérique, moyennant l’utilisation d’une suite d’outils dits « de synthèse ». Afin d’améliorer les capacités des FPGA’s, les constructeurs ont toutefois ajouté d’autres éléments en plus des blocs logiques comme par exemple de la mémoire RAM. On retrouve effectivement des blocs contenant de la mémoire à accès direct (BRAM) facilitant le stockage de grandes quantités d’information. Enfin, tout autour de ces CLB et de ces BRAM, on trouve des blocs d’entrées/sorties (IOB) dont le rôle est de gérer les entrées-sorties réalisant l’interface avec les modules extérieurs au FPGA.

	Slices	LUT's (Look Up Table)	Flip-Flops (Registre)	BRAM (9 Kb)	IOB (In/Out Block)	Temps d'exécutions (ns)
AES seul	1473 (12%)	2742 (5%)	3526 (3%)	8 (2%)	...	1586
AES avec contre-mesure <i>faking</i> en mode parallèle	2822 (24%)	5493 (11%)	6734 (7%)	8 (2%)		2392
AES avec contre-mesure <i>faking</i> en mode séquentiel	2741 (23%)	4970 (10%)	6609 (7%)	16 (4%)		3192

Table 9.1 : Tableau récapitulatif des performances observées pour les implémentations de l'AES seul, de l'AES avec contre-mesure *faking* en mode parallèle et de l'AES avec contre-mesure *faking* en mode séquentiel.

Sur base du tableau 9.1, nous constatons que :

- L'ajout de la contre-mesure (dans les deux modes) double le nombre de LUT's et de flip-flops configurés.
- La contre-mesure implantée en mode parallèle conserve la même taille mémoire (BRAM) que celle utilisée pour l'AES seul.
- La contre-mesure implantée en mode séquentiel double la taille mémoire (BRAM) que celle utilisée pour l'AES seul.
- La contre-mesure implantée en mode parallèle augmente le temps d'exécution à 2392 ns.
- La contre-mesure implantée en mode séquentiel augmente le temps d'exécution à 3192 ns.

Chapitre 10

Conclusion

Ancienne conclusion ... (stage)

Nous vivons dans une société de consommation où les technologies évoluent sans cesse. Il y a 40 ans, la gestion des signaux relevait de la science-fiction. Il y a 25 ans, les GSM constituaient une innovation majeure. Aujourd’hui, tout le monde se demande comment il était possible de vivre à cette époque sans toutes ces nouvelles technologies ! Le sujet qu’il m’a été proposé d’étudier durant ces 6 semaines de stage en immersion fut, à première vue, aussi complexe qu’inconnu. En effet, ce sujet, les attaques par canaux cachés, est encore assez méconnu à ce jour dans le domaine de l’ingénierie. Tenter d’expliquer à quiconque qu’il est possible de subtiliser des données sensibles précautionneusement chiffrées par divers algorithmes, à partir de moyens simples (oscilloscope, ordinateurs, ...) et de calculs élémentaires, semble au premier abord, difficilement concevable.

Depuis ces découvertes à la fin des années 90, la sécurité matérielle a pris une tournure particulière pour les grandes industries telles que Thales. C’est désormais sur un nouvel axe de recherche, s’écartant des sentiers traditionnels, que se porte la problématique de protection de données sensibles. Imaginez qu’un algorithme utilisé pour chiffrer des données bancaires puisse être cassé par simple application d’une attaque par canaux cachés ? Il n’est donc pas anormal de voir certaines industries développer leurs propres contre-mesures pour s’affranchir contre ce type d’attaque. Certes, avant de développer des contre-mesures, il est nécessaire, dans un premier temps, de bien comprendre les fondements des attaques par canaux cachés. Certes, la compréhension des notions théoriques et leurs mises en pratique est une tâche ardue qui nécessite la compréhension et l’intégration de concepts très spécifiques. Néanmoins, je suis convaincu que ce type d’attaque va, au fil des années, devenir une piste d’enseignement sérieuse et nécessaire aux futurs ingénieurs en électronique.

