Introduction d'aléas dans les architectures matérielles pour une contribution à la sécurisation de chiffreurs AES dans un contexte IoT

Ghita Harcha

THESE DE DOCTORAT DE

L'UNIVERSITE BRETAGNE SUD

ECOLE DOCTORALE : MATHSTIC N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : ELECTRONIQUE
-

Par
Ghita Harcha

Introduction d'aléas dans les architectures matérielles pour une contribution à la sécurisation de chiffreurs AES dans un contexte IoT

Thèse présentée et soutenue à Lorient, le 13 juillet 2021
Unité de recherche : Lab-STICC (UMR 6285)
Thèse N° : 603

Rapporteurs avant soutenance :

Cécile Belleudy	Maîtresse de Conférences – HDR, Université Côte d'azur
Roselyne Chotin	Maîtresse de Conférences – HDR, Sorbonne Université

Composition du Jury :

Président :	Lilian Bossuet	Professeur des Universités, Université Jean Monnet
Examinateurs :	Emmanuel Casseau	Professeur des Universités, Université Rennes 1
	Christophe Jego	Professeur des Universités, Bordeaux INP
Dir. de thèse :	Philippe Coussy	Professeur des Universités, Université Bretagne Sud
Co-encadrants :	Vianney Lapôtre	Maître de conférence, Université Bretagne Sud
	Cyrille Chavet	Maître de conférence, Université Bretagne Sud

Titre : Introduction d'aléas dans les architectures matérielles pour une contribution à la sécurisation de chiffreurs AES dans un contexte IoT

Mots clés : Systèmes embarqués, attaques par canaux cachés, chiffrement AES

Résumé : Nous vivons dans un monde où l'information et l'échange de données sont devenus des éléments clefs de nos économies. Il faut ajouter à cela l'explosion et la diffusion rapide de ce que l'on appelle l'internet des objets (IoT, Internet of Things) à tous les niveaux de nos sociétés et dans nos vies, tant professionnelles que personnelles. Il s'agit là de systèmes embarqués communicants très contraints en taille et en énergie, déjà largement déployés. Toutefois, ces derniers présentent de nombreuses vulnérabilités et de ce fait font partie des cibles privilégiées pour des attaques malveillantes. C'est pourquoi, ces dispositifs s'accompagnent de plus en plus de systèmes de chiffrement. Malheureusement, leurs implémentations peuvent elles-mêmes être sujettes à des failles. Dans cette thèse nous nous intéressons à la sécurisation d'une architecture de chiffrement AES face à des attaques par canaux cachés, notamment les attaques dites par « *observation de consommation de puissance* ». Le domaine de l'IoT étant ciblé, des architectures d'AES faible coût sont visées et l'objectif est de minimiser l'impact en termes de surface, débit, latence et consommation.

L'approche proposée consiste à adjoindre à un composant AES un module de génération d'aléa. Dans ce contexte, plusieurs solutions architecturales sont : le nombre de permutations et le type de d'informations permutées. La robustesse des différents architectures face à différentes attaques de l'état de l'art est évaluée. Les surcouts induits par le composant de brassage sont quantifiés et les effets des options de synthèse sont étudiés.

Les résultats montrent qu'avec notre modèle architectural le plus sûr (et donc le plus complexe et le plus coûteux) aucun octet de la clef de chiffrement n'est révélé après un million d'échantillons mesurés sur FPGA. Cet apport en sécurité a un coût, contenu au regard des solutions présentes dans la littérature : un débit divisé d'un facteur *2,3* ; une réduction de *11%* de la fréquence max ; un surcoût matériel équivalent à environ *3,9* fois l'architecture d'origine.

Title : Introducing shuffling into hardware architectures: a contribution to the security of AES cyphers in an IoT context

Keywords : Embeded systems, side channel attacks, AES

Abstract : Nowadays, information and data exchange are key elements of our economies. Furthermore, one must add to this the explosion and rapid diffusion of the so-called Internet of Things (IoT, Internet of Things) on every aspect of our lives, professional and personal. These are area and energy-constrained embedded communicating systems, which are already widely deployed. However, IoT nodes present many vulnerabilities and therefore are among the targets of malicious attacks. Therefore, these devices rely more and more on encryption systems. Unfortunately, their implementations themselves can be prone to defaults.

In this PhD thesis, we are interested in securing an AES encryption architecture against side-channel attacks, and in particular on power analysis attacks. As the field of IoT is targeted, low-cost AES architectures are targeted and the goal is to minimize the area, throughput, latency, and energy overhead. The proposed approach consists in adding a shuffling module to an AES component.

In this context, several architectural solutions have been studied: the number of permutations and the types of permuted information. The robustness of the different architectures against different state-of-the-art attacks is evaluated. The additional costs induced by the shuffling module are quantified and the effects of the synthesis options are studied.

The results show that no byte of the encryption key was revealed on our most secure (and therefore the most complex and expensive) architectural model is revealed after one million power traces measured on the FPGA. This safety contribution is not for free, which is contained with regard to the solutions proposed on the state of the art: Regarding throughput, the factor overhead is 2.3; and the maximum frequency overhead is about 11%; the hardware cost is approximately equivalent to 3.9 times compared to our reference design.

Remerciements

En premier temps je tiens à remercier mon directeur de thèse Philippe COUSSY ainsi que mes deux encadrants Vianney LAPOTRE et Cyrille CHAVET, qui m'ont accompagnée tout au long de cette thèse par leurs conseils, leur soutien et leur patience.

Je remercie également Arnaud TISSERAND pour ces précieux conseils dans le domaine de sécurité, ainsi que les deux membres de mon comité de suivi individuel de thèse Emmanuel CASSEAU et Jean-Louis LANET.

Enfin je remercie mes parents, ma sœur et ma famille ainsi que mes amis et toutes les personnes que j'ai rencontrées et avec lesquelles j'ai passé de bons moments durant cette thèse.

Table des matières

Table des figures

Avant-propos

Ces dernières années le nombre d'objets connectés, ou nœuds IoT (Internet of Things), a augmenté de façon exponentielle. Au milieu des années 2010, les spécialistes estimaient le nombre d'IoT dans le monde autour de 20 milliards en 2020 [1]. Désormais les analyses estiment ce nombre à 43 milliards en 2023 [2].

Quand on parle de nœuds IoT, on parle de dispositifs petits, peu chers, presque "jetables". Il n'en demeure pas moins que ces objets connectés, si basiques soient-ils, constituent autant de vecteurs d'attaques potentiels pour des individus mal intentionnés.

Les exemples récents ont démontré les risques que faisaient porter ce type de produits pour l'ensemble des systèmes connectés à internet : dénis de service, fuite d'information... En 2016, un malware nommé MIRAI (cf. [3]) a tiré profit d'une flotte de composants connectés, dont les mots de passe par défaut n'avaient pas été modifiés. Ce type de composants se retrouve souvent dans des petites cameras connectées ou bien des petits routeurs pour les particuliers. L'objectif de MIRAI était de créer un « botnet » et de l'utiliser ensuite pour mener des attaques DDOS (déni de service distribué) inondant les réseaux de plusieurs entreprises, les rendant ainsi inaccessibles aux utilisateurs. Ainsi, en septembre 2016, une telle attaque a surchargé les serveurs de sites web (p.ex. Krebs on security) ou d'hébergeurs (p.ex. OVH) avec des débits pouvant atteindre le Térabits par seconde. Quelques semaines plus tard, ce sont les services DNS de l'entreprise Dyn qui ont été à leur tour visés par le botnet MIRAI, avec entre autres conséquences des difficultés pour accéder à des sites comme Netflix, GitHub, Twitter….

Ce simple exemple montre l'importance d'intégrer des mécanismes de sécurité dès la conception des systèmes. Toutefois, pour des produits à bas coût, les entreprises ne voient pas d'intérêt financier (à court terme) pour investir dans la sécurisation de leurs systèmes. Il n'en reste pas moins que la vie privée des clients et la garantie du secret de leurs informations personnelles doivent être assurées. Pour ce faire, il existe des solutions de chiffrement et/ou de signature suffisamment robustes (si elles sont correctement mises en œuvre et exploitées) pour un coût contenu pour le fabriquant.

La prise en compte des aspects « sécurité » lors de la conception des systèmes n'est toutefois pas une garantie absolue contre les attaques. En effet, la mise en œuvre de ces mécanismes peut être imparfaite ou bien laisser « fuiter » involontairement des informations

secrètes. Comme dans un jeu de poker menteur, les machines doivent garder le secret de leurs atouts, mais parfois il se peut qu'elles laissent échapper des indices pouvant être exploités par des attaquants.

De telles attaques ont été par exemple récemment utilisées [4] dans la contrefaçon de batteries des cigarettes électroniques. Le firmware a été attaqué en volant la clé secrète du chiffrement depuis une batterie authentique, via une attaque par canaux cachés en mesurant le rayonnement électromagnétique du microcontrôleur intégré. Une autre expérience a montré que dans la pratique, il est possible de récupérer une clé secrète de chiffrement en utilisant une antenne et un amplificateur dans une chambre adjacente à travers un mur d'une épaisseur de 15cm renforcé avec des montants métalliques, en 3.3 secondes [5]. En 2015, un « gadget » compact a été conçu par une équipe de chercheur de Tel Aviv. Celui-ci est capable de retrouver la clé de chiffrement manipulée par le CPU durant le chiffrement des données [6]. Là encore, l'idée est de récupérer et d'analyser les émissions électromagnétiques de la cible. Des chercheurs de la même université ont démontré dans [7] qu'un microphone collectant le son d'un ordinateur (ventilation, alimentation…) tandis qu'il déchiffre un message, est capable de révéler des informations secrètes. Si l'on connait l'algorithme de chiffrement, ces informations peuvent être corrélées pour tenter de retrouver des clés chiffrement par exemple, et être exploitées pour révéler ce qu'une personne est en train de regarder sur Netflix ou Youtube sans avoir à accéder à l'ordinateur cible.

Afin de définir le degré de sécurité d'un système (quel qu'il soit), une attestation de certification est attribuée par une entité reconnue au niveau mondial et permet d'offrir une garantie de sécurité contre un niveau d'attaque donné. En France, c'est l'ANSSI (Agence Nationale de la Sécurité des Systèmes d'Information) qui est en charge de ces aspects, mais il existe d'autres organismes à l'international (ISO, NIST, ENISA, CLUSIF, CSA...). Ce certificat est délivré après une évaluation faite par des laboratoires tierces agréés par les organismes de certification. En France, les laboratoires CESTI se chargent de ces évaluations en tenant compte de toutes les règles du schéma élaboré par l'ANSSI.

L'un des mécanismes de sécurité utilisé couramment est la cryptographie. Celle-ci est utilisée pour assurer la confidentialité des communications depuis l'antiquité, mais la cryptographie moderne n'est apparue que depuis la fin du XX^{ième} siècle. Dans cette thèse on s'intéresse à la protection des solutions de mise en œuvre matérielle d'algorithmes de chiffrement modernes, contre des attaques exploitant des fuites d'informations pouvant résulter de leur

conception même. Les systèmes de chiffrement que nous considérons se doivent d'être peu coûteux, puisqu'ils sont destinés à la protection de systèmes embarqués (c.-à-d. contrainte de surface, de débit, de consommation énergétique…). Dans cette optique, notre approche consiste à intégrer dans un crypto-système matériel, un module de génération d'aléa (module de brassage) à la fois simple et efficace.

Ce document est organisé comme décrit ci-après.

Dans le premier chapitre, nous faisons un rappel des techniques de chiffrement moderne, et nous introduisons les grandes familles d'attaques physiques et enfin les protections contre les attaques par canaux cachés. Dans le second chapitre, nous présentons des architectures matérielles faible coût d'un chiffreur AES 8 bits dont celle que nous avons retenue pour nos expériences. Nous introduisons également les différentes solutions de protection existant dans la littérature, en nous focalisant sur les protections par ajout d'aléa basées sur des implémentations matérielles. Le troisième chapitre détaille nos contributions pour l'ajout d'aléa sur un seul bloc de données avec génération de permutation matérielle à l'aide d'un réseau de permutation Benes. Nous présentons les résultats en termes de surface et de performance, ainsi qu'une évaluation en termes de sécurité. Dans le quatrième chapitre, nous analysons l'impact de l'utilisation d'un réseau alternatif (c.-à-d. moins coûteux) offrant un nombre plus faible de permutations, ainsi que l'impact des options de synthèse sur les résultats. Le cinquième chapitre décrit notre contribution pour l'ajout d'aléa dans un module matériel AES ayant un mode de chiffrement multi-blocs. Nous y présentons les résultats en termes de surface, performance et sécurité. Enfin, le dernier chapitre conclut sur les travaux de cette thèse et ouvre sur des perspectives.

Chapitre 1 Introduction

I. La cryptographie moderne

Le rôle de la cryptographie est d'assurer l'intégrité des communications tout en assurant la protection des données sensibles, l'identification des parties et l'authenticité du message. Pour garantir cette protection, les algorithmes de chiffrement utilisent une clé secrète selon le principe énoncé par Kerckhoffs : "La sécurité d'un crypto système ne doit reposer que sur le secret de la clef" [8].

Dans le cas d'un algorithme symétrique, cette même clé est aussi utilisée pour le déchiffrement, tandis que dans le cas d'un algorithme asymétrique une clé publique connue de tous est utilisée pour le chiffrement et une clé secrète (qui ne peut être dérivée de la clé publique) est utilisée pour le déchiffrement.

Le rôle de la clé de chiffrement est très important pour chaque utilisation (chiffrement, authentification, signature…) une clé unique doit être utilisée. L'utilisation d'une même clé de chiffrement pour de multiple usages va grandement affaiblir la sécurité du système, et peut interférer avec son fonctionnement dans certains cas. Toute clé doit posséder une durée de vie limitée appelée *crypto-période*, qui a pour rôle de limiter le nombre de chiffrements qui pourraient être exploités par cryptanalyse. En limitant le nombre d'utilisations d'une clef de chiffrement donnée, on limite ainsi la taille du jeu de données exploitables pour un attaquant.

La crypto-période peut dépendre de facteurs multiples comme la robustesse du mécanisme (algorithme de chiffrement, taille du bloc, taille de la clé, mode opératoire), du flux de données ainsi que de la méthode de génération de ladite clé. Selon les recommandations du NIST pour la gestion de ces clés [9], le cycle de vie d'une clé utilisée pour chiffrer un grand volume de données sur une courte période est de l'ordre d'un jour (au plus une semaine), tandis que pour un faible volume de données cette crypto-période peut aller jusqu'à deux ans.

Les algorithmes symétriques ont pour avantage d'être plus rapides et moins complexes que les algorithmes asymétriques. Ils sont aussi plus efficaces pour de grands flux de données. Néanmoins, une même clé étant utilisée pour le chiffrement et le déchiffrement, le problème de son partage se pose. Ce dernier peut être résolu en s'appuyant sur des protocoles d'échange de clés en utilisant un algorithme asymétrique.

Les algorithmes asymétriques sont plus complexes et donc plus lents. Ils sont non seulement utilisés pour résoudre le problème de partage de clé, mais permettent d'apporter une sécurité supplémentaire via une signature numérique pour vérifier l'identité de l'expéditeur et l'authenticité du message. Cette signature chiffre une empreinte de données produits par une fonction de hachage en utilisant la clé secrète.

Dans ces travaux de thèse, nous nous intéressons aux algorithmes symétriques et plus particulièrement aux chiffrements par bloc, une sous-catégorie des chiffrements symétriques.

II. Algorithmes symétriques

Les algorithmes symétriques de cryptographie se divisent en deux catégories : les algorithmes de chiffrement par flux, qui s'appuient sur des générateurs pseudos aléatoires qui traitent un flux de données dont la longueur n'est pas définie à l'avance; et les algorithmes de chiffrements par bloc, qui sont des algorithmes qui opèrent sur des blocs de données de mêmes tailles.

1. Chiffrement par flux

Les algorithmes de chiffrements par flux ont pour avantage d'être compacts et rapides. L'état initial est défini à l'aide d'un vecteur d'initialisation généré aléatoirement. Les algorithmes n'opèrent que sur un seul bit à la fois. Les séquences de bits sont concaténées et on obtient un flux de données pseudo-aléatoires ou *keystream* qui est alors utilisé pour chiffrer le texte clair (*plaintext*) avec une opération OU-exclusif (XOR). On obtient ainsi le texte chiffré (*cyphertext*) comme illustré dans la Figure 1.

Un algorithme de chiffrement de flux est dit synchrone quand le flux de données pseudo aléatoire ne dépend que du vecteur d'initialisation et de la clé secrète ; il est dit asynchrone quand il dépend en plus du texte chiffré.

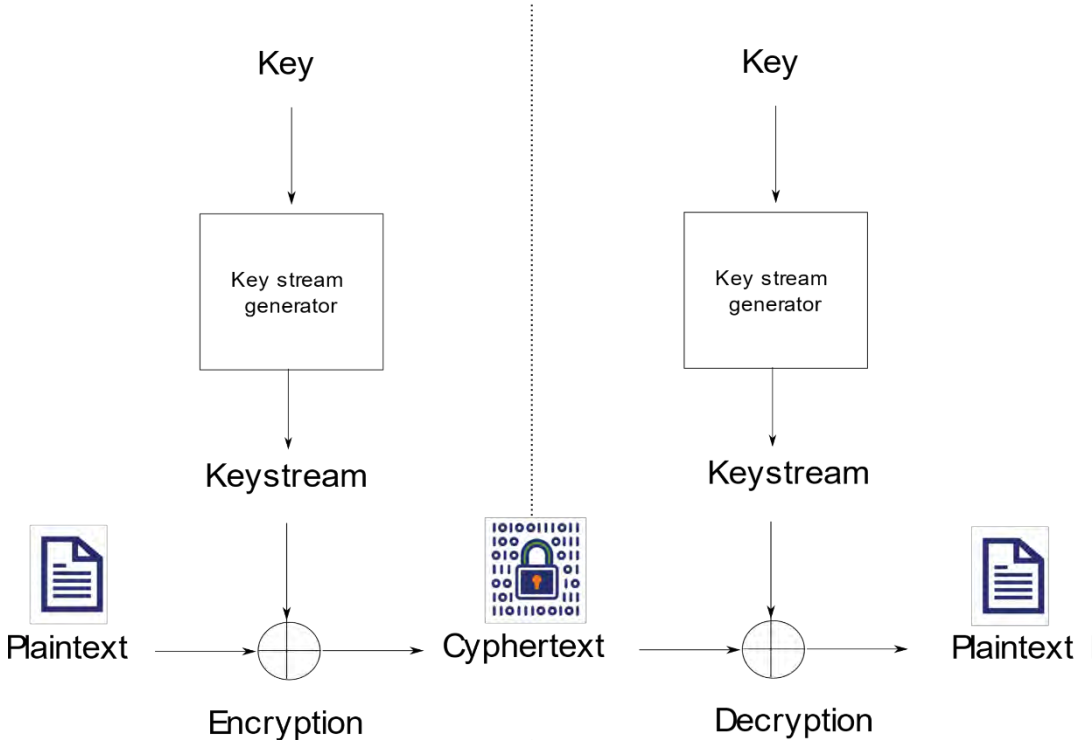


Figure 1: Principe du chiffrement et déchiffrement par flux

2. Chiffrement par bloc

Ce type d'algorithme de chiffrement travaille sur des données de longueur fixe découpées en blocs (la taille dépend de l'algorithme) et se caractérise par un ensemble de transformations prédéfinies opérées sur chacun des blocs. On parle parfois de ronde, ainsi que de clé de ronde (potentiellement dérivée d'une clé secrète initiale).

Dans cette catégorie, les deux algorithmes les plus connus sont DES - *Data Encryption Standard* [10] qui a été le premier algorithme de chiffrement par bloc normalisé en 1977, et AES - *Advanced Encryptions Standard* (ou algorithmes de « Rijindael ») [11] qui est à présent la norme internationale en remplacement du DES depuis 2001.

a. AES - Advanced Encryptions Standard

L'algorithme AES est un réseau de substitutions-permutations (Subsitution-Permutation Network) sur le modèle proposé par Shannon en 1949 pour définir un chiffrement idéal apportant une forte résistance aux attaques statiques [12]. Il est constitué d'opérations de permutations qui ont pour rôle de générer de la confusion, c.-à-d. de créer une relation non triviale entre le texte clair et le texte chiffré. De plus, des opérations de substitutions permettent à chaque bit du texte chiffré d'être hautement non linéaire avec les bits du texte clair et de la clé.

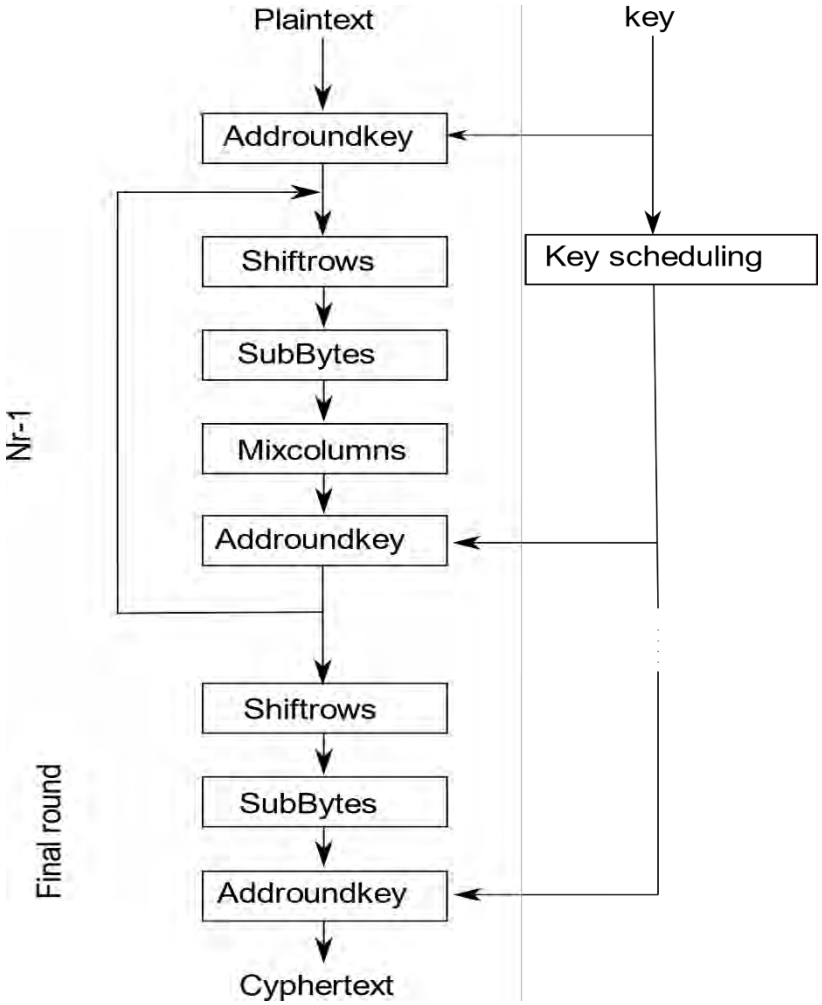


Figure 2: L'algorithme de l'AES

Un bloc de données AES a une longueur fixe (p.ex. 128 bits) représenté sous la forme d'une matrice d'états (4×4 octets) aussi appelée *state*. Il existe trois longueurs de clé dans le standard : 128, 192 ou 256 bits. Pour la NSA, les trois longueurs de clés sont suffisamment robustes pour protéger des informations jusqu'au niveau SECRET. Cependant, pour des informations classées TOP SECRET les tailles 192 et 256 sont recommandées [13]. L'AES est un algorithme exploitant une fonction composée de quatre opérations de base (*AddRoundKey*, *ShiftRows*, *SubBytes* et *MixColumns*) qui sont itérées un nombre *Nr* de fois en fonction de la longueur de la clé (cf. Figure 2).

Comme l'illustre la Figure 2, la première opération réalisée par un AES se nomme *AddRoundKey*. Cette opération consiste appliquer une opération XOR entre le *plaintext* et la clé initiale (on parle de *Key Whitening*). Ensuite, les différentes

opérations de base sont itérées respectivement 10, 12 ou 14 fois, selon la longueur de la clé retenue. Enfin, il faut noter que les opérations présentées précédemment sont réalisées dans le corps de Gallois, $GF(2^8)$:

- ***AddRoundKey*** : lors de cette opération les octets de la matrice d'états (*state*) se voient appliquer une opération XOR avec la clé de ronde courante, elle-même dérivée de la clé initiale via une opération dédiée de *Key scheduling* (voir Figure 2).
- ***ShiftRows*** : il s'agit d'une opération de permutation où les octets de la *i*^{éme} ligne de la matrice d'états sont décalés d'un nombre i de colonne, avec $0 \leq i \leq 3$.
- ***SubBytes*** : c'est une opération de substitution non-linéaire dont le but est la diffusion des bits au sein d'un même octet. Cette opération se compose de deux fonctions : 1) une inversion dans l'espace de gallois $GF(2^8)$, puis 2) une transformation affine via un polynôme. Cette opération peut être représentée par une table pré-calculée.
- ***MixColumns*** : cette opération engendre une dépendance entre les octets de chaque colonne de la matrice d'états. Chaque octet est multiplié par un coefficient $a(x) \times mod(x^4 + 1)$ avec $a(x) = \{03\}_{16}x^3 + \{01\}_{16}x^2 + \{01\}_{16}x + \{02\}_{16}$. Cette opération est omise durant la dernière ronde.

b. Modes de chiffrement de l'AES

Il existe différentes façons de chiffrer les blocs : les modes opératoires. Parmi ces modes, on en distingue trois principaux : le mode ECB - *Electronic CodeBook*, CBC - *Cipher Block Chaining* et CTR - *CounTeR*.

- ***Mode ECB***

  Le mode ECB est décrit dans la Figure 3. Chaque bloc est chiffré de la même façon, c'est le mode le plus simple de l'AES. Toutefois, ce dernier n'offre pas de sécurité sémantique (c.à.d. ne donne pas la garantie qu'un texte chiffré ne peut être décrypté sans la clé) et il ne protège pas contre les attaques à texte clair choisi (ou IND-CPA, *INDistinguishability under Chosen-Plaintext Attack*). En d'autres termes, étant donné que deux textes clairs identiques seront chiffrés de la même façon, il est mathématiquement possible de retrouver les clefs de chiffrement.

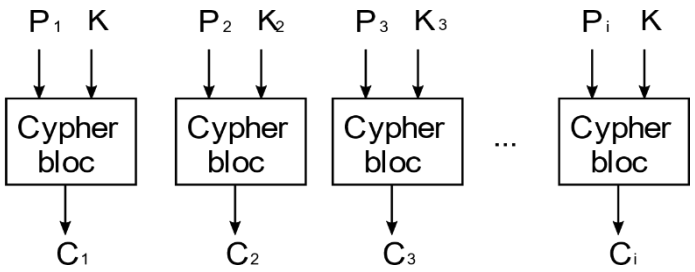


Figure 3: AES mode ECB

- ***Mode CBC***

  Dans le mode CBC, le texte clair se voit appliquer une opération XOR avec le chiffré du précédant bloc. Pour le premier chiffrement, cette opération XOR est réalisée avec un vecteur d'initialisation noté *IV* (*Initialization Vector)*, choisi de façon à être imprédictible. Chaque bloc de chiffrement est dépendant du chiffrement précédant comme décrit dans la Figure 4. Il n'est donc pas possible de paralléliser les chiffrements : le débit résultant est ainsi moindre qu'avec une architecture ECB.

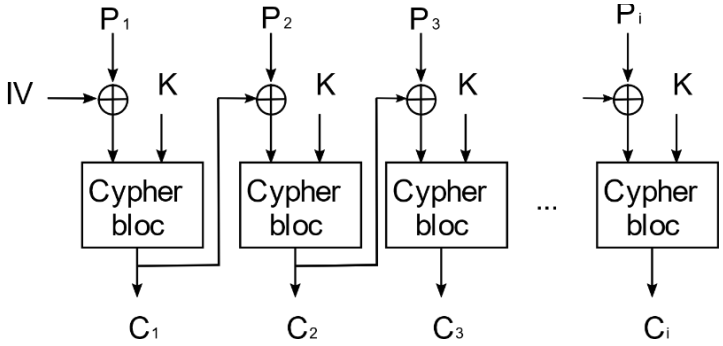


Figure 4: AES mode CBC

- ***Mode CTR***

  Le mode CTR, quant à lui, chiffre les blocs avec un compteur (Cnt_i). Les bits de poids forts de ce compteur constituent le vecteur d'initialisation et les bits de poids faibles sont incrémentés à chaque chiffrement d'un bloc. Le résultat est alors utilisé pour une opération XOR avec le plaintext. Ce mode de chiffrement est décrit dans la *Figure 5*. Contrairement au mode CBC, les chiffrements sont indépendants et peuvent donc être parallélisés.

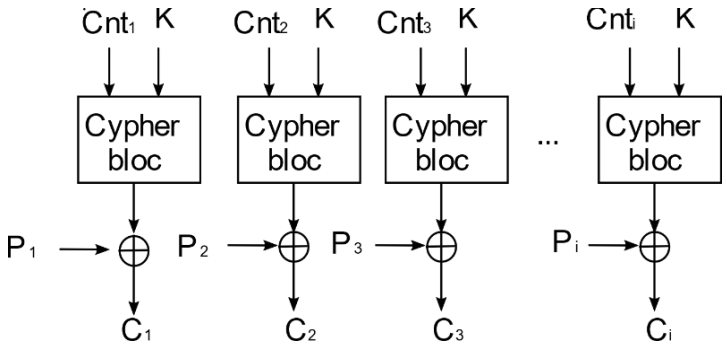


Figure 5: AES mode CTR

Comme évoqué précédemment, les algorithmes de chiffrement sont conçus pour être mathématiquement solides. Cependant leur implémentation matérielle peut être sujette à des fuites “physiques” comme la consommation de puissance, les émanations électromagnétiques, le temps de calcul... Ces grandeurs, en lien avec les calculs réalisés, peuvent être exploitées par un attaquant par simple observation. Il est également possible de coupler ces observations avec des attaques par injection de fautes dont le but est de provoquer des fuites d’informations [RM07].

III. Attaques physiques

Il existe différents types d’attaques physiques qui ont pour but d’extraire des informations en ciblant des implémentations de systèmes électroniques. Dans cette partie différents types d’attaques physiques sont expliqués selon une classification illustrée dans la Figure 6 proposée dans [14]. Un focus particulier est mis sur les attaques par observation qui ont motivées ce travail de recherche.

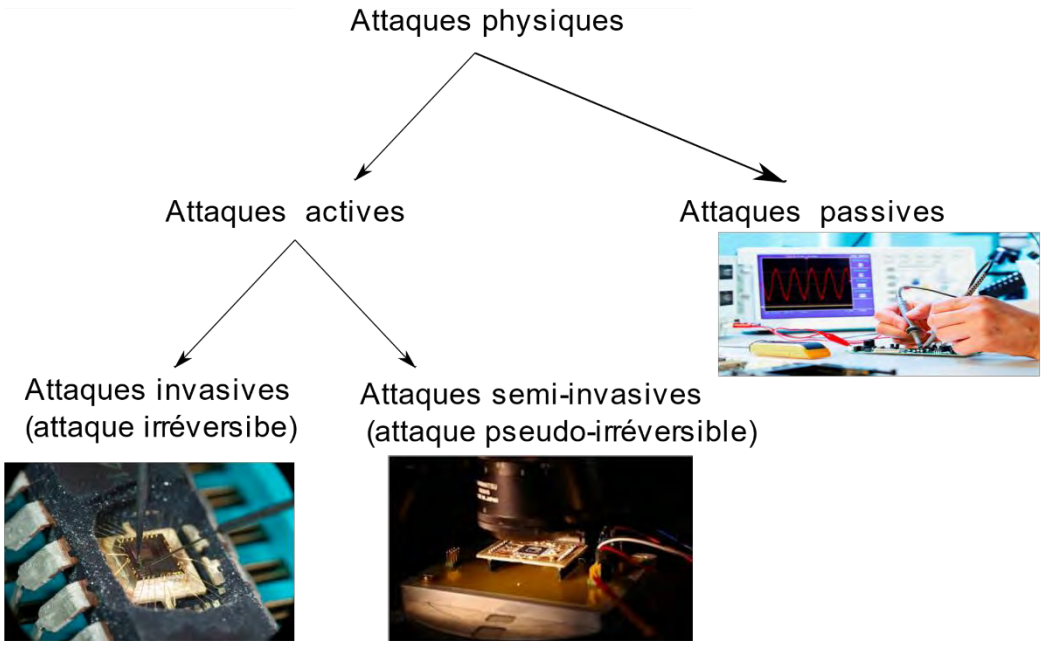


Figure 6: Classification des attaques physiques

1. Attaques invasives et semi-invasives

Les attaques invasives sont des attaques destructives et irréversibles : elles permettent d'accéder directement à l'architecture par la décapsulation de la puce. Le composant ainsi mis à nu peut alors être photographié couche par couche afin d'en étudier l'architecture [15], voir également de tenter d'en altérer le fonctionnement nominal en désactivant certaines protections internes. Ce type d'attaques très puissante présentent toutefois un coût très élevé au regard du matériel nécessaire à sa mise en œuvre.

Pour les attaques semi-invasives, une décapsulation du circuit peut être nécessaire mais son fonctionnement n'est pas forcement altéré. L'attaquant peut mesurer des grandeurs physiques comme la consommation de puissance, faire des mesures directement sur les pistes du circuit ou injecter des fautes. Cela peut se faire par exemple en faisant varier la tension d'alimentation, altérant ainsi le fonctionnement des *flip-flop* (c.-à-d. *mémoires internes*) du circuit, ou en augmentant la fréquence d'horloge au-dessus de sa fréquence maximale. Dans ce dernier exemple, le temps de transit des données étant physiquement insuffisant, il est par exemple possible d'obtenir des valeurs instables, ou bien la valeur précédente des registres.

L'attaquant peut également injecter des fautes par impulsion laser ou électromagnétique. Le comportement du circuit est alors observé, puis exploité afin de retrouver une clé secrète de chiffrement par exemple.

2. Attaques passives

Dans les travaux de cette thèse, nous nous intéressons aux attaques par observation qui consistent à exploiter les fuites non intentionnelles induites par l'implémentation physique du circuit. On peut observer par exemple sa consommation de puissance, son rayonnement électromagnétique ou son temps d'exécution.

Ces fuites sont ciblées par ce que l'on appelle des *attaques par canaux cachés* (*SCA*, *Side Channel Attacks*). Le terme est apparu pour la première fois en 1998 dans un article [16] décrivant la façon dont ces "canaux cachés" peuvent être exploités afin de "casser" des systèmes cryptographiques. Ce type d'attaque exploite des informations fuitées non intentionnellement par le système cible, en raison de la liaison existante entre un calcul et des grandeurs physiques mesurées.

a. Observation du temps de calcul

Puisque chaque opération prend en fonction de son type et de sa complexité un certain temps pour être exécutée, ce temps d'exécution peut être exploité. Dans le cas où il existe une dépendance entre le temps de calcul et la clé secrète (p.ex. pour le calcul d'une exponentiation modulaire, dont le temps de calcul dépend du nombre de '1' dans la clé), il est alors possible de retrouver la clé secrète par la mesure du temps d'exécution. Une telle attaque a été présentée pour la première fois par Paul Kocher en 1996 [17].

b. Observation de la consommation de puissance et du rayonnement électromagnétique

La consommation de puissance dépend du nombre de cellules logiques activées et de la façon dont elles sont construites. La consommation de puissance totale d'un circuit CMOS se divise en deux parties :

- la *puissance statique*, représentée par l'équation $P_{stat} = V_{DD} \times I_{Sat}$ correspondant à la consommation de puissance dissipée quand le circuit est alimenté ;
- la *puissance dynamique* est représentée par l'équation :

$$P_{dyn} \approx \alpha \times \tau \times V^2 \times C_L \times f.$$

Cette consommation est causée par la charge et la décharge de capacités C_L du circuit.
Avec C_L: la capacité de charge logique.

f : la fréquence d l'horloge.