À titre personnel, je tire un bilan très positif de ce stage d’immersion. Il fut très enrichissant. Ces 6 semaines de stage sont passées à la vitesse de l’éclair : plongeant à certains moments dans les articles scientifiques et livres de références rédigés sur le sujet ; à d’autres dans les simulations MATLAB afin de mieux assimiler les concepts ; ou encore à discuter avec les étudiants stagiaires afin de partager notre expérience de stage. 6 semaines, c’est assez court mais déjà suffisant pour une première introduction sur les attaques par canaux cachés et les contre-mesures qu’il est possible d’implémenter. L’ensemble des objectifs fixés avant le début de stage (énoncés à la section ??) a pu être traité et atteint. Étant désormais convaincu de la puissance de ce type d’attaque, développer des contre-mesures me paraît constituer un challenge intéressant et très utile dans le cadre de la protection des données sensibles. Je vais donc désormais pouvoir m’attaquer à la phase suivante, consécutive à ce stage, mon Travail de Fin d’Étude. Ce TFE consistera à développer une contre-mesure contre les attaques par canaux cachés.

Crédits

- Figure ?? provenant du site internet : <http://monipag.com/victoria-petitier/wp-content/uploads/sites/1363/Thales-Group-1.png>
- Figure 3.1 provenant du site internet : <https://hal.archives-ouvertes.fr/hal-00753215/document>
- Annexe A.4 provenant du site internet de J.M. Dutertre "*Synthèse AES 128*" : https://www.emse.fr/~dutertre/documents/synth_AES128.pdf
- Annexe ?? provenant du site internet UPCommons "*Faking Countermeasure Against Side-Channel Attacks*" : https://upcommons.upc.edu/bitstream/handle/2117/112973/Advances_in_Microelectronics_V_1_chapter19.pdf?sequence=1

Bibliographie

- [1] Thomas Popp Stefan Mangard, Elisabeth Oswald. *Power Analysis Attacks*. Springer, 2007.
- [2] Sri Parameswaran Jude Ambrose, Alexandar Ignjatovic. *Power Analysis Side Channel Attacks*. VDM Verlag Dr. Müller, 2010.
- [3] Eric Peeters. *Advanced DPA Theory and Practise*. Springer, Septembre 2012.
- [4] J.M. Dutertre. *Synthèse AES 128*. https://www.emse.fr/~dutertre/documents/synth_AES128.pdf, 2011.
- [5] Lilian BOSSUET. *Approche didactique pour l'enseignement de l'attaque DPA ciblant l'algorithme de chiffrement AES*. <https://hal.archives-ouvertes.fr/hal-00753215/document>, 25 Octobre 2012.
- [6] Stephane Fernandes Medeiros. *Attaques par canaux auxiliaires : nouvelles attaques, contre-mesures et mises en oeuvre*. <https://dipot.ulb.ac.be/dspace/bitstream/.../12a25345-b54f-4169-9c32-e7d2cce5af65.txt>, 2017.
- [7] François Durvaux and François-Xavier Standaert. From improved leakage detection to the detection of points of interests in leakage traces. Cryptology ePrint Archive, Report 2015/536, 2015. <https://eprint.iacr.org/2015/536>.
- [8] Francois Durvaux Francois-Xavier Standaert A. Adam Ding, Liwei Zhang and Yunsi Fei. Towards sound and optimal leakage detection procedure. In the proceedings of CARDIS 2017, Lecture Notes in Computer Science, November 2017. <https://perso.uclouvain.be/fstandae/PUBLIS/196.pdf>.
- [9] MICHAL VARCHOLA MAREK REPKA, JOZEF TOMECEK. Correlation hamming distance power analysis of 16-bit integer multiplier in fpga. <http://www.wseas.us/e-library/conferences/2014/Istanbul/TELEDU/TELEDU-06.pdf>, 2014.
- [10] Mariano López-García Rubén Lumbiarres-López and Enrique Cantó-Navarro. Faking countermeasure against side-channel attacks. https://upcommons.upc.edu/bitstream/handle/2117/112973/Advances_in_Microelectronics_V_1_chapter19.pdf;jsessionid=62B833C8C8C140B9EE46F0279D3E0B61?sequence=1, Décembre 2017.
- [11] Douglas Carson Owen Lo, William J. Buchanan. Power analysis attacks on the aes-128 s-box using differential power analysis (dpa) and correlation power analysis (cpa). <https://www.tandfonline.com/doi/pdf/10.1080/23742917.2016.1231523>, Septembre 2016.
- [12] Mohsen Machhou Rached Tourki Hassen Mestiri, Noura Benhadjyoussef. A comparative study of power consumption models for cpa attack. <http://www.mecs-press.net/ijcnis/ijcnis-v5-n3/IJCNIS-V5-N3-3.pdf>, 2013.
- [13] Correlation power analysis. https://wiki.newae.com/Correlation_Power_Analysis.
- [14] Abdelaziz Elaabid. *Attaques par canaux cachés : expérimentations avancées sur les attaques template*. <https://tel.archives-ouvertes.fr/tel-00937136/document>, 2014.
- [15] 6.1.2 *Distance de Hamming et poids d'un mot de code*. https://icwww.epfl.ch/~chappeli/it/courseFR/I2subsec_Hdist.php.
- [16] Jean-Philippe Muller. *Le bruit dans les systèmes électroniques*. <http://www.ta-formation.com/acrobat-cours/bruit.pdf>, Juillet 2002.