V: la tension d'alimentation

α: l'activité de commutation

Lors d'une transition de 0 à 1, la capacité de sortie est chargée. Pour une transition de 1 à 0 elle est déchargée, comme c'est illustré dans Figure 7.

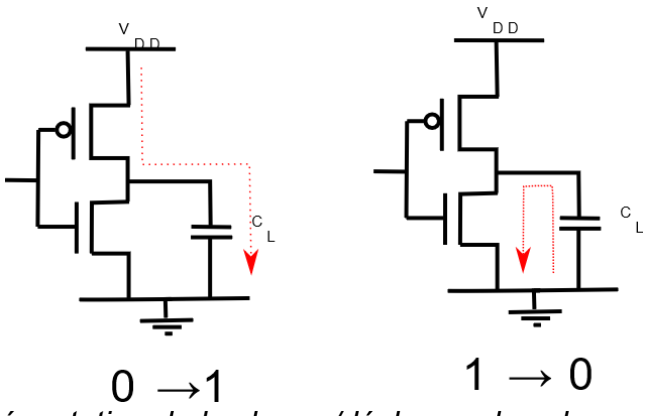


Figure 7: Représentation de la charge/décharge dans le cas d'un inverseur

La consommation de puissance dynamique est directement liée aux transitions et donc aux données manipulées. C'est donc ce type d'informations que l'on va chercher à exploiter dans une attaque par canaux cachés.

Il est aussi possible de mesurer le champ électromagnétique : ce dernier est proportionnel à la consommation de puissance de tout ou partie d'un circuit selon la taille et la position de la sonde.

La première attaque par consommation de puissance a été introduite par Paul Kocher en 1999 en [18]. Dans ce travail les attaques SPA - Simple Power Analysis et la DPA - Differential Power Analysis ont été introduites.

- ***Attaque de type SPA***

  L'attaque SPA repose sur l'analyse directe d'une très faible quantité de traces de consommation de puissance afin d'en déduire la clé secrète. Une attaque temporelle peut être réalisée en observant les motifs d'une trace de consommation de puissance.

  Par exemple, l'opération d'expansion de la clé de l'algorithme DES [19] comporte une rotation dans le registre qui stocke les bits de la clé. Une opération de branchement est alors utilisée pour vérifier le bit à décaler. Dans la Figure 8, les flèches montrent ces rotations. On observe qu'il y a eu une rotation dans la ronde 2 et deux rotations dans la ronde 3.

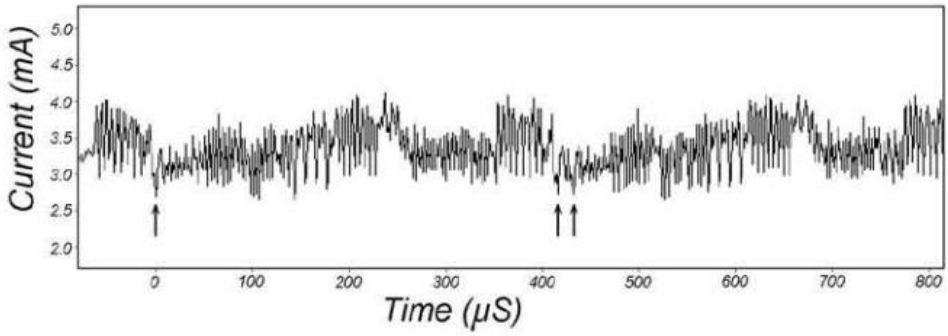


Figure 8: Trace de consommation des rondes 2 et 3 d'un algorithme DES [19]

Les itérations des calculs étant facilement distinguables dans une trace, il est alors possible d'identifier l'algorithme de chiffrement avant de mettre en place une attaque différentielle (p.ex. en cherchant à identifier des motifs particuliers répétitifs dans la trace), comme illustré dans la Figure 9 où l'on peut facilement distinguer les 16 rondes du chiffrement DES.

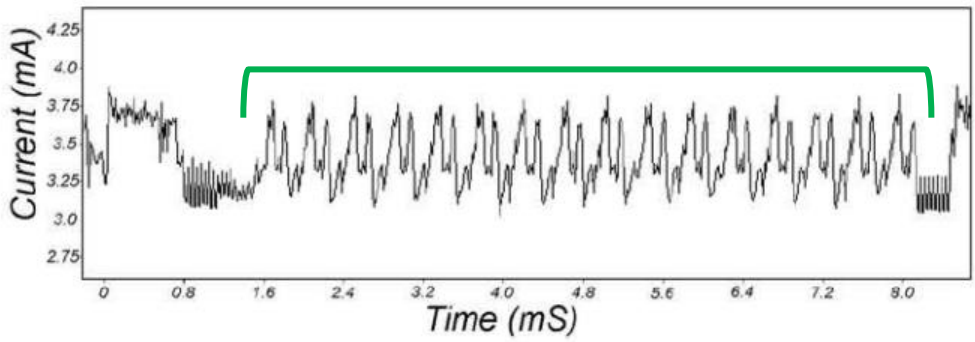


Figure 9:Trace de consommation d'un chiffrement DES complet [19]

- ***Attaque de type DPA***

  Dans une attaque différentielle, contrairement à la SPA, un nombre important de traces de consommation de puissance est utilisé. L'analyse ne se fait pas au niveau de l'axe du temps, mais se focalise sur la dépendance entre la consommation de puissance et le calcul des données à un point fixe (on observe la différence de la consommation de puissance d'un même point pour chaque chiffrement).

  Dans un premier temps, une valeur intermédiaire manipulée durant les traitements est choisie. Cette valeur doit être le résultat d'une fonction $f(d,k)$ dont d est connu et k est le secret. Des traces de consommation de puissance sont collectées pendant le chiffrement/déchiffrement de D différents blocs. On

note $d = (d_1 \dots d_D)$ le vecteur de données connu et t une matrice de dimension $D \times T$ avec T la longueur de la trace.

Les valeurs intermédiaires possibles sont précalculées par la fonction $f(d, k)$. k étant une inconnue, toutes les hypothèses de k sont considérées : on a une matrice de dimension $D \times K$ avec K le nombre total de choix possibles pour k de $v_{i,j} = f(d_i, k_j)$ avec $i = 1 \dots D, j = 1 \dots K$.

Ces valeurs sont alors abstraites selon un modèle qui représente la consommation de puissance. En pratique on opte souvent pour un modèle basé sur des poids de Hamming ou la distance de Hamming afin d'obtenir un ensemble d'hypothèses. Finalement, les traces de consommation collectées et les hypothèses sont comparées.

Pour réaliser ces comparaisons une attaque DPA calcule la différence de moyennes de consommation pour chaque hypothèse de sous-clé. On obtient alors une matrice R d'une dimension de $D \times T$ avec D le nombre de blocs et T la longueur de la trace. La clé secrète peut être retrouvée en étudiant la corrélation entre l'hypothèse calculée et la consommation de puissance mesurée. La consommation non liée aux valeurs intermédiaires est considérée comme du bruit blanc (un mélange stable de valeurs aléatoires) que l'attaquant peut éliminer avec un nombre suffisant de traces. A une certaine position ct, la consommation de puissance et l'hypothèses correcte de k sont fortement liées et la valeur est donc maximal dans la matrice : on note cette valeur $r_{ck,ct}$. Un pic de corrélation peut alors être observé avec la bonne hypothèse de clé.

- **Attaque CPA - *Correlation Power Analysis***

  Le coefficient de corrélation est un outil de comparaison, proposé comme une optimisation de l'attaque DPA, utilisé pour améliorer la recherche de liens entre deux données. Karl Pearson a proposé en 1896 une formule mathématique (Équation 1) utilisée dans les attaques différentielles pour rechercher une corrélation mathématique entre les différentes hypothèses émises et la consommation de puissance mesurée. Le résultat de ce calcul est une valeur comprise entre -1 et 1.

  Le résultat est dit *fortement corrélé,* s'il se rapproche de 1 ou -1 et inversement faiblement corrélé s'il se rapproche de 0.

Dans une attaque différentielle, le coefficient de corrélation est utilisé pour chercher la linéarité entre h_i et t_j avec $i = 1 \dots K, j = 1 \dots T$. On obtient alors une matrice R ont les éléments $r_{i,j}$ sont estimés selon D éléments h_i et t_j:

Équation 1 : calcul du coefficient de corrélation de Pearson

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \acute{h}_i) \times (t_{d,j} - \acute{t}_j)}{\sum_{d=1}^{D}(h_{d,i} - \acute{h}_i)^2 \times \sum_{d=1}^{D}(t_{d,j} - \acute{t}_j)^2}$$

Les éléments $r_{i,j}$ représentent l'estimation du coefficient de corrélation, h l'hypothèse et t la trace de consommation de puissance avec d dans $[1 \dots \mathrm{N_d}]$ avec $\mathrm{N_d}$ le nombre de chiffrements, i *dans* $[1 \dots 256]$ le nombre d'hypothèses et j dans $[1 \dots \mathrm{N_j}]$ le nombre de points. En balayant la trace point par point et en calculant le coefficient de corrélation pour toutes les hypothèses de clé, on observe un pic correspondant pour la bonne hypothèse de clé, pour un nombre suffisant de traces.

c. *Modèles de consommation de puissance*

Afin de calculer la valeur théorique de la fuite, on choisit un modèle qui représente la consommation de puissance. Naturellement, plus ce modèle de fuites est précis, plus le nombre de traces nécessaires sera faible pour réussir l'attaque.

- **Le model '*bit*'** : on choisit un bit de la variable intermédiaire *v*, en général le bit de poids faible (*LSB*) puisque c'est celui qui commute le plus $h_{i,j} = LSB(v_{i,j})$.
- **Le model '*poids de Hamming*'** : on prend en compte tous les bits de la variable v et on considère le poids de Hamming (*Hamming Weight*) qui représente le nombre de bit à '1' dans v, $h_{i,j} = HW(v_{i,j})$.
- **Le model '*distance de Hamming*'** : la distance de Hamming (*Hamming Distance*) est le model le plus représentatif de la consommation de puissance. Ce modèle prend en compte la commutation et représente donc la somme des bits qui transitent de '1' à '0' ou de '0' à '1'. Cependant l'utilisation de ce modèle suppose de connaitre le type d'implémentation matérielle que l'on cible, en particulier l'état des registres du circuit.

$$h_{i,j} = HD(v_{i,j}, d_i) = HW(v_{i,j} \oplus d_i)$$

Pour estimer le nombre de traces nécessaires, une règle de calcul a été proposée dans [19]. Une étude statistique est réalisée afin d'estimer le nombre de traces

nécessaires pour distinguer un pic de corrélation pour $\alpha = 0.0001$ le dégrée de confiance :

$$n = 3 + 8 \times \frac{z_{1-\alpha}^2}{ln^2\left(\frac{1+\rho_{ck,ct}}{1-\rho_{ck,ct}}\right)}$$

Avec $\rho_{ck,ct}$ le coefficient de corrélation mesuré et $z_{0.999} = 3.719$.

3. Modèles d'attaque de l'AES

Nous avons vu précédemment les différentes opérations d'un AES. La première opération consiste en une opération XOR entre le *plaintext* P et la clé initiale K : c'est une opération d'*AddRoundKey* $(\mathrm{P} \oplus \mathrm{K})$. Il s'agit là d'une opération linéaire entre le plaintext et la clé constituant un point de fuite. Cependant il n'est pas recommandé de construire une hypothèse en ciblant cette opération car celle-ci engage une faible variation d'un bit de sortie par rapport aux bits d'entrée contrairement à une opération non linéaire.

Après cette première opération *d'AddRoundKey,* vient l'opération de *SubBytes*. C'est une opération non-linéaire, dont la propriété de *diffusion* fait de celle-ci le modèle théorique le plus privilégié pour une attaque. Le principe est de cibler la sortie de la table de substitution $Sbox(P \oplus K)$. En effet, avec un nombre suffisant de traces il est possible de mettre en exergue l'hypothèse correcte de la clé par rapport aux fausses hypothèses, via une attaque différentielle à texte (clair ou chiffré) connu. Dans la littérature il existe trois cibles d'attaque :

- La sortie de la *Sbox* de la première ronde : On se retrouve avec une fonction $\mathrm{f(P, K)}$ avec un seul inconnu (l'octet de la clé secrète), avec P le plaintext et K la clé secrète initiale. Dans cette configuration, la complexité de l'attaque par recherche exhaustive suppose de tester 2^8 (= 256) qui représentent toutes les valeurs possibles d'un octet de la clé K.
- La sortie de la *Sbox* de la dernière ronde : L'opération de *MixColumns* est omise et le chemin inverse peux être emprunté en utilisant le cyphertext $Sbox^{-1}(C \oplus K_{10})$. Là encore on se retrouve avec une fonction f avec une seule inconnue $f(C, K_{10})$, avec C le cyphertext et K_{10} la clé de la dernière ronde, en émettant les 256 hypothèses sur les octets de cette clé, on peut alors remonter à la clé initiale avec l'expansion de clé inverse.

- La sortie de l'opération du *MixColumns* : Ce modèle est plus complexe. En effet, étant donné que chaque octet est dépendant de 4 autres octets, on se retrouve avec une complexité de 2^{32} lors de l'exploration des hypothèses. Il faut noter qu'il existe des travaux visant à réduire le degré de complexité de ce type d'attaque [20] [21].

IV. Protections contre les attaques par canaux cachés

Dans la littérature plusieurs contre-mesures ont été mises en place pour protéger les circuits contre les attaques par canaux cachés, que ce soit sur les technologies employées, sur des implémentations logicielles (microcontrôleur, processeur…) ou accélérateurs matériels dédiés. L'objectif reste dans tous les cas de briser la dépendance qui existe entre la consommation de puissance et les valeurs intermédiaires dépendantes du secret.

En simplifiant un peu ce panorama de contre-mesures, on peut distinguer deux grandes catégories de mécanisme de protection : le masquage et l'obscurcissement. Dans ce chapitre nous présentons rapidement les solutions de l'état de l'art, mais qui ne rentrent pas dans le spectre de nos travaux (modifications technologiques ou solutions logicielles), avant de nous intéresser aux solutions matérielles.

1. Le masquage

Ce type de contremesure consiste à ajouter de l'aléa dans l'algorithme lui-même. Les opérations ne sont pas effectuées directement sur les valeurs intermédiaires, mais sur des valeurs dites « masquées » permettant de ne pas corréler la consommation de puissance aux valeurs intermédiaires manipulées. La valeur masquée v_m est le résultat d'une opération entre la valeur intermédiaire v et le masque qui est une valeur générée aléatoirement en interne et inconnue de l'attaquant. Cette opération peut être soit un masquage booléen (avec un XOR) $v_m = v \oplus m$, soit un masquage arithmétique de type multiplication modulaire $v_m = v * m$ (ou addition modulaire $v_m = v + m$). Il est important de noter que la valeur intermédiaire est masquée tout au long de l'algorithme et le masque m est retiré à la fin du calcul afin de retrouver le texte chiffré correct.

Dans le cas d'un algorithme qui utilise des fonctions booléennes et arithmétiques, les deux types de masquage sont utilisés. Cela permet de basculer entre le masquage booléen et arithmétique de façon à ce que le calcul ne crée pas de vulnérabilités

pouvant être exploitées via une attaque par canaux cachés (une méthode de conversion sécurisée a été proposée par Goubin [29]).

Dans l'état de l'art, le masquage a été proposé à la fois en logiciel et en matériel. Ce dernier permet de garantir un bon niveau de sécurité face aux attaques DPA de premier ordre, s'il est correctement mis en œuvre [30]. Cependant il s'avère être vulnérable face aux attaques d'ordre supérieur[1] HODPA (High Order DPA) [31]. Des contremesures intégrant des techniques de masquages d'ordre supérieur ont donc été proposées dans la littérature [33] [34] [35].

Bien que le masquage soit une contremesure très puissante, le surcoût en termes de performance est généralement élevé. La latence d'un AES masqué est deux fois supérieure à celle d'une implémentation AES similaire non protégée [19]. Dans une implémentation matérielle, le coût en surface est tout aussi important. Par exemple, dans le travail proposé par Regazzoni et al. [36], le surcoût de l'implémentation masquée d'un AES est 3 fois supérieur en termes de slices et subit une dégradation des performances temporelles d'un facteur 2,5 par rapport à une architecture non protégée de référence.

La section suivante présente des contre-mesures basées sur des méthodes d'obscurcissement.

2. Les méthodes d'obscurcissement (par réduction du signal)

Ce type de contre-mesures cherche à limiter la dépendance entre la consommation de puissance et les calculs. Pour cela il est possible de réduire le rapport signal sur bruit (Signal to Noise Ratio), avec la puissance du signal exploitable et Pb la puissance du bruit. Il existe donc deux options : réduire le signal ou augmenter le bruit.

Idéalement on devrait arriver à un SNR égal à zéro. Cependant il n'existe aucune contre-mesure capable d'atteindre ce but. Il faut toutefois garder à l'esprit que toute réduction du SNR entraine mécaniquement une réduction de la valeur du coefficient de corrélation lors d'une attaque. Cela a donc un impact non négligeable sur le nombre de traces nécessaires pour casser l'algorithme de chiffrement (c.-à-d. pour en retrouver la clé secrète).

[1] Une attaque est dite de premier ordre quand une seule valeur intermédiaire est exploitée, et de second ordre lorsque deux valeurs intermédiaires sont exploitées. Il peut s'agir soit de deux valeurs intermédiaires utilisant le même masque ou bien le masque et la valeur masquée.

Une première stratégie vise à augmenter le bruit dans l'architecture. Cela peut se faire par l'ajout d'aléa. Cette stratégie, sur laquelle sont basés ces travaux, sera présentée en détails dans le chapitre suivant.

Une seconde stratégie vise à réduire le signal en rendant la consommation de puissance uniforme [37] pour supprimer toute corrélation potentielle avec les calculs réalisés. Les sous-sections suivantes fournissent des détails relatifs aux méthodes associées : transistor, logique et logiciel.

a. Réduction du signal au niveau transistor

Ce type de contre-mesure utilise des approches analogiques qui ne dépendent pas du système cryptographique. L'architecture du circuit ne change pas et la contre-mesure repose sur un composant additionnel chargé de réguler la consommation de puissance à l'extérieur du circuit de chiffrement.

Une première approche proposée dans [38] décrit dans la Figure 10**Erreur ! Source du renvoi introuvable.** consiste à isoler le circuit en utilisant deux capacités qui sont alternées, quand l'une est chargée par l'alimentation externe l'autre est déchargée pour alimenter le circuit cryptographique ainsi isolé (cf. Figure 10). Il est important de noter que cette approche permet de fournir une protection uniquement face à une mesure externe de la puissance consommée. Cependant, le circuit reste vulnérable face à une mesure du champ électromagnétique.

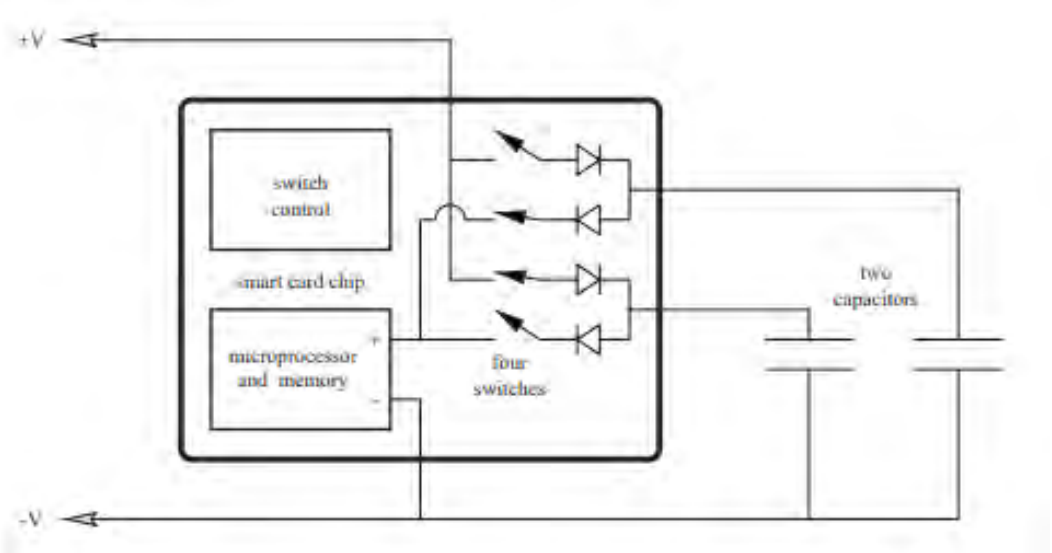


Figure 10: Architecture proposée dans [38]

Une deuxième approche a été proposée par Mesquita *et al.* [39]. Elle consiste à utiliser une technique de contre-réaction pour uniformiser la consommation de puissance. Le circuit est illustré dans la Figure 11. Dans un premier temps, le pic de consommation

du circuit cryptographique (*CC*) est identifié, puis le masque de génération de courant (*CMG*) est chargé de réguler le circuit à la valeur maximale. Il se compose d'un miroir de courant et d'un suiveur de tension utilisé comme un retour de courant. Par conséquent, il induit une hausse non négligeable de la consommation d'énergie, qui est prohibitive dans un système embarqué.

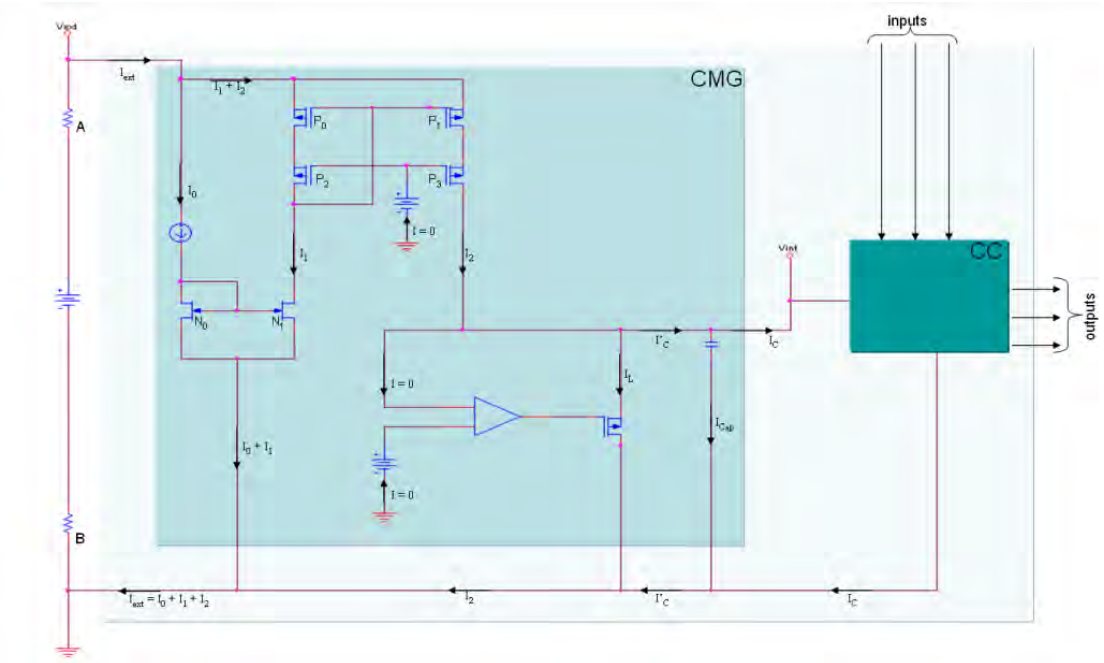


Figure 11: Architecture à contre-réaction

b. *Réduction du signal au niveau logique*

Durant les transitions d'état ('0' à '1' et '1' à '0'), les circuits CMOS consomment de l'énergie en lien avec les données manipulées. Afin d'avoir une consommation de puissance fixe, il faut s'assurer d'avoir une charge fixe. Pour cela l'approche utilisée dans l'état de l'art consiste à avoir le même nombre de transitions de 0 à 1 et de 1 à 0 au sein du circuit. Cette logique s'appelle le DPL - *Dual Rail with Precharge Logic*. Son fonctionnement repose sur deux phases,

- *La pré-charge* : dans cette phase tous les signaux sont réinitialisés à un état connu.
- *L'évaluation* : dans cette phase les calculs sont réalisés avec le même nombre de transitions.

L'une des approches les plus simples est la WDDL - *Wave Dynamic Differential Logic* proposée par Tiri et al. [37]. Cette approche peut être réalisée en réutilisant des libraires de cellules existantes. Dans un premier temps il y a une phase de pré-charge

pendant laquelle tous les signaux sont mis à l'état '0' ; cet état est propagé jusqu'aux sorties. S'en suit une phase d'évaluation constituée de deux circuits implémentant deux fonctions : une fonction vraie et une fonction fausse (le complément logique de la fonction vraie) comme illustrée dans la *Figure 12*.

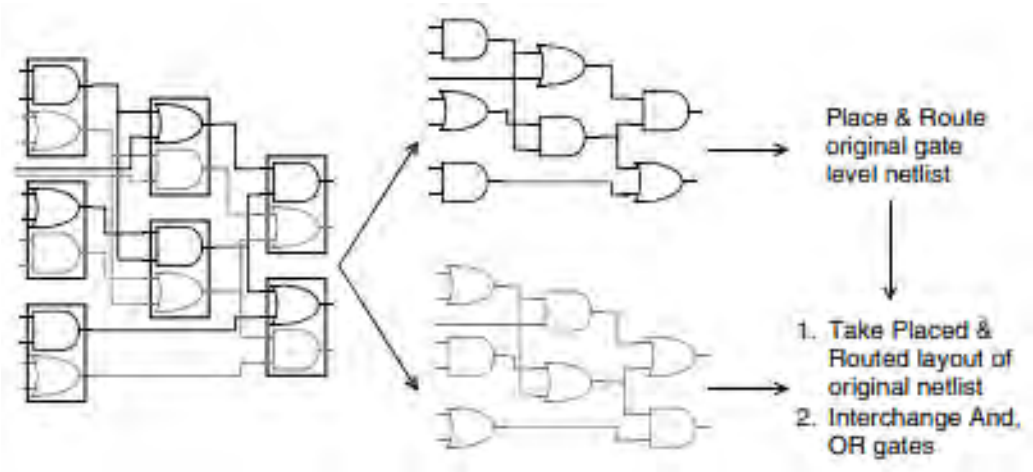


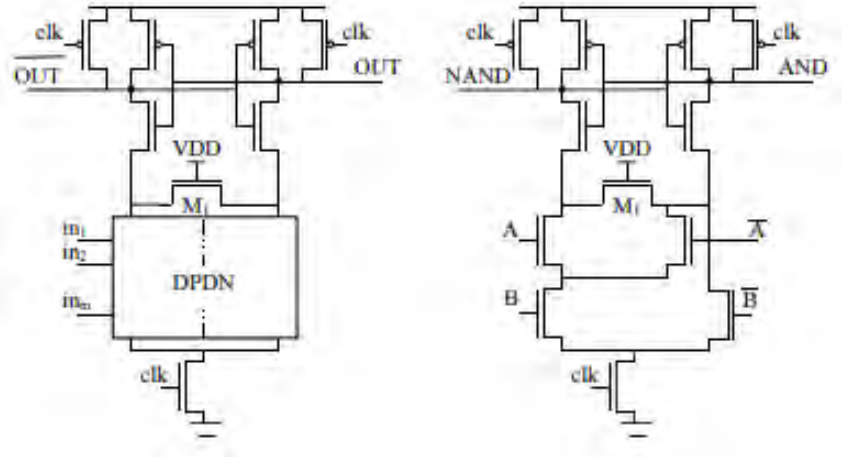
Figure 12: Architecture WDDL

Cependant, du fait des variations liées au processus de fabrication des circuits, il existe des différences de temps de propagation entre les chemins vrai et faux, menant à des *glitchs* (variations de courant transitoires) qui peuvent être exploités par un attaquant. Ce type de problèmes est d'autant plus critique dans une implémentation sur FPGA, en raison de leur architecture fixe et du manque de liberté dans les outils de conception.

Pour tenter d'y répondre Guilley et al. [40] ont proposé le *WDDL+ power constant logic* qui n'utilise que des fonctions combinatoires positives (AND, OR) au lieu des négative (NAND, NOR) afin d'éviter l'utilisation d'inverseurs. D'autres travaux ont également essayé de corriger ce même problème. On peut citer le BCDL - *Balanced Cell-based Dual-rail Logic* [41] qui propose de synchroniser les variables avant la phase d'évaluation, cela a aussi pour avantage d'être moins complexe et plus rapide.

Une autre approche proposée par Tiri et al. [42] est appelée SABL - *Sense Amplifier Based Logic*. Cette solution nécessite la définition d'une nouvelle librairie de cellules. Elle combine la logique de pré-charge et la logique différentielle, le circuit générique *n-gate* ainsi qu'un exemple d'une porte *AND-NAND* est illustré dans la *Figure 13*. Ainsi, il y a une transition par cycle, et à chaque transition le circuit charge et décharge une valeur fixe des capacités, indépendamment de la valeur d'entrée. La phase

d'évaluation est assurée à l'aide d'un système de DPDN - *Differential Pull-Down Network* constitué de transistor NMOS.



(a) (b)

Figure 13: Logique SABL : (a) n-gate générique, (b) porte AND-NAND

Bien que ces solutions offrent un moyen de réduire le SNR efficacement, elles imposent un surcoût important. Dans le travail proposé par [41], où une technologie BCDL est implémentée sur une FPGA Altera Statix X, le surcoût en surface est évalué à 4 fois le nombre registres et de mémoire RAM par rapport à la version d'origine, et 1,7 fois plus d'ALM - *Adaptaive Logic Module* (un composant de base des FPGA Altera Stratix X). Concernant les performances, le débit est 1,8 fois inférieur à celui de l'implémentation AES non protégée. De plus, l'équilibre des deux rails n'est pas toujours garanti [43].

c. Solution logicielle pour la réduction du signal

La stratégie d'uniformisation de consommation de puissance est généralement utilisée en matériel, néanmoins dans [44] [45] les auteurs ont proposé une implémentation logicielle de cette contre-mesure sur un processeur avec un jeu d'instructions PISA - *Portable Instruction Set Architecture*, en exploitant deux cœurs du processeur. Quand le premier cœur exécute l'algorithme de chiffrement DES, le second cœur exécute la version complémentaire de l'algorithme en parallèle. La contre-mesure a été évaluée avec une attaque DPA sur des traces simulées avec l'outil *PrimePower* et les résultats ont démontré l'échec de l'attaque sur le système. Cependant, la réservation d'un cœur complet pour la protection est un coût qui ne peut être supporté par la majorité des applications, particulièrement dans le cadre des nœuds IoT.

V. Conclusion

À l'heure actuelle, la sécurité des informations est une garantie de confiance dans la technologie. Malgré la mise en place de mécanismes de protection « cryptographique », considérée comme sûre mathématiquement, celle-ci se retrouve menacée par des fuites physiques. La sécurisation de ces implémentations est nécessaire. Une implémentation non protégée peut être attaquée avec un faible nombre de traces (de l'ordre de quelques dizaines pour un AES s'exécutant sur un CPU). La robustesse du système face aux attaques dépend alors de plusieurs facteurs : cible, chemin de donnée, mode de chiffrement…

Il existe ainsi de nombreux travaux proposant des contre-mesures ayant pour rôle de complexifier ou d'empêcher ces attaques. Certaines contre-mesures sont très puissantes et coûteuses et peuvent multiplier par trois ou quatre la surface d'un composant matériel ce qui, dans un contexte IoT, est par essence même inapproprié. En conséquence, il faut s'intéresser à des contre-mesures moins coûteuses, mais qui permettent néanmoins d'apporter un degré de sécurité acceptable pour l'utilisateur final.

Dans le chapitre suivant, après avoir présenté les principales architectures matérielles dédiées au chiffrement AES, nous étudions l'état de l'art des protections par ajout d'aléa pour déjouer des attaques par canaux cachés.

Chapitre 2 Chiffrement AES

Dans ce chapitre, nous présentons dans un premier temps les différentes architectures matérielles de référence pour l'AES. Dans un deuxième temps, nous présentons les différentes contre-mesures de l'état de l'art permettant de protéger les opérations de chiffrement symétrique contre les attaques par observation dans sur une cible matérielle.

I. Architecture matérielle d'un AES

Il existe plusieurs solutions d'implantation matérielles qui peuvent varier selon les priorités des concepteurs (débit, surface, consommation d'énergie). Pour atteindre des débits élevés, le choix se porte généralement sur une architecture AES avec un chemin de données de 128 bits. Par ailleurs, avec une architecture entièrement pipelinée, il est possible d'atteindre de très haut débits (jusqu'à 21Go/s) [22]. Ce type d'implémentation est en général réservé pour des appareils haut de gamme ou pour des lignes de communication très haut débit. Dans le but de supporter un meilleur compromis débit/surface, des architectures 32, 16 ou 8 bits sont proposées. Ainsi, une architecture AES basée sur un chemin de données de 32 bits [23], offrant un débit logiquement inférieur à celui d'un AES possédant un chemin de données de 128 bits, se révèle par exemple intéressant pour des applications dans certains réseaux sans fil bas débits (Sigfox, LoRA…). Une architecture AES avec un chemin de données de 8 bits sera retenue dans des applications telles que les étiquettes RFID qui ont quant à elles pour priorité de minimiser les ressources utilisées ainsi que l'énergie consommée [24].

Dans l'état de l'art, plusieurs travaux présentent des architectures matérielles implémentées sur ASIC ou FPGA. Chaque architecture a ses atouts et ses inconvénients, au-delà même des optimisations spécifiques liées à la technologie ciblée. Concernant le stockage des données, celles-ci peuvent être mémorisées dans des registres, dans un ou plusieurs bancs mémoires ou encore dans des registres barillets (SRL - *Shift Register Look-up table*) sur FPGA. L'opération de SubBytes peut être soit pré-calculée et mise en œuvre dans une *Look-Up-Table* sur cible FPGA, soit implémentée en logique, les mémoires étant plus coûteuses pour des implémentations ASIC. L'opération de *Key Scheduling* (expansion de la clé) peut également être soit pré-calculée et stockée dans une mémoire, soit être calculée à l'exécution.

Dans la suite de cette section, nous recensons les principaux travaux proposés dans la littérature. Nous détaillons ensuite les implémentations séquentielles d'un AES avec chemin de données de 8 bits, solutions qui sont privilégiées dans un contexte IoT en raison de leur faible consommation d'énergie et de leur surface réduite.

1. AES-128 avec un chemin de données de 128 bits

Dans une architecture « naïve » d'un AES sur 128 bits, les 16 octets sont traités en parallèle en un seul cycle horloge. Un total de 10 cycles d'horloge est alors nécessaire pour réaliser un chiffrement complet. Ce type d'architecture peut être optimisé :

- En augmentant le débit en jouant sur la fréquence maximum,
- En utilisant un pipeline interne dans la boucle de calcul,
- En dupliquant les rondes pour calculer un chiffrement à chaque cycle d'horloge.