- [17] Renaud Dumont. *Cryptographie et Sécurité informatique INFO0045-2*. <http://www.montefiore.ulg.ac.be/~dumont/pdf/crypto09-10.pdf>, 2010.
- [18] Thales Group. *Historique*. <https://www.thalesgroup.com/fr/global/groupe/historique>, 2018.
- [19] Thales. *Wikipedia*. <http://fr.wikipedia.org/w/index.php?title=Thales&action=history>, Octobre 2018.

Annexe **A**

Liste des annexes du chapitre 2

- Annexe [A.1](#) : Chiffre de César.
- Annexe [A.2](#) : Code algorithme AES-128 - Ordre d'exécution des opérations.
- Annexe [A.3](#) : Code algorithme AES-256 - Ordre d'exécution des opérations.
- Annexe [A.4](#) : Sbox - Table de substitution.
- Annexe [A.5](#) : Opération *KeySchedule*.

A.1 Chiffre de César

Texte clair	Texte chiffré
A	D
B	E
C	F
D	G
E	H
F	I
G	J
H	K
I	L
J	M
K	N
L	O
M	P
N	Q
O	R
P	S
Q	T
R	U
S	V
T	W
U	X
V	Y
W	Z
X	A
Y	B
Z	C

Table A.1 : Chiffre de César.

Exemple : le message clair "*chiffrement*" devient le message chiffré "*fkluiuhphqw*". En effet, le détail du procédé de chiffrement est donné à la table A.2.

Message clair	C	H	I	F	F	R	E	M	E	N	T
Message chiffré	F	K	L	I	I	U	H	P	H	Q	W

Table A.2 : Exemple de chiffre de César.

A.2 Code algorithme AES-128 - Ordre d'exécution des opérations.

```
Require : STATE, Key
Ensure : STATE

KeySchedule (Key)
AddRoundKey (STATE, ExpandedKey [ 0 ] )
for i=1 < 10 do
    SubBytes (STATE)
    ShiftRows (STATE)
    MixColumns (STATE)
    AddRoundKey (STATE, ExpandedKey [ i ] )
    i = i + 1
end for
SubBytes (STATE)
ShiftRows (STATE)
AddRoundKey (STATE, ExpandedKey [ 10 ] )
```

A.3 Code algorithme AES-256 - Ordre d'exécution des opérations.

```
Require : STATE, Key
Ensure : STATE

KeySchedule (Key)
AddRoundKey (STATE, ExpandedKey [ 0 ] )
for i=1 < 14 do
    SubBytes (STATE)
    ShiftRows (STATE)
    MixColumns (STATE)
    AddRoundKey (STATE, ExpandedKey [ i ] )
    i = i + 1
end for
SubBytes (STATE)
ShiftRows (STATE)
AddRoundKey (STATE, ExpandedKey [ 10 ] )
```

A.4 Sbox - Table de substitution

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

Table A.3 : La Sbox représente une table de substitution utilisée pour l'opération non-linéaire SubBytes de l'algorithme AES. À chaque donnée hexadécimale d'entrée correspond une donnée hexadécimale de sortie.