Dans [22], Hodjat *et al*. ont proposé une architecture entièrement pipelinée implantée sur une carte Xilinx Virtex II-Pro. Leur architecture est présentée dans la Figure 14.
Les rondes de l'AES sont pipelinées, le circuit de la ronde est dupliqué 10 fois et chaque ronde est elle-même pipelinée (un registre est alors ajouté après chaque opération). De plus, l'architecture matérielle de la fonction SubBytes est également pipelinée (à cause du délai de propagation des données). Les résultats ont montré que les architectures les plus efficaces (débit/surface) se basent sur des combinaisons de pipeline inter- et intra-ronde : l'implémentation avec le plus haut débit est celle avec sept étages de sous-pipeline dont six dans l'architecture du SubBytes et un après chaque ronde qui suit les opérations (MixColumns, AddRoundKey et ShiftRows).
Cette architecture supporte un débit de 21,64 Gbits/s sans utilisation de BRAM, avec une latence de 71 cycles d'horloge par chiffrement, en utilisant 6400 slices. Une seconde architecture avec quatre étages de pipeline dans la boucle de calcul d'une ronde, dont trois dans l'architecture du SubBytes et un étage après chaque ronde, sans l'utilisation de BRAM, atteint un débit de 21,54 Gbits/s avec une latence de 31 cycles par chiffrement, implémentée sur 5177 slices.

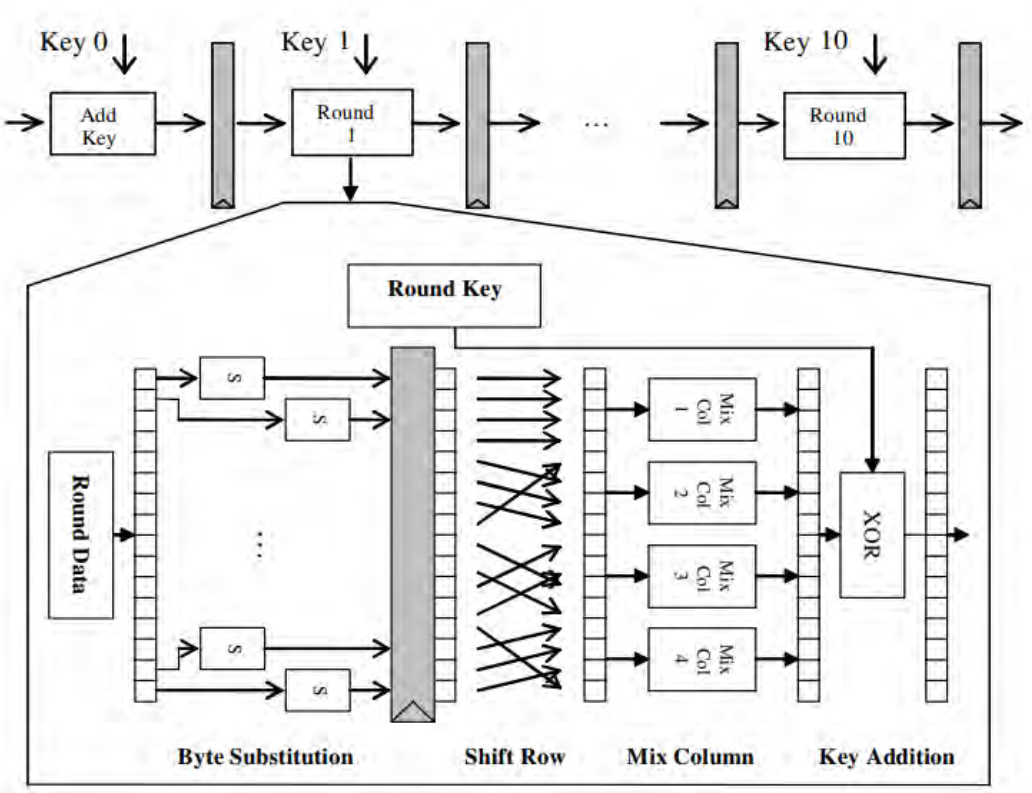


Figure 14: Architecture AES-128 entièrement pipelinée proposée dans [22]

2. AES-128 avec un chemin de données de 32 bits

Cette architecture est un compromis entre le débit et la surface, avec un chemin de données de 32 bits. L'architecture proposée par Kris Gaj [23] comporte un bloc de chiffrement et un bloc de déchiffrement. Chaque ronde est calculée en 4 cycles d'horloge, les clés de ronde sont pré-calculées et stockées dans une mémoire avec une latence de 44 cycles avant le début du chiffrement. La matrice d'état se compose de 16 octets agencés dans une matrice 4x4. Les données sont organisées par colonne de la matrice d'état au sein d'une mémoire. L'opération ShiftRows est assurée par le bon adressage des octets en mémoire, comme illustré dans la Figure 15.

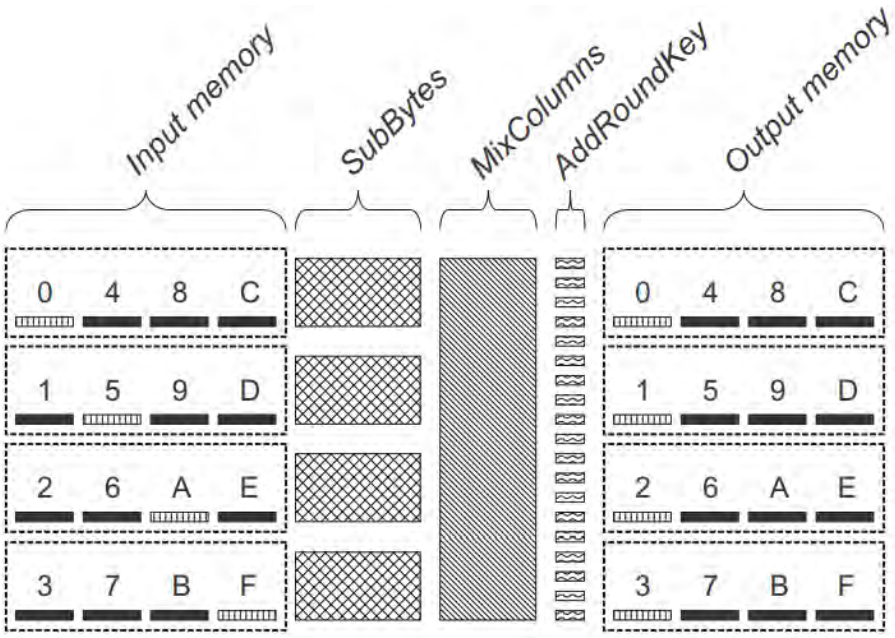


Figure 15 : Disposition des données

Pour calculer la première colonne, les octets 0, 5, A et F sont lus en parallèle. L'opération du SubBytes est alors réalisée suivie des opérations MixColumns et AddRoundKey avec la clé de ronde courante. Enfin, les octets sont stockés dans la mémoire de sortie aux adresses 0, 1, 2 et 3. De la même façon, au second cycle d'horloge, la deuxième colonne est calculée. Les octets adressés sont : 4, 9, E et 3 et les résultats sont alors stockés aux adresses 4, 5, 6 et 7 de la mémoire de sortie. Puis la troisième colonne est calculée au cycle suivant en adressant les octets 8, D, 2 et 7 dont les résultats seront stockés aux adresses de la mémoire de sortie 8, 0, A et B. Au quatrième cycle, la quatrième colonne est calculée en adressant les octets C, 1, 6 et B, puis les résultats sont écrits aux adresses C, D, E et F de la mémoire de sortie. À partir de là, cette mémoire devient la mémoire d'entrée de l'AES, et l'ancienne mémoire d'entrée devient la mémoire de sortie.

Une optimisation a été ajoutée dans le chemin de données du déchiffrement : les coefficients de multiplication dans la matrice du MixColumns inverse $d(x)$ sont plus complexes et occupent une plus large surface. Une alternative a alors été implémentée en se basant sur une méthode proposée par Satoh [25]. L'idée consiste à partager les ressources du MixColumns $C(x)$, et à multiplier le résultat par un polynôme d^2 tel que :

$$C(x) \times d^2(x) = d(x).$$

Pendant le chiffrement, les données transitent par le module SubBytes puis MixColumns et enfin AddRoundKey. Pendant le déchiffrement, les données transitent par le module SubBytes inverse, AddRoundKey et enfin par le module MixColumns inverse qui se compose du sous-module $c(x)$ et $d^2(x)$. En effet, l'opération AddRoundKey précède le module de MixColumns_Inverse afin d'éviter un multiplexeur supplémentaire. Durant la dernière ronde, l'opération du MixColumns / MixColumns_Inverse est omise.

L'architecture a été implémentée sur une carte Xilinx Spartan II *XC2S30* avec 222 slices et 3 blocs RAM en utilisant Xilinx ISE 5.2i avec une fréquence maximale de 60Mhz et un débit de 166 Mbit/s.

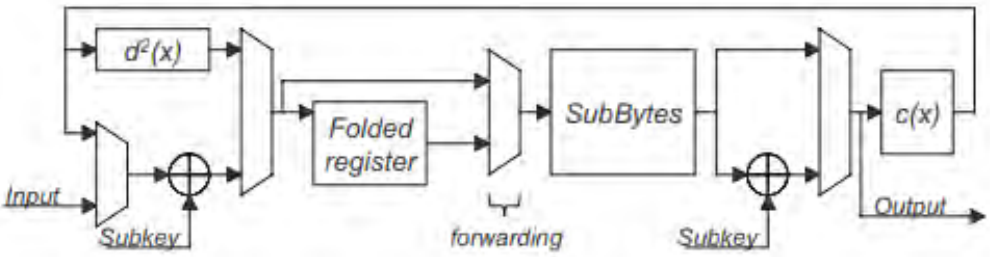


Figure 16: Architecture AES-128 avec chemin de données 32 bits [23]

3. AES-128 avec un chemin de données de 8 bits

Les implémentations sur un chemin de données de 8 bits sont séquentielles et traitent les données octet par octet. Elles sont conçues pour des applications à faible coût n'ayant pas pour priorité les performances, mais plutôt la réduction de la consommation d'énergie et de la surface. Dans l'état de l'art, l'architecture proposant le meilleur rapport débit/surface que nous ayons trouvée est celle proposée par Hamalainen *et al.* [24]. L'implémentation, ciblant une technologie ASIC, est décrite dans la Figure 17.

Le chiffrement complet d'un bloc se fait en 160 cycles d'horloge. Les données sont stockées dans un registre à décalage et leur « rotation » est assurée par des multiplexeurs, permettant la réalisation du ShiftRows.

Dans un premier temps, les données d'entrée arrivent octet par octet et passent par les quatre registres du MixColumns ; elles subissent alors une opération de XOR avec la clé initiale qui est décalée de quatre cycles en sortie du module d'expansion de la clé. Le résultat du Key Whitening est alors transféré au module de ShiftRows (composé de registres et de multiplexeurs). L'opération de ShiftRows est assurée par le bon adressage des bits de contrôle des multiplexeurs. Ensuite, l'opération de SubBytes

implémentée en logique est réalisée. Puis l'opération de MixColumns est effectuée : pour cela, chaque colonne de l'état intermédiaire est calculée en quatre cycles. Les octets de la colonne sont multipliés par les coefficients $\{02\}$ et $\{03\}$ puis stockés dans quatre registres et enfin accumulés avec le résultat précédant. Au quatrième cycle on obtient le résultat des quatre octets de la colonne. Ce résultat est alors décalé à l'aide d'un barillet (*Barrel Shifter*), puis une opération de XOR réalisée entre ce résultat et la clé de ronde en quatre cycles. Cette clé de ronde est elle-même calculée en parallèle par le module d'expansion de clé qui comporte deux sorties décalées de quatre cycles.

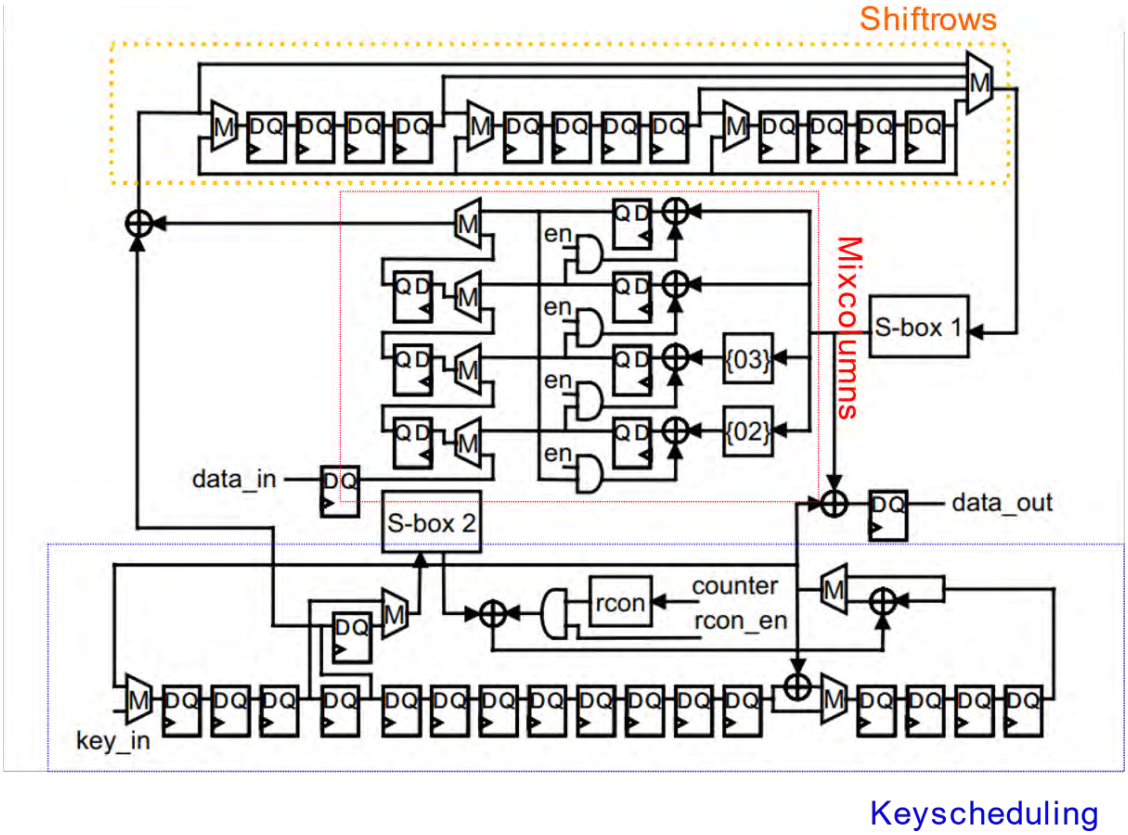


Figure 17: Architecture AES-128 avec un chemin de données de 8 bits [24]

Les résultats en surface, consommation de puissance et performance sont présentés pour différentes options de synthèse (optimisation en surface, performance et puissance). Pour une implémentation optimisée en surface, le nombre de portes équivalentes est 3100, le débit est de 121 Mb/s et l'énergie consommée est de 37 µW/Mhz. Pour une optimisation en puissance le nombre de portes équivalentes est de 3200, le débit est 104 Mb/s et l'énergie consommée est 30 µW/Mhz. Pour une optimisation en performance le nombre de porte équivalant est 3900, le débit est 232 Mb/s et l'énergie consommée est 62 µW/Mhz.

L'architecture a été adaptée pour un FPGA sans bloc RAM dans [26] et les registres sont remplacés par des SRLs. Cette approche a été mise en œuvre sur une carte FPGA XILINX SPARTAN-6 en utilisant 80 slices, pour un débit de 58.13Mbts.

Le module ShiftRows, décrit dans Figure 18, a été adapté pour une implémentation FPGA où des SRLs sont utilisées à la place des registres. Les données sont décalées dans deux SRLs connectées et qui partagent les mêmes adresses. Le résultat de l'opération se trouve en sortie des deux multiplexeurs. Les bits de contrôle sont intégrés au mot d'adresse permettant la réalisation du ShiftRows par la génération des adresses.

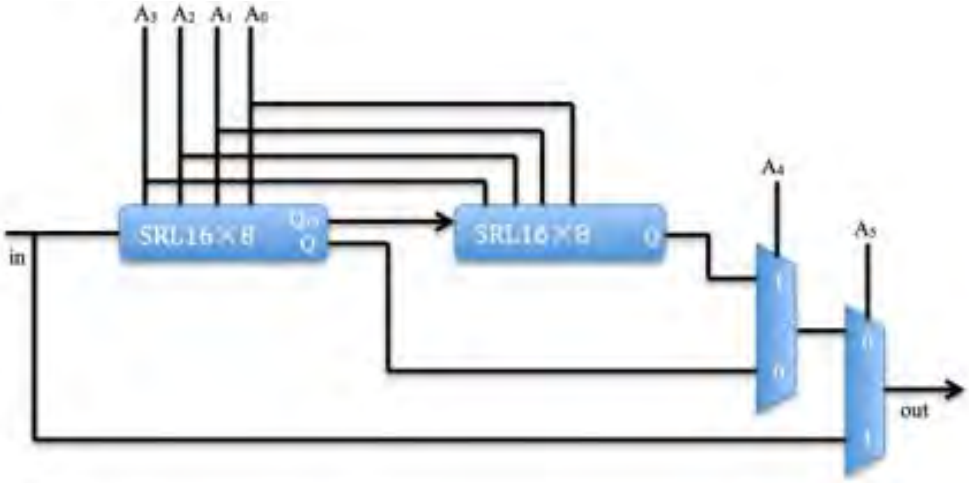


Figure 18: Architecture du ShiftRows proposé dans [26]

Dans le travail proposé par Sasdrich *et al.* [27], une autre architecture d'un AES basé sur un chemin de 8 bits est présentée (voir Figure 19). Celle-ci est conçue et optimisée pour une carte FPGA XILINX SPARTAN-6. L'architecture finale occupe seulement 21 slices, la latence est de 1471 cycles, avec un débit de 9.12 Mb/s.

Les adresses de stockage ont été adaptées à l'opération de ShiftRows. L'opération de SubBytes est mise en œuvre à l'aide de mémoires distribuées en utilisant 8 slices contenant des LUT à 6 entrées.

Chaque slice réalise une fonction MUX 8 vers 1 en utilisant des multiplexeurs internes : on a alors besoin des Slice-L ou Slice-M d'un FPGA qui possède des capacités spéciales pour effectuer des opérations arithmétiques et logiques. En effet les slice-L possèdent des multiplexeurs internes pouvant être combinés avec les LUTs. Les Slices-M quant à eux, permettent la réalisation de mémoires distribuées via la configuration des LUT en mémoire de 256 bits ou en registres à décalage de 128 bits.

Dans cette architecture, chaque octet est traité en quatre cycles dans l'opération du MixColumns (intégrant l'opération d'AddRoundKey), contrairement à l'architecture de

référence où les quatre octets de la colonne sont calculés en quatre cycles. Pour ce faire, un registre 8 bits est utilisé pour l'accumulation des valeurs intermédiaires x afin de réaliser quatre opérations :

- $r_i = x_i$ (Écriture de x dans le registre r)
- $r_i = r_{i-1} + \{01\} \times x_i$· xi ($x$ est multiplié par $\{01\}$ et accumulé avec la valeur précédente)
- $r_i = r_{i-1} + \{02\} \times x_i$ (x est multiplié par $\{02\}$ et accumulé avec la valeur précédente)
- $r_i = r_{i-1} + \{03\} \times x_i$ (x est multiplié par $\{03\}$ et accumulé avec la valeur précédente)

Les 16 octets sont adressés selon l'opération courante en utilisant 4 bits : 2 bits pour sélectionner la colonne de la matrice d'état à traiter et 2 bits pour sélectionner un octet dans la colonne courante. Les octets sont alors multipliés par le coefficient $\{01\}$, $\{02\}$ et $\{03\}$ et accumulés dans le registre r_i. Ce dernier est initialisé avec la valeur de la clé de ronde courante avant le début de l'opération du MixColumns. Les résultats de l'accumulation des multiplications dans l'espace de Galois incluent le résultat des deux opérations MixColumns et AddRoundKey.

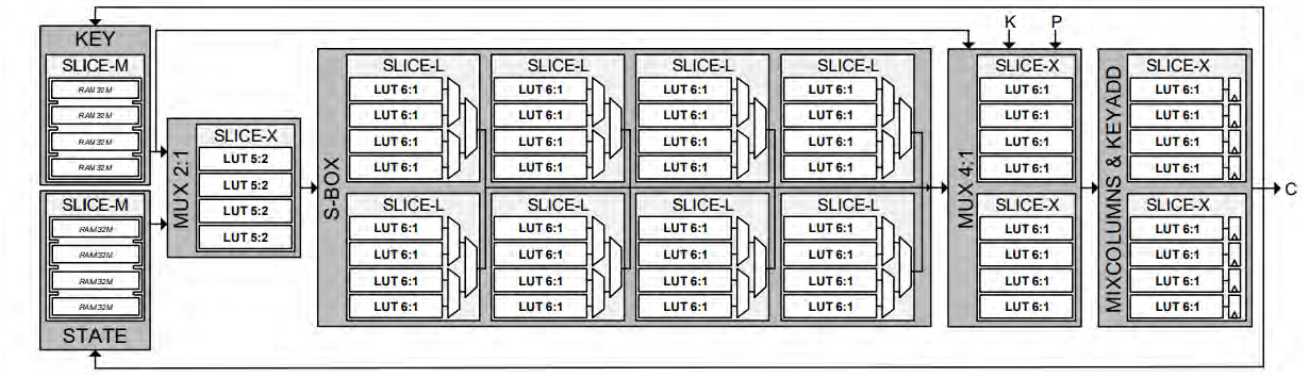


Figure 19 : Implémentation proposée dans [27]

Dhnuskodi *et al.* [28] ont proposé une architecture ASIC, dont l'objectif est de réduire l'énergie dissipée par l'architecture proposée dans [24] et décrite précédemment. Pour ce faire, la conception de leur architecture cherche à minimiser des activités de « transitions » inutiles aux calculs du fait du décalage des données dans les registres (induit par l'architecture à base de registres à décalage proposée par Hamalainen *et al.* [24]).

Avec un surcoût en surface raisonnable (45%) par rapport à [24], cette approche a permis de réduire l'énergie dissipée par l'horloge de 70%, et celle du chiffrement par

45%. Cette architecture se compose de 16 registres de 8 bits P_i qui stockent les 128 bits de la matrice d'état. Les 16 octets B_i correspondent aux données et spécifient les correspondances des octets et des registres. En effet les 16 octets sont mis à jour à chaque ronde de l'AES. L'opération de ShiftRows est alors assurée par un multiplexeur contrôlé par l'unité *Enable Generator* chargée d'assurer le bon adressage des octets. Les registres sont alors arrangés dans quatre groupes comme décrits dans la Figure 20. La latence d'un chiffrement est de 160 cycles d'horloge.

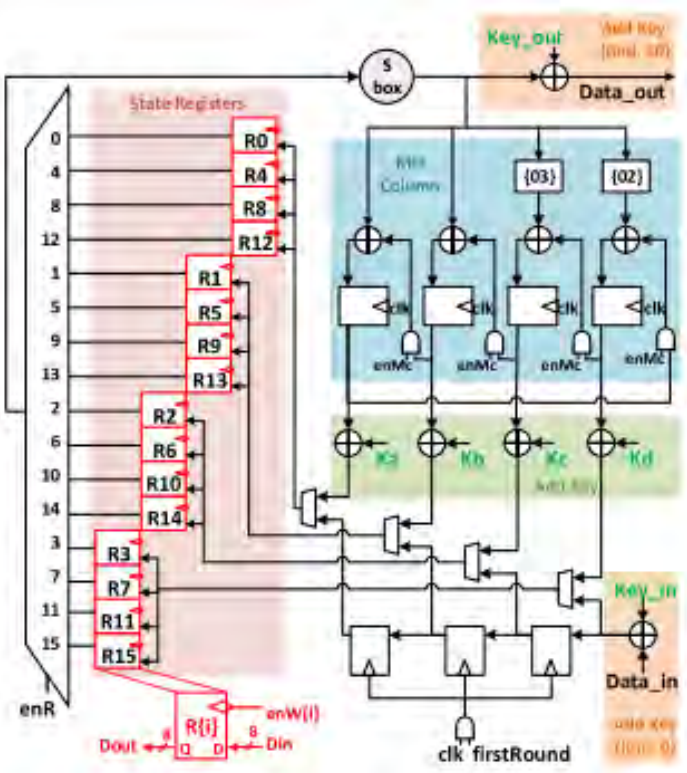


Figure 20: Chemin de données dans [28]

4. Bilan partiel

Dans le cadre de cette thèse, nous cherchons à améliorer la sécurité des petits objets connectés dans un contexte IoT pour lesquels une implémentation matérielle d'un AES se doit d'être compacte, avec une faible emprunte énergétique. C'est pourquoi nos travaux se concentrent sur des implémentations séquentielles 8 bits.

Les implémentations AES non protégées précédemment présentées sont comparées dans le Tableau 1. L'implémentation la plus efficace en termes de surface et de débit en technologie CMOS est [24] alors que celle ciblant la technologie FPGA est [26]. La priorité dans le travail proposé dans [27] est la surface aux dépens des performances, quant au travail proposé dans [28] l'enjeu majeur est de réduire l'énergie dissipée.

Tableau 1 : Comparaison Surface/Performance/Energie des implémentations AES non protégées

Référence	[22]	[23]	[24] (Surface)	[24] (Performance)	[24] (Puissance)	[26]	[27]	[28]
Technologie	Virtex II-Pro	XC2S30-6	0,16µm CMOS	0,16µm CMOS	0,16µm CMOS	XC6SLX4	XC6SLX4	45nm CMOS
Surface	5177 (Slices)	222 (Slices)	3100 (Portes)	3900 (Portes)	3200 (Portes)	80 (Slices)	21 (Slices)	4009,4 (µm²)
BRAM	84	3				0		
Débit	21.51 Gbits/s	166 Mbits	121	232	104	58,13 (Mbits)	912 (Mbits)	403
Latence	31	46	160	160	160	160	1471	160
Fréquence (Mhz)	168.3	60	152	290	130	72,66	105	510
Puissance	n/a	n/a	37(µW/Mhz)	62(µW/Mhz)	30(µW/Mhz)	n/a	n/a	5,66pJ/bit
Chemin de donnée	128	32	8	8	8	8	8	8

On notera que ces implémentations ne sont pas protégées contre les attaques par canaux cachés. Dans la section suivante, nous étudions les mécanismes de protection associés aux implémentations matérielles ciblant majoritairement l'AES.

II. Protection par ajout d'aléa

Dans cette partie nous présentons les différentes contre-mesures par ajout d'aléa. Il existe beaucoup d'autres types de contre-mesure dans la littérature que nous avons choisi de ne pas présenter ici pour rester concis.

1. Pour les processeurs

a. Ajout d'aléa dans l'ordre d'exécution (*Shuffling*)

Le « *shuffling* » (ou brassage) est l'une des contre-mesures les plus populaires de l'état de l'art. L'idée est d'exécuter le code de façon différente à chaque exécution, afin d'avoir une trace de consommation de puissance différente à chaque fois.

Pour ce faire, une première approche consiste à exécuter les instructions indépendantes séquentiellement et dans un ordre aléatoire. Plusieurs méthodes ont été proposées dont la première est une approche dynamique non liée à l'algorithme, proposées par May et al. [50]. Dans ces travaux, un processeur non déterministe identifie les instructions indépendantes et les exécute dans un ordre aléatoire. Une solution d'implémentation de ce processeur a été proposée : les conflits entre les instructions sont identifiés par un bloc matériel dédié ; on a alors un ensemble d'instructions exécutables dont une est sélectionnée de façon aléatoire. Ce processeur

de type RISC a été émulé au niveau instructions. Dans ce travail l'algorithme utilisé comme cas d'étude est l'algorithme DES. Pour l'évaluation de sécurité, une attaque DPA a été réalisée sur la version standard et la version non déterministe du processeur en utilisant des traces simulées de consommation de puissance sur la base d'un modèle en poids de Hamming. Le nombre d'hypothèses se limite à quatre parmi les 64 hypothèses possibles des 6 bits de la clé. La puissance est alors divisée en puissance statique et dynamique : la puissance statique correspond à la somme des poids de Hamming des instructions adressées et des opérandes ; la puissance dynamique quant à elle correspond aux transitions dans les registres, ALUs... Un pic de corrélation a été distingué en attaquant la version standard du processeur, dans la version non déterministe le pic a disparu. De plus, le nombre de chemins d'exécution a été calculé : le nombre de traces différentes obtenues est 10^{11} pour un algorithme DES.

Irwin et al. [46] ont ajouté un étage de pipeline appelé « mutation unit » qui permet d'amplifier l'aléa en ajoutant des calculs « fictifs ». Une réalisation sur FPGA de ce processeur a été proposée par Grabher et al. [51]. Deux étages de pipeline « *mutation unit* » et « *random issuing* » ont été ajoutés dans un processeur de type RISC entre l'étage de décodage et d'exécution. L'architecture a été mise en œuvre sur une carte Xilinx *XC2VP7-5FG456C*. Le cas d'étude utilisé dans l'évaluation est l'algorithme AES. Une attaque DPA a été réalisée en utilisant des traces de consommation de puissance collectées avec un oscilloscope Tektronix DPO-7104. L'évaluation proposée indique que les performances ont été divisées par 1,75 et que la taille du programme est multipliée par 8. L'AES non protégé a pu être attaqué avec 50 traces, la version avec aléas quant à elle a été attaquée avec 5000 traces, soit un coefficient de sécurité multiplié par 100.

Afin de réduire la complexité de l'architecture, les auteurs de [49] ont proposé une implémentation d'une unité de brassage de données matérielle appelée « *shuffler* » sur une cible FPGA Xilinx Virtex-5. L'ensemble de l'architecture est illustré dans la Figure 21 : l'unité de permutation est située entre le cœur et le cache d'instructions et a pour rôle de générer des permutations utilisées pour ajouter de l'aléa dans l'ordre d'exécution des instructions, en tenant compte des dépendances.

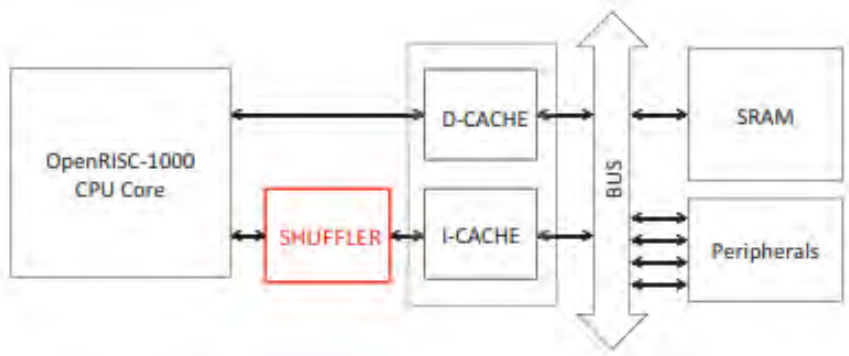


Figure 21 : Implémentation proposée par [49]

Les permutations sont générées en utilisant un réseau de commutateurs proposé par Waksman dans [52] : pour N entrées, $n \times log_2 n - n + 1$ commutateurs sont nécessaires. Les commutateurs permettent deux commutations comme décrit dans la Figure 22 et sont pilotés avec des bits générés aléatoirement par un TRNG, un PRNG ou un composant externe. Ainsi l'aléa est ajouté dans l'ordre d'exécution des N_b entrées qui correspondent aux blocs d'instructions indépendantes contenant N_i instructions.

Le surcoût en surface est de 1,5% en logique séquentielle et de 2,5% en logique combinatoire. La sécurité apportée par le brassage ajouté à la contre-mesure basée sur l'ajout d'instructions "fictives" a été évaluée en utilisant la carte SASEBO [53]. Des attaques CPA et DPA ont été réalisées avec 100 000 traces de consommation enregistrées. Dans la version non protégée, la clé a été révélée avec un coefficient de corrélation de 0,1 en utilisant une attaque DPA et 0.248 avec une attaque CPA, tandis que dans la version protégée la clé n'est pas révélée et le coefficient de corrélation est de 0,013 dans les deux attaques CPA et DPA.

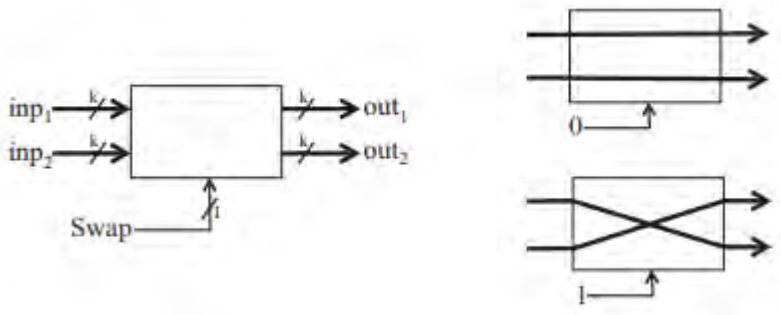


Figure 22 : Commutateur de deux bits proposés dans [49]

En utilisant la règle de base détaillée précédemment pour estimer le nombre de traces minimum nécessaire pour révéler la clé, le MTD - *Measure to Disclosure*, le rapport de sécurité a été estimé à 366 par rapport à la version non protégée.

Dans les travaux de Couroussé *et al.* [54], un environnement est proposé pour générer un code polymorphe à l'exécution. Contrairement à un compilateur classique dont la priorité est d'optimiser les performances et où les instructions sont ordonnancées de sorte à minimiser le temps d'exécution, l'ordonnancement est ici exploité afin de générer diverses variantes du code, sémantiquement équivalentes, à partir d'un code source unique (ces modifications peuvent éventuellement résulter sur une dégradation des performances). Ainsi, chaque fois qu'une nouvelle instruction est ajoutée dans le buffer, une liste de possibilités d'insertion est calculée, puis une position est sélectionnée aléatoirement parmi la liste des créneaux possibles identifiés ; si aucun créneau n'est disponible l'instruction est ajoutée à la fin du buffer d'instruction.

Une autre approche consiste à ajouter l'aléa au niveau algorithmique. Ces travaux sont centrés sur l'évaluation de l'apport de la contre-mesure du brassage en termes de sécurité. Dans une étude menée par Veyrat-Charvillon *et al.* [55], l'aléa est ajouté dans l'ordre d'exécution des 16 opérations indépendantes de l'AES. L'algorithme de génération de permutation a été optimisé pour être capable de générer toutes les permutations en 362 cycles d'horloge.
Trois méthodes ont été mise en œuvre :

- Une version basique avec double indexage : un compteur est utilisé pour indexer le vecteur de permutation, le résultat est alors utilisé pour indexer le vecteur d'opérandes.
- Une version optimisée avec un chemin d'exécution avec aléa consistant à ajouter l'aléa dans le chemin d'exécution. Pour cela, le code assembleur de chaque ronde est divisé en 16 blocs indépendants identifiés par un label. Durant l'exécution, un macro-code est chargé de récupérer la permutation et d'effectuer des sauts d'adresses selon l'ordre de la permutation générée.
- Une version optimisée avec aléa sur le programme en mémoire. L'ajout d'aléa est effectué par la modification des emplacements mémoire des données en exploitant la propriété d'auto-programmation de l'ATMega664P.

L'algorithme a ensuite été implanté sur un microcontrôleur ATMega644P. La version « *Furious* » de l'AES [56] a été mise en œuvre avec les trois méthodes décrites. Dans le cas du double indexage, le besoin en RAM est multiplié par 1,3 et la latence est multipliée par 11. Dans la version optimisée avec un chemin d'exécution avec aléa, le besoin en RAM est multiplié par 2,2 et la latence est multipliée par 2,5. Quant à la

version optimisée avec aléa sur le programme en mémoire, le besoin en RAM est multiplié par 2,7 et la latence est multipliée par 1,2.

La sécurité apportée par le brassage a été évaluée avec des attaques « *template* » [57] en simulation et avec des traces de consommation de puissance mesurées avec un oscilloscope numérique. Les attaques template ont été réalisées avec et sans considération des fuites indirectes liées à la réalisation de la permutation dans le microcode. Les résultats ont montré qu'au maximum le nombre de traces nécessaires pour atteindre un taux de succès de 90% est multiplié par 10 avec une attaque template, sans considérer les fuites dans les permutations. En considérant les permutations, l'apport de sécurité est négligeable.