A.5 Opération *KeySchedule*

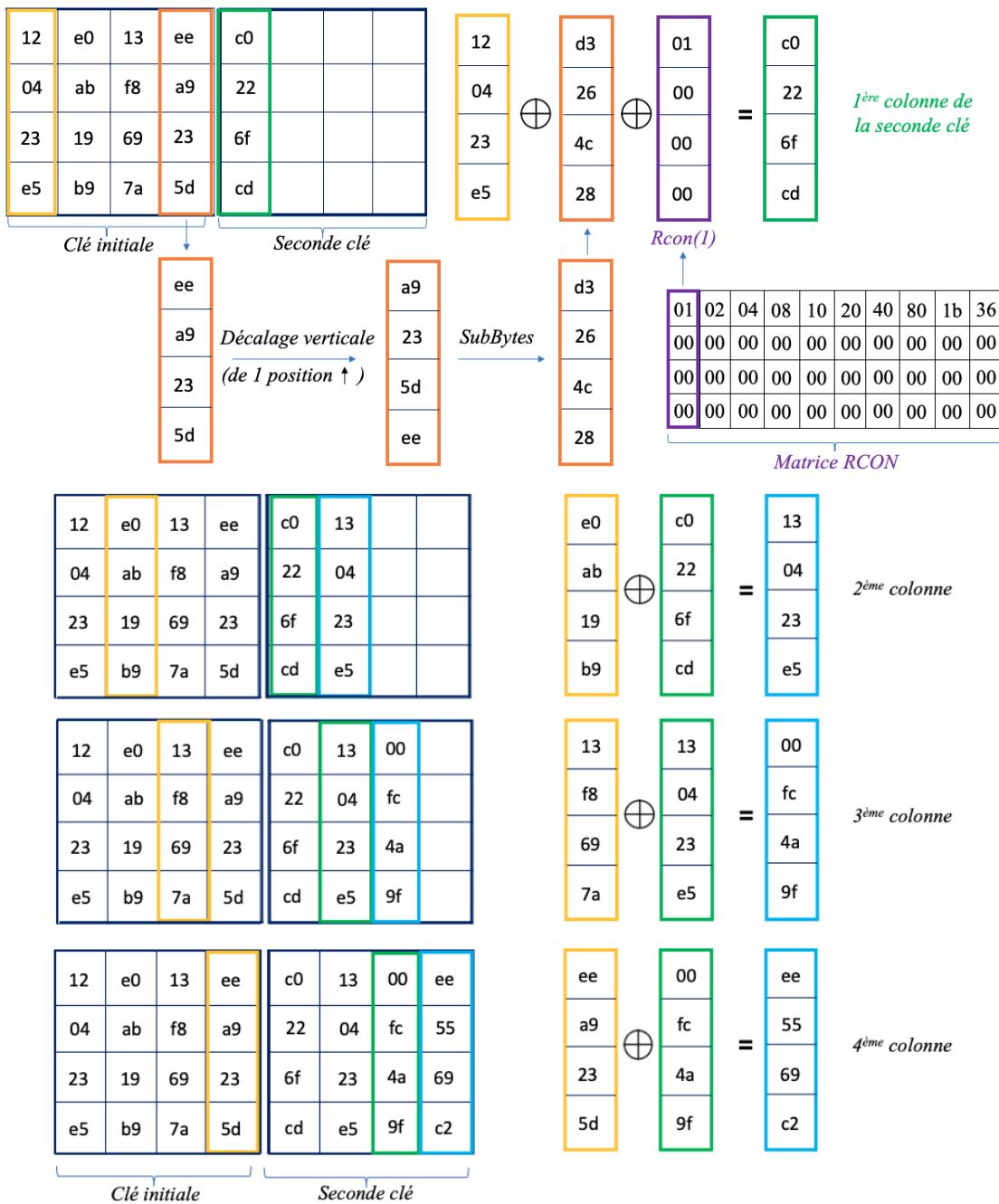


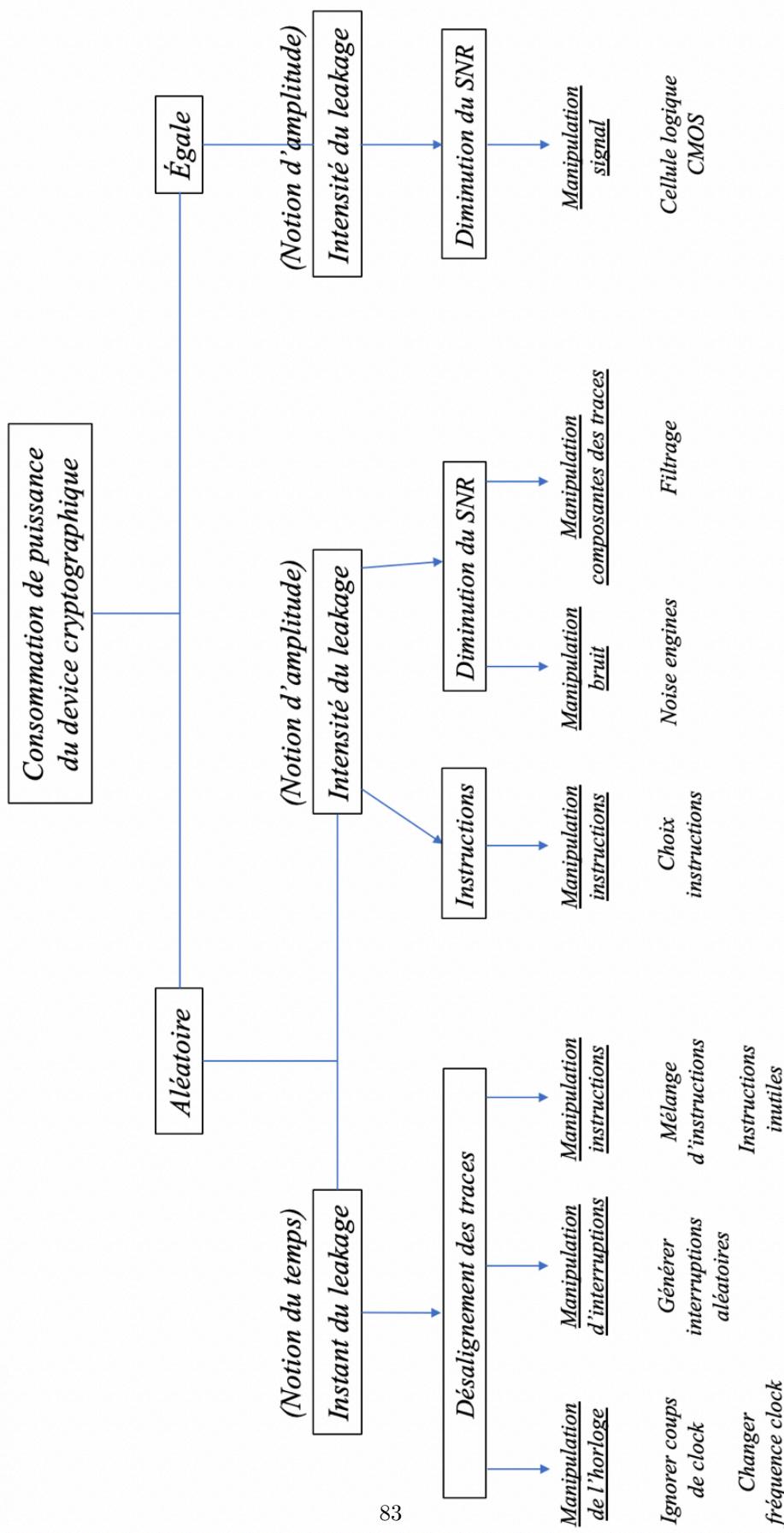
Figure A.1 : Principe de fonctionnement de l'opération *KeySchedule*.

Annexe **B**

Liste des annexes du chapitre 5

— Annexe B.1 : Contre-mesures *Hiding*.

B.1 Contre-mesures *Hiding*



Annexe C

Liste des annexes du chapitre 7

- Annexe C.1 : Présentation des caractéristiques fréquentielles (amplitude, phase et réponse impulsionnelle) du filtre FIR de type passe-bas utilisé.

C.1 Caractéristiques du filtre FIR (Hamming)

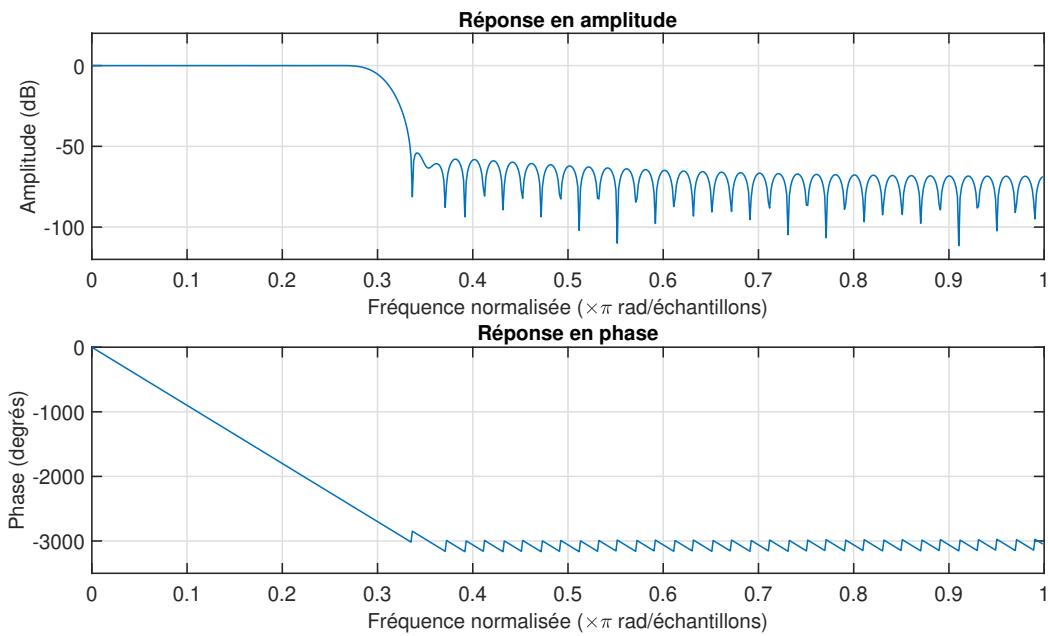


Figure C.1 : Réponse en amplitude et en phase du filtre FIR passe-bas utilisé pour le projet.

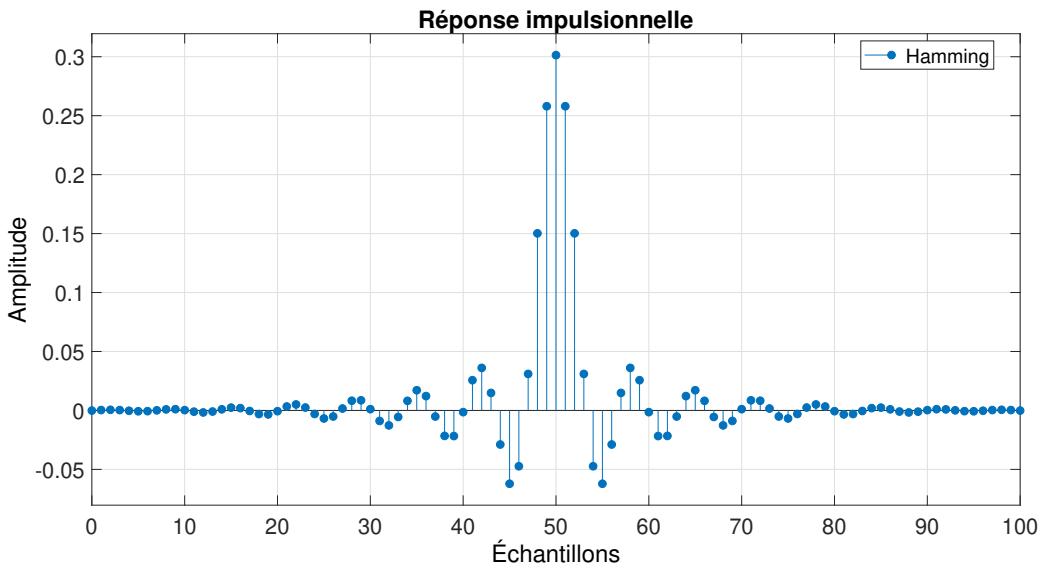


Figure C.2 : Réponse impulsionnelle du filtre FIR passe-bas utilisé pour le projet.

Annexe **D**

Liste des annexes du chapitre 9

- Annexe D.1 : Rapport de *design* complet pour l'implémentation de l'AES-256.
- Annexe D.2 : Rapport de *design* complet pour l'implémentation de la contre-mesure *faking* en mode parallèle.
- Annexe D.3 : Rapport de *design* complet pour l'implémentation de la contre-mesure *faking* en mode séquentiel.

D.1 Rapport de l'AES-256

main_fpga_aes_256 Project Status			
Project File:	Main_FPGA.xise	Parser Errors:	No Errors
Module Name:	main_fpga_aes_256	Implementation State:	Programming File Generated
Target Device:	xc6slx75-3csg484	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	17 Warnings (15 new)
Design Goal:	Balanced	• Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• Timing Constraints:	X 1 Failing Constraint
Environment:	System Settings	• Final Timing Score:	64 (Timing Report)