Plos et al. [58] ont mis au point un algorithme AES avec aléa sur l'ordre d'exécution des opérations sur microcontrôleur ATmega128. Leur but était d'évaluer la sécurité de deux prototypes tag RFID (HF et UHF) dans lesquels un AES avec aléa sur l'ordre de calcul est mis en œuvre. Seuls les résultats en sécurité ont été présentés en considérant deux attaques : la première est une attaque différentielle de fréquence DFA [59], la seconde est une attaque par analyse électromagnétique précédée par un pré-traitement *DEMA* [60] (filtrage analogique et d'intégration d'émanation électromagnétique). Dans chaque expérience 10 000 traces ont été utilisées. Dans le cas du prototype tag-HF des pics de corrélation d'amplitude 0,06 et 0,1 ont été observés avec des attaques *DEMA* et *DFA* respectivement, tandis que dans le cas d'un prototype UHF des pics de 0,23 et 0,14 ont été observés avec des attaques *DEMA* (avec intégration et filtrage) et *DFA* respectivement.

b. Ajout d'aléa dans la sémantique du calcul

Une autre stratégie pour augmenter l'aléa consiste à calculer une même fonction de façons diverses (en utilisant une variété d'algorithmes ou différentes instructions par exemple) sans altérer le résultat final. On parle dans ce cas de polymorphisme, l'une des techniques utilisées en complément de l'ajout d'aléa dans le calcul et de l'ajout d'instructions « fictives », dans le but de dériver des chemins d'exécutions multiples à partir d'une même fonction et obtenir ainsi des traces de consommation différentes.

Dans le travail proposé par May [50], un processeur de type SPARC non déterministe a été émulé au niveau instructions. On retrouve globalement les mêmes principes que

ceux proposés dans [47]. Pour rappel, dans cet article le polymorphisme a été déployé sur une plateforme ARM926 32-bits pour l'exécution d'un algorithme AES. Les traces de consommation de puissance ont été mesurées avec un oscilloscope Agilent *InfiniumDSO80204B*. En attaquant la version non protégée, le MTD est de 11600 traces avec une attaque CPA tandis qu'en attaquant la version protégée la clé n'a pas pu être révélée. Les auteurs n'ont pas jugé nécessaire de collecter des traces supplémentaires au regard du faible coefficient de corrélation obtenu. Par conséquent, avec une baisse des performances de 20%, il a été conclu qu'une protection peut être apportée par un polymorphisme toutes les 2000 à 3000 exécutions. Dans [61], cette contre-mesure a été évaluée dans le cadre d'une attaque exploitant les mémoires caches [62]. Les conclusions des auteurs sont similaires aux travaux présentés dans [47].

Pour le travail présenté dans [54], une étape de sélection des instructions est ajoutée. La sélection est réalisée parmi une liste d'instructions sémantiquement équivalentes. Ensuite, un mécanisme de sélection, piloté par une valeur générée aléatoirement, détermine quelle variante d'instruction va être exécutée. Une évaluation expérimentale a été réalisée sur une plateforme avec un cortex-M3 dans laquelle un AES avec un chemin de données de 8 bits a été mis en œuvre. Les traces de consommation de puissance ont été collectées avec un oscilloscope 2208A PicoScope et une attaque CPA a été réalisée sur les versions non protégée et protégée. Dans la version non protégée, le MTD est de 35 traces, tandis que dans la version protégée un taux de succès de 100% est atteint avec 120 000 traces dans le cas où le polymorphisme est généré à chaque exécution. L'évaluation des auteurs a montré une multiplication par 23,9 du temps d'exécution de l'algorithme.

c. Ajout d'aléa dans le stockage de l'information

Afin d'affaiblir les attaques par observation différentielle, il est possible d'ajouter de l'aléa dans le stockage des données manipulées. May et al. [63] ont étendu leur travail en proposant l'ajout d'aléa dans le stockage avec un renommage aléatoire des registres.

Le renommage des registres est une technique généralement utilisée pour optimiser les performances d'un processeur (exécution *Out-of-Order*). En effet, dans un processeur, chaque registre virtuel est assigné à un registre physique de façon

permanente, en revanche la technique de renommage, permet un changement d'assignation chaque fois qu'un registre virtuel est réécrit. La consommation de puissance est impactée à chaque fois qu'un registre est réécrit étant donné qu'elle est liée aux nombres de bits qui commutent. Dans cette contre-mesure, les registres sont renommés de façon non déterministe, de tel sorte à ajouter de l'aléa dans la consommation de puissance.

Chaque fois qu'une instruction est lue depuis la mémoire, des identifiants virtuels sont assignés aux registres physiques via une table d'assignation. L'architecture est illustrée dans la *Figure 23*. En partant du principe que le nombre de registres physiques est supérieur ou égal aux nombres de registres virtuels, une série de bits est utilisée pour vérifier si le registre est libre ou non. Un registre physique libre est alors assigné aléatoirement. L'ancien registre est marqué comme libre tandis que le nouveau registre est marqué comme utilisé. Dans ces travaux aucune évaluation de sécurité n'est présentée.

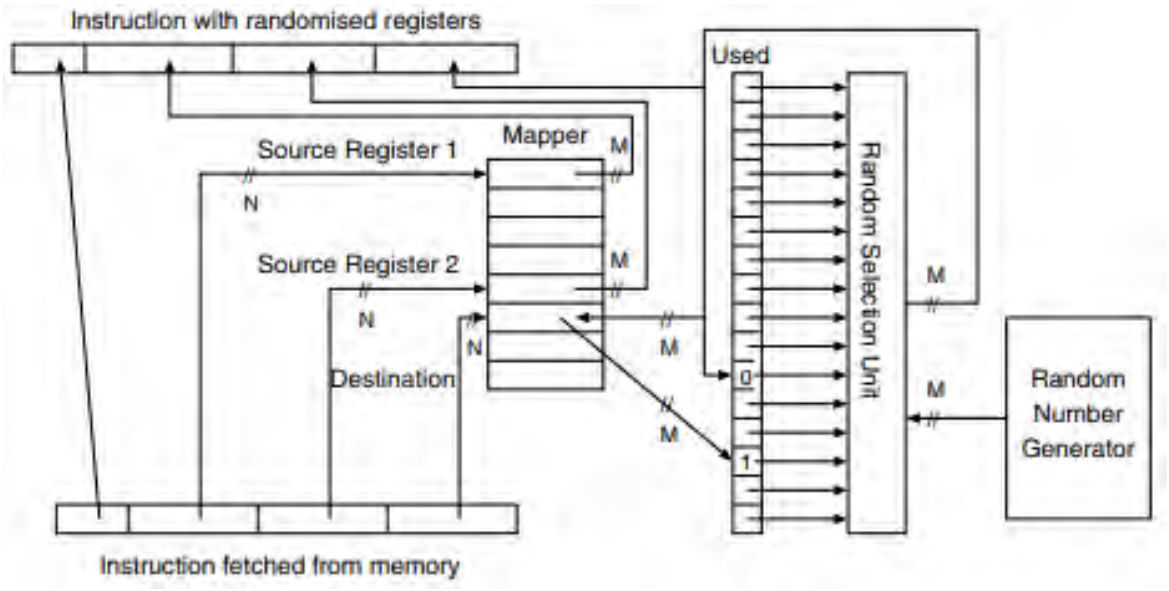


Figure 23: Principe de renommage des registres proposé dans [63]

Une solution alternative moins coûteuse a été proposée par Irwin et al. [46]. Dans cette publication les auteurs présentent une analyse de la durée de vie des variables dans le code (cf. *Figure 24*). Il s'agit d'une table de réassignation utilisée pour gérer au mieux la réutilisation des registres dans l'application.

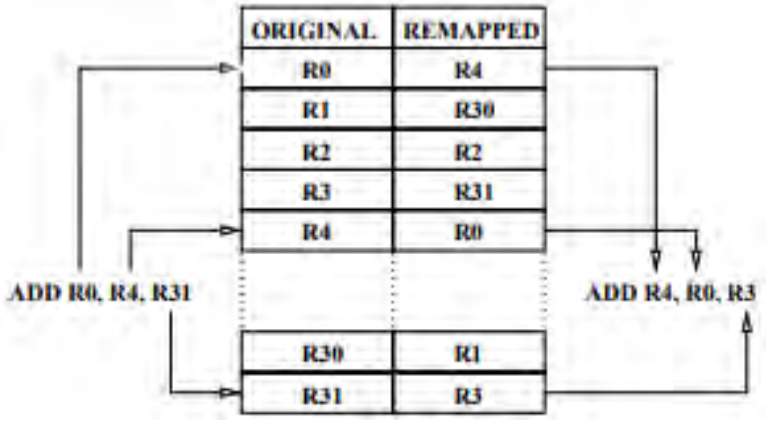


Figure 24: Table de réallocation des registres proposée dans [46]

Un bit est utilisé pour déterminer si le registre est « vivant » (actif, bit a '0') ce qui signifie que la valeur stockée est utile, ou « mort » (inactif, bit a '1') ce qui signifie que la valeur stockée est inutile et que le registre peut être réutilisé. Une table de vie est déduite à partir des résultats d'allocation des registres et de la durée de vie des variables. Cette table est mise à jour après chaque instruction (cf. *Figure 25*). Cette technique est combinée à l'ajout d'instructions fictives et à l'ajout d'aléa dans l'ordre des opérations. L'architecture a été évaluée en utilisant l'algorithme AES comme cas d'étude. Une attaque DPA basée sur la covariance a été réalisée avec 10 000 traces, les résultats ont montré que sans contre-mesure la clé est révélée, tandis qu'avec les contre-mesures elle ne l'est pas.

REGISTER NO.	R0	R1	R2	.....	R29	R30	R31
LIVENESS STATUS	1	0	0	.....	1	1	0

Figure 25: Tableau de durée de vie des variables

2. Pour les accélérateurs matériels non programmables

a. *Ajout de fausses rondes*

Dans [67], Jerabek et al. proposent une approche pour obscurcir le signal dans le temps et l'espace en calculant de fausses rondes. Le cas d'étude utilisé est l'algorithme PRESENT [68] constitué de 32 rondes. Dans ces travaux, l'architecture proposée permet de réaliser au maximum 3 rondes successives dans des blocs dédiés. À chaque cycle d'horloge, un nombre entre 1 et 3 de rondes à calculer est déterminé aléatoirement. Deux des trois blocs auront pour effet de causer du bruit à chaque cycle d'horloge, mais leurs résultats ne seront pas stockés (cf. *Figure 26*). Pour

une moyenne de 2 rondes calculées par cycle, 16 cycles sont requis pour le chiffrement et un surcoût maximal en surface est de 200%.

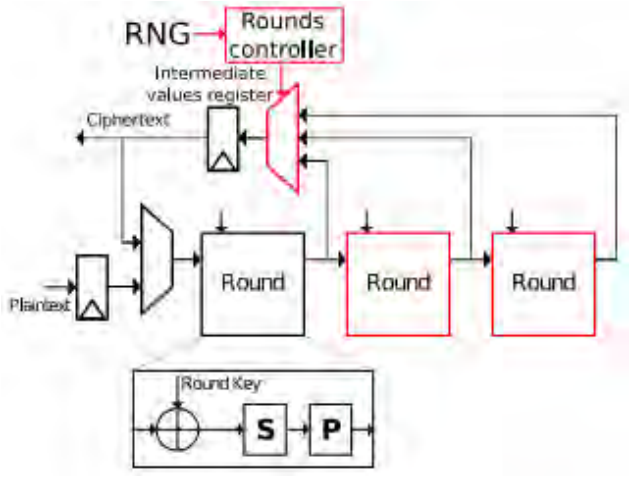


Figure 26: Architecture avec ajout de fausses rondes

Un contrôleur a pour rôle d'assurer que le calcul de chiffrement soit correctement effectué au bout de 16 cycles d'horloge. Il existe alors 5 196 627 façons possibles de calculer les 32 rondes de l'algorithme PRESENT. L'architecture a été mise en œuvre sur une cible FPGA Xilinx Spartan-6 à l'aide du kit SAKURA-G, et évaluée avec la métrique TVLA - *Test Vector Leakage Assessment* [69] sur 100 000 traces. La TVLA est une évaluation qui permet de détecter n'importe quel type de fuite (Dépendance entre les mesures et le calcul). Une valeur *t-statistique* est calculée, si la valeur absolue n'excède pas un seuil prédéfini de 4,5 le test est réussi et l'implémentation est considérée comme sécurisée contre les attaques par canaux cachés.

Dans [67], la plus grande fuite se situe au début du chiffrement étant donné que la probabilité que le registre contienne la sortie de la première ronde est de 1/3. Ce travail a été étendu en présentant une version alternative dans [70]. Cette fois le nombre de rondes valides est configurable. Une option a été ajoutée pour permettre un calcul « fictif » de la première ronde durant lequel la valeur intermédiaire est réécrite dans le registre (cf. *Figure 27*). Toutefois, les résultats de l'évaluation TVLA sur ce modèle ont montré que l'utilisation de cycles vides dans lequel aucune ronde n'est calculée détériore les résultats en termes de sécurité. Pour remédier à cela une valeur aléatoire est systématiquement écrite dans le registre de sortie durant les cycles vides où aucune ronde n'est calculée, dans ce scénario un million de traces ont été collectées est la valeur maximale *t-statistique* de 19,16 qui est approximativement quatre fois supérieure au seuil maximal de 4,5. L'apport en sécurité ne peut toutefois pas être

estimé étant donné que les résultats ne sont pas comparés avec une version non protégée.

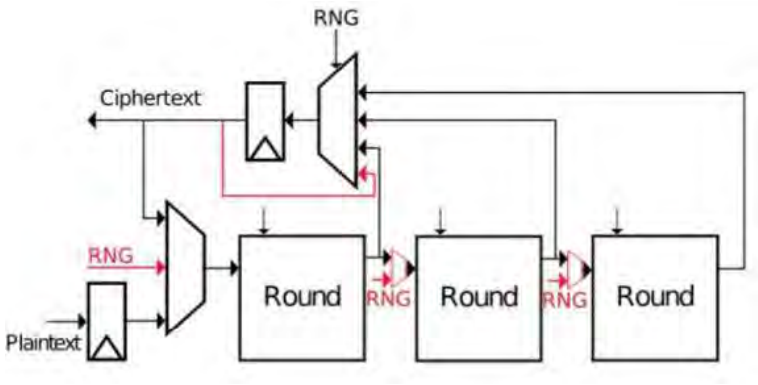


Figure 27: Architecture avec fausses rondes modifiées [70]

b. Ajout d'aléa dans l'ordre de calcul

Dans la littérature on trouve peu de travaux portant sur l'ajout d'aléa dans l'ordre des opérations au sein d'architectures matérielles.

En 2016, Sasdrich et al. [27] ont proposé une architecture compacte entièrement séquentielle d'un algorithme AES dans laquelle une résistance aux attaques par canaux cachés est proposée via l'ajout d'aléa dans l'ordre des calculs. L'architecture a été validée sur une carte Xilinx Spartan-6, pour un coût de 24 slices, une latence de 1471 cycles d'horloge et un débit de 7.82Mbit/s. Les permutations sont générées en suivant l'approche proposée par Veyrat-Charvillon et al. [55]. À chaque chiffrement, une nouvelle permutation est dérivée à partir de la précédente. Alors les signaux d'adresses sont replacés selon la permutation courante.

L'architecture a été mise en œuvre sur une carte XILINX Spartan-6 XC6SLX75 intégrée dans un kit SAKURA-G. Dans leur évaluation de sécurité une attaque CPA a été réalisée en utilisant des mesures de consommation de puissance réalisées avec un oscilloscope numérique en condition de laboratoire (c.-à-d. sans bruit additionnel) et en condition réelle (c.-à-d. avec un second cœur exécutant des opérations arithmétiques en parallèle qui occupe 80% de la surface du FPGA). Dans des conditions de laboratoire, le MTD de la version non protégée est de 200 traces, celui de la version protégée est 500 000 traces (soit une amélioration du coefficient de sécurité de 250). Dans un environnement plus réaliste où le système cryptographique est embarqué, l'architecture sans protection le MTD est de 5000 traces, alors qu'avec

la protection il est de 800 000 traces pour obtenir le même résultat améliorant la sécurité d'un facteur 160.

Dans le travail proposé par [74] en 2016, une contre-mesure d'ajout d'aléa sur deux rondes consécutives a été implémentée sur une carte FPGA XILINX Virtex-5 XC5VLX50. Un algorithme est chargé de générer une tout en respectant les contraintes des dépendances. L'architecture a été implémentée avec 503 slices pour une fréquence maximale de 70 Mhz. Cependant l'architecture n'a pas été détaillée et il n'y a pas d'information sur la latence.

La solution proposée est évaluée avec la métrique TVLA en utilisant des traces simulées en poids de Hamming (avec ajout de bruit avec différentes variances), ainsi qu'avec des traces collectées avec un oscilloscope Tektronix DPO4034B en utilisant une carte SASEBO-GII. Pour cela, 10 000 traces avec des textes clairs fixes et aléatoires ont été collectées. Les résultats ont montré que l'ajout d'aléa sur deux rondes consécutives augmente la protection. En effet, avec 10 000 traces, le seuil de 4,5 n'est pas atteint en ajoutant de l'aléa sur deux rondes tandis que sur une seule ronde le seuil est atteint avec 7000 traces approximativement. Les résultats en simulation sont similaires.

Dans un travail récent proposé dans [75], l'aléa a été ajouté dans une architecture AES sur un chemin de 8 bits non protégé, présenté dans [28], afin d'en augmenter la résistance face aux attaques par canaux cachés. Pour ce faire, les auteurs proposent de modifier le module *Enable Generator* qui a pour rôle de générer des bits de contrôle pour l'adressage des octets de sorte que l'ordre de calcul sur les mots (un mot correspondant à une colonne de la matrice d'état AES) et les octets de ces mots soient aléatoires. Ainsi, un octet peut être calculé dans n'importe quel cycle parmi les 16 cycles de la ronde. Par conséquent, le module réalisant le MixColumns peut recevoir une colonne dans quatre ordres différents (s0, s1, s2, s3), (s1, s2, s3, s0), (s2, s3, s0, s1), (s3, s0, s1, s2).