Device Utilization Summary [-]				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	3,526	93,296	3%	
Number used as Flip Flops	3,526			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	2,742	46,648	5%	
Number used as logic	2,626	46,648	5%	
Number using O6 output only	2,348			
Number using O5 output only	0			
Number using O5 and O6	278			
Number used as ROM	0			
Number used as Memory	0	11,072	0%	
Number used exclusively as route-thrus	116			
Number with same-slice register load	116			
Number with same-slice carry load	0			
Number with other load	0			
Number of occupied Slices	1,473	11,662	12%	
Number of MUXCYs used	92	23,324	1%	
Number of LUT Flip Flop pairs used	4,517			
Number with an unused Flip Flop	1,244	4,517	27%	
Number with an unused LUT	1,775	4,517	39%	
Number of fully used LUT-FF pairs	1,498	4,517	33%	
Number of unique control sets	28			
Number of slice register sites lost to control set restrictions	58	93,296	1%	
Number of bonded IOBs	35	328	10%	
Number of LOCed IOBs	35	35	100%	
IOB Flip Flops	5			
Number of RAMB16BWERS	0	172	0%	
Number of RAMB8BWERS	8	344	2%	
Number of BUFI02/BUFI02_2CLKs	0	32	0%	
Number of BUFI02FB/BUFI02FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	12	0%	
Number of ILOGIC2/ISERDES2s	0	442	0%	

D.2 Rapport de la contre-mesure *faking* en mode parallèle

main_fpga_aes_256 Project Status (05/07/2019 - 10:50:52)			
Project File:	AES_256_Faking.xise	Parser Errors:	No Errors
Module Name:	main_fpga_aes_256	Implementation State:	Programming File Generated
Target Device:	xc6slx75-3csg484	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	21 Warnings (5 new)
Design Goal:	Balanced	• Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• Timing Constraints:	All Constraints Met
Environment:	System Settings	• Final Timing Score:	0 (Timing Report)

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	6,734	93,296	7%	
Number used as Flip Flops	6,734			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	5,493	46,648	11%	
Number used as logic	5,313	46,648	11%	
Number using O6 output only	4,686			
Number using O5 output only	0			
Number using O5 and O6	627			
Number used as ROM	0			
Number used as Memory	0	11,072	0%	
Number used exclusively as route-thrus	180			
Number with same-slice register load	180			
Number with same-slice carry load	0			
Number with other load	0			
Number of occupied Slices	2,822	11,662	24%	
Number of MUXCYs used	92	23,324	1%	
Number of LUT Flip Flop pairs used	9,118			
Number with an unused Flip Flop	2,766	9,118	30%	
Number with an unused LUT	3,625	9,118	39%	
Number of fully used LUT-FF pairs	2,727	9,118	29%	
Number of unique control sets	41			
Number of slice register sites lost to control set restrictions	58	93,296	1%	
Number of bonded IOBs	35	328	10%	
Number of LOCed IOBs	35	35	100%	
IOB Flip Flops	5			
Number of RAMB16BWERS	0	172	0%	
Number of RAMB8BWERS	8	344	2%	
Number of BUFI02/BUFI02_2CLKs	0	32	0%	
Number of BUFI02FB/BUFI02FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	12	0%	
Number of ILOGIC2/ISERDES2s	0	442	0%	

D.3 Rapport de la contre-mesure *faking* en mode séquentiel

main_fpga_aes_256 Project Status (05/07/2019 - 10:50:52)			
Project File:	AES_256_Faking.xise	Parser Errors:	No Errors
Module Name:	main_fpga_aes_256	Implementation State:	Programming File Generated
Target Device:	xc6slx75-3csg484	• Errors:	No Errors
Product Version:	ISE 14.7	• Warnings:	21 Warnings (0 new)
Design Goal:	Balanced	• Routing Results:	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• Timing Constraints:	All Constraints Met
Environment:	System Settings	• Final Timing Score:	0 (Timing Report)

Device Utilization Summary				
Slice Logic Utilization	Used	Available	Utilization	Note(s)
Number of Slice Registers	6,609	93,296	7%	
Number used as Flip Flops	6,609			
Number used as Latches	0			
Number used as Latch-thrus	0			
Number used as AND/OR logics	0			
Number of Slice LUTs	4,970	46,648	10%	
Number used as logic	4,804	46,648	10%	
Number using O6 output only	4,305			
Number using O5 output only	0			
Number using O5 and O6	499			
Number used as ROM	0			
Number used as Memory	0	11,072	0%	
Number used exclusively as route-thrus	166			
Number with same-slice register load	166			
Number with same-slice carry load	0			
Number with other load	0			
Number of occupied Slices	2,741	11,662	23%	
Number of MUXCYs used	92	23,324	1%	
Number of LUT Flip Flop pairs used	8,643			
Number with an unused Flip Flop	2,275	8,643	26%	
Number with an unused LUT	3,673	8,643	42%	
Number of fully used LUT-FF pairs	2,695	8,643	31%	
Number of unique control sets	44			
Number of slice register sites lost to control set restrictions	55	93,296	1%	
Number of bonded IOBs	35	328	10%	
Number of LOCed IOBs	35	35	100%	
IOB Flip Flops	5			
Number of RAMB16BWERS	0	172	0%	
Number of RAMB8BWERS	16	344	4%	
Number of BUFI02/BUFI02_2CLKs	0	32	0%	
Number of BUFI02FB/BUFI02FB_2CLKs	0	32	0%	
Number of BUFG/BUFGMUXs	2	16	12%	
Number used as BUFGs	2			
Number used as BUFGMUX	0			
Number of DCM/DCM_CLKGENs	0	12	0%	
Number of ILOGIC2/ISERDES2s	0	442	0%	

Annexe **E**

Autres annexes

- Annexe **E.1** : Lettre de motivation.
- Annexe **E.2** : Certificat de stage.

E.1 Lettre de motivation

Anizet Thomas
 Rue Henri Loriaux, n°38
 6210 Frasnes-Lez-Gosselies
 Belgique
 +32 476 62 69 90
thomas.anizet@hotmail.com

Thales Communication
 Rue des Frères Taymans, 28
 1480 Tubize

Les Bons Villers, le 27 Avril 2018

OBJET : LETTRE DE MOTIVATION
 DOSSIER DE STAGE D'INSERTION PROFESSIONNELLE
 DOSSIER DE TRAVAIL DE FIN D'ÉTUDE (TFE)

Monsieur Dardenne,

Candidat "Officier de carrière" à l'École Royale Militaire, je suis actuellement ma formation académique à l'École Centrale des Arts et Métiers (ECAM) en option électronique.

Durant notre 2^{ème} année de Master, les étudiants doivent réaliser dans un premier temps un stage d'insertion professionnelle en entreprise d'une durée de 30 jours (Septembre – Novembre 2018). Ce stage consiste en l'étude d'une organisation entrepreneuriale avec son management, son contexte social, son insertion économique, ses aspects techniques et ses produits. Il se veut également participatif en nous encourageant à se consacrer de manière autonome et active à un projet industriel.

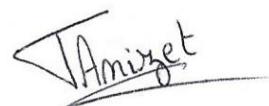
Par la suite, durant le second semestre (Février - Juin 2019), il est prévu que les élèves consacrent 2 jours de leur semaine à la réalisation de leur « Travail de Fin d'Étude » (TFE). Il est généralement convenu qu'une entreprise propose à un étudiant un sujet de TFE basé sur le stage préalablement suivi. Le stage peut donc servir d'objectif intermédiaire à atteindre avant de poursuivre le projet dans le cadre du TFE.

Ayant réalisé mon stage de 3^{ème} Bachelor chez AIRBUS DS SLC, j'aimerais découvrir une nouvelle entreprise travaillant dans les télécommunications. C'est pourquoi, je suis fortement intéressé de réaliser mon stage d'insertion professionnelle ainsi que mon TFE chez Thales afin d'en apprendre davantage sur la société mais aussi afin d'approfondir mes connaissances dans les télécommunications.

Vous trouverez ci-joint mon CV.

Je me tiens à votre entière disposition pour tout renseignement supplémentaire.

Dans l'attente d'une réponse favorable à ma demande, je vous prie d'agréer, Monsieur Dardenne, mes respectueuses salutations.



Thomas Anizet

E.2 Certificat de stage



THALES BELGIUM
Rue en Bois 63
4040 HERSTAL
Belgium
Tel. : +32 (0) 4 248 20 77
Fax : +32 (0) 4 248 25 10
www.thalesgroup.com

CERTIFICAT DE STAGE

Je soussigné, Alain QUEVRIN, en qualité de Chief Executive Officer, atteste par la présente que Thomas Anizet , né à Nivelles (Belgique), le 30 Avril 1996, et domicilié à Rue Henri Loriaux n°38, 6210 Les Bons Villers, a effectué un stage dans notre entreprise du 17 Septembre 2018 au 26 Octobre 2018.

Fait à Tubize, le 26 Octobre 2018.

Alain QUEVRIN

CEO



(po. Alain Quevrin)

Cachet de l'entreprise

THALES Belgium
Rue en Bois 63
4040 Herstal