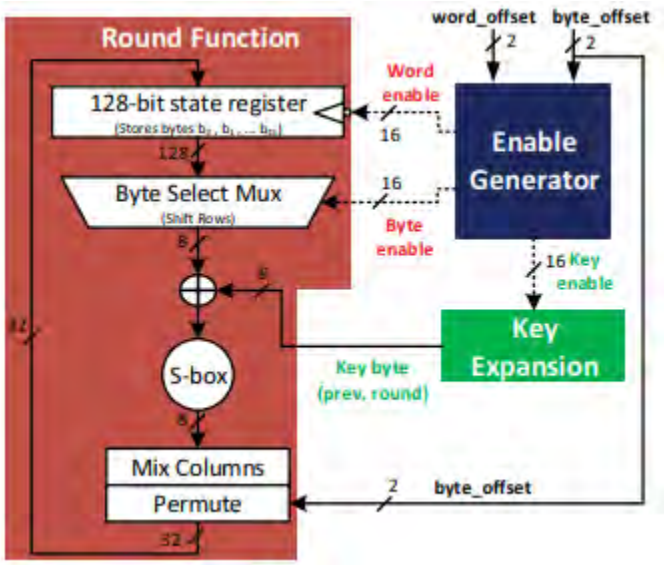


Figure 28: Architecture AES proposée dans [75]

L'architecture présente un étage de permutation nommé « *Permute* » qui se compose de quatre multiplexeurs 4:1 chargés de réordonner les octets avant qu'ils soient stockés dans le registre (cf. *Figure 29*).

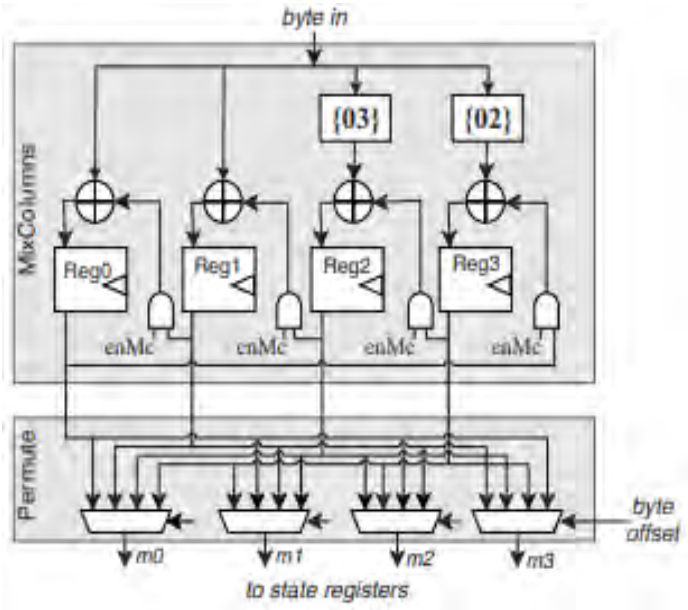


Figure 29: Module de MixColumns et Permute dans [28]

L'architecture a été conçue pour une cible ASIC avec une technologie 16nm. Elle occupe une surface de 1201 µm², propose un débit de 184 Mb/s et engendre des surcoûts de 36% en surface et 25% en consommation d'énergie en comparaison avec leur architecture de référence non protégée. Enfin, la latence du circuit protégé est de 204 cycles d'horloge ce qui est 1,275 fois supérieur à la latence de référence (160 cycles d'horloges). L'impact sur la sécurité a été évalué en simulant des traces avec

l'outil *PrimTime* et l'architecture a été attaquée avec différentes clés. La moyenne du MTD pour révéler la bonne clé dans l'architecture protégée est de 15 983 contre 1492 traces pour l'architecture non protégée.

Une étude expérimentale a été effectuée dans [76]. Dans ce travail, l'architecture de référence [24], la version non protégée optimisée en énergie proposée par les mêmes auteurs [28] et la version avec protégée [75] ont été implémentées en ciblant une technologie ASIC 16-nm FinFET. Afin de tester leurs circuits, les puces ont été intégrées sur un PCB au côté d'un FPGA Artix 7 gérant les communications entre MATLAB et le circuit. MATLAB a pour rôle d'envoyer les textes clairs et valider les textes chiffrés. La conversion du niveau de tension entre le FPGA et le circuit se fait à l'aide de convertisseurs dédiés. Des traces de consommation de puissance sont alors collectées, en utilisant un oscilloscope Keysight *MSOX4154A* et sont traitées avec MATLAB.

Le MTD est de 118 400 traces avec protection tandis que, sans protection, seules 761 traces sont nécessaires dans la version optimisée en énergie [28] donnant un coefficient d'amélioration de la sécurité de 155. L'architecture non protégée de référence [24], quant à elle offre un MTD de 1879 traces. Les résultats montrent que dans cette version il existe beaucoup plus de points de fuite du fait du décalage des données dans le registre, mais que le coefficient de corrélation est beaucoup plus faible que celui de la version avec renommage des registres où l'écriture se fait en un seul coup d'horloge. Ceci est certainement dû à l'augmentation de la proportion de bruit dans les traces mesurées.

c. Ajout d'aléa au niveau bits

Dans Levi *et al.* [77], les auteurs ont proposé une alternative pour ajouter de l'aléa dans les architectures parallèles au niveau bit. En considérant 3 bits, 3 signaux d'horloge sont générés à partir de l'horloge principale avec un déphasage (lui-même obtenu par la *PLL - Phase Locked Loop* du FPGA). Dans le cas classique, une seule horloge est assignée à tous les registres. Dans le but d'ajouter de l'aléa sur le temps d'échantillonnage des bits et de distribuer la fuite au sein d'un seul cycle d'horloge, chacun des trois bits d'entrée de la Sbox est assigné à une FF « Flip-Flop » et chacune de ces trois FF est contrôlée par un signal d'horloge indépendant (cf. Figure 30). L'écriture des bits dans les registres se fait à différents instants au sein d'un même cycle d'horloge. Les *3!* permutations possibles sont stockées dans une mémoire

dédiée, et trois multiplexeurs sont en charge de sélectionner une permutation pour les trois signaux d'horloges déphasés, et ce à chaque coup d'horloge.

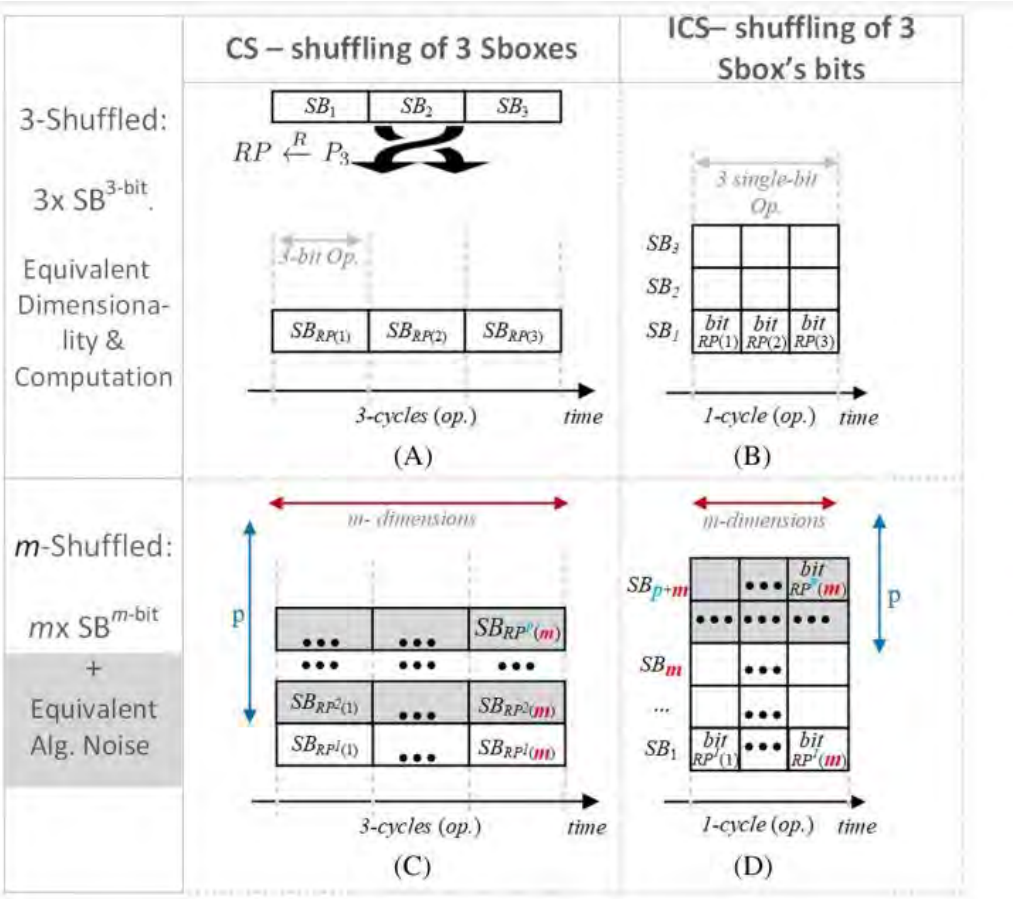


Figure 30: Architecture proposée dans [77]

L'ajout d'aléa dans l'ordre des opérations (*CS*) et l'ajout d'aléa dans les bits (*ICS*) sont évalués. La Figure 30(A) illustre un exemple l'ajout d'aléa dans l'ordre des opérations « CS » sur 3 octets de la Sbox ; La Figure 30(B) illustre l'ajout d'aléa dans les bits en un cycle d'horloge « ICS » sur 3 bits d'un octet de la Sbox. La Figure 30(C) illustre l'ajout d'aléa dans l'ordre de calcule des trois octets de la Sbox en considérant le bruit algorithmique qui correspond aux calculs parallèles. La Figure 30(D) illustre l'exemple d'ajout d'aléa dans les bits en un cycle d'horloge couplé avec le bruit algorithmique.

Dans ce contexte deux stratégies de génération d'aléa sont mises en œuvre sur un FPGA Xilinx Spartan-6 embarqué dans un kit SAKURA-G. La première, basée sur une mémoire, engendre un surcoût en termes de slices de 80% pour une architecture avec un chemin de données de 32 bits et 44% pour une architecture possédant un chemin de données de 128 bits. La seconde est réalisée à la volée en utilisant l'algorithme Fisher-Yates. Le surcoût en termes de nombre de slices consommés est de 73% pour une architecture avec un chemin de données de 32 bits et 33% pour une architecture

possédant un chemin de données de 128 bits en comparaison avec les versions non protégées de ces architectures.

Dans le but d'évaluer l'approche en termes de sécurité, une analyse théorique de l'information mutuelle MI est effectuée sur une implémentation séquentielle de l'AES différent niveau de parallélisme de l'AES ($p = 1$, $p = 4$ et $p = 8$ pour des chemins de données de 8, 32 et 128 bits respectivement). Il est conclu que l'ajout d'aléa combiné avec le bruit algorithmique (un chemin de données large) peut avoir un apport important en sécurité. En utilisant l'information mutuelle MI, les auteurs ont montré qu'un attaquant peut apprendre un maximum de 9×10^{-2} bits pour une architecture protégée avec un chemin de données de 32 bits et 2×10^{-2} bits pour une architecture protégée avec un chemin de données de 128 bits, tandis qu'une architecture non protégée peut révéler 20 bits d'information, et un maximum de 40 bits sans considérer le bruit.

d. Ajout d'aléa dans le stockage

Dans l'approche proposée par [70], un second registre est ajouté à l'architecture afin que la valeur intermédiaire et la valeur aléatoire puissent être stockées en alternance. Ainsi à chaque cycle d'horloge, les contenus des deux registres sont échangés. L'architecture décrite dans la Figure 31 et a été évalué en TVLA : avec 100 000 traces collectées à l'aide d'un oscilloscope PicoScope *6404D*, la valeur t-statistique maximale est *14,27* indiquant la présence d'une fuite d'information importante malgré la protection.

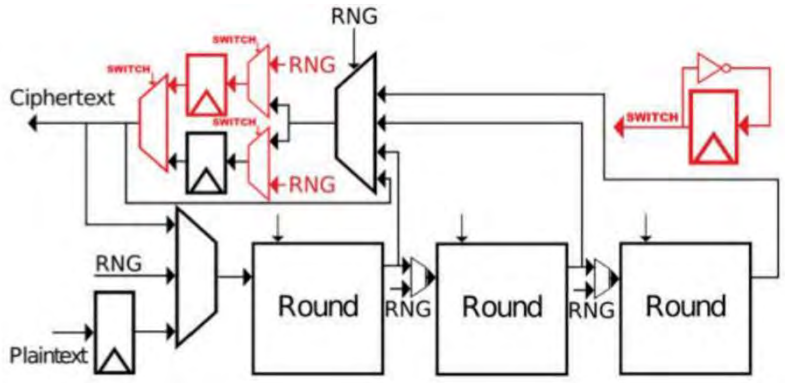


Figure 31 : Architecture avec aléa dans le stockage proposée dans [70]

III. Conclusion

Dans ce chapitre, après une introduction de différentes mises en œuvre de l'AES en matériel, différentes contre-mesures par ajout d'aléa ont été présentées. Pour les

processeurs, beaucoup de travaux se sont penchés sur l'automatisation de l'ajout d'aléa de différentes natures : ordre des opérations, opérations fictives, sémantique ou stockage. Beaucoup de travaux implémentent l'aléa de façon dynamique en utilisant différentes techniques combinées dans le but d'avoir un nombre exponentiel de chemins d'exécution possibles. Cependant, lorsque l'on s'intéresse aux accélérateurs matériels non programmables, ces approches ont été très peu étudiées. Dans [74] et [27] les permutations des données sont générées de façon algorithmique ce qui est très coûteux en temps d'exécution. Dans le travail le plus récent, évalué en simulation dans [75] et en mesure dans [76], les auteurs n'ont pas donné d'informations sur la façon dont les permutations des données sont générées. Les résultats au niveau performance et surface ainsi que les méthodes d'évaluation et les technologies ciblées pour ces travaux sont comparées dans le Tableau 2.

Tableau 2: Comparaison des travaux portant sur l'ajout d'aléa matériel

Réf	Contre-mesure	Technologie	Surface	Latence	Débit (Mbit/s)	Evaluation	Méthode de collecte de traces	Nombre de traces	Coefficient de sécurité
[27]	Ajout d'aléa sur l'ordre des opération	FPGA Xilinx Spartan-6 XC6SLX4	24 Slices	1471	7.82	CPA (clé récupéré)	Oscilloscope numérique LeOmeg-16y	5000 - 500 000	160-250
[74]	Ajout d'aléa sur l'ordre des opérations sur deux rondes	FPGA Xilinx Virtex-5 (XC5VLX50)	503 Slices	N/A	N/A	TVLA (seuil non atteint)	Oscilloscope Tektronix DPO4034B	10 000	-
[75]	Ajout d'aléa sur l'ordre des colonne et mot sur deux rondes	ASIC (16-nm FinFET)	1201 μ^2m	204	184	CPA (clé récupéré)	PrimTime (simulation)	15 000	10
[76]	Ajout d'aléa sur l'ordre des colonne et mot sur deux rondes	ASIC (16-nm FinFET)	1201 μ^2m	204	184	CPA (clé récupéré)	Oscilloscope Keysight MSOX4154A	118 400	155

L'architecture la plus compacte a été proposée dans [27] où l'aléa a été ajouté sur l'ordre de calcul des 16 octets de l'état intermédiaire. Cependant, le surcoût en performance est important et la génération est faite de façon algorithmique. Dans [74] l'aléa a été ajouté sur deux rondes consécutives. De plus, le peu de détails fourni sur l'architecture proposée ne permet pas d'en décrire le fonctionnement réel. Dans [75], l'aléa a été ajouté uniquement dans l'ordre des colonnes et des octets appartenant à une même colonne. Cette contre-mesure a été détaillée et évaluée en simulation et en mesure dans [76]. Cependant, dû à la limitation dans l'ajout d'aléa, le coefficient de sécurité se limite à 155. Bien que l'ajout d'aléa ait été formalisé en logiciel [55], les

études présentées ne permettent pas une évaluation du potentiel de l'ajout d'aléa en matériel. La sérialisation des opérations dans [27] engendre une multiplication des points de fuite et une réduction du bruit algorithmique, ce qui a pour effet de fragiliser l'implémentation contre les attaques différentielles d'analyse de consommation de puissance. Les auteurs de [74] présentent très peu de détail sur l'architecture. Enfin, dans [75] et [76], les permutations sont limitées et la méthode de génération de ces permutations n'est pas décrite.

Le chapitre suivant présente le modèle d'architecture que nous proposons pour intégrer l'aléa sur les calculs et dans le stockage.

Chapitre 3 Modèle d’architecture proposé

I. Description de notre contribution

L'objectif de notre contre-mesure est d'introduire de l'aléa dans l'ordre des opérations de calcul et de stockage dans une architecture matérielle d'un algorithme AES-128. Bien qu'il existe beaucoup de travaux sur l'ajout d'aléa en logiciel, très peu visent le matériel. En effet, en logiciel, mélanger les instructions indépendantes engendre peu de surcoûts au niveau des ressources et des performances. Il faut simplement s'astreindre à respecter les dépendances de calcul entre les octets d'une même colonne de la matrice d'état AES lors de l'opération du MixColumns.

Une architecture matérielle est, quant à elle, conçue et optimisée selon l'algorithme et les dépendances de données pour offrir du parallélisme. Dans ce contexte, l'ajout d'aléa peut, en modifiant le flot d'exécution de l'algorithme, réduire le niveau de parallélisme et impacter les performances.

Dans l'implémentation matérielle la plus efficace de l'état de l'art d'un AES avec un chemin de données de 8 bits, le calcul de chaque colonne de la matrice d'états se fait en quatre cycles pour l'opération du MixColumns. Les octets sont adressés et accumulés séquentiellement cycle par cycle. Il existe alors deux solutions pour l'ajout d'aléa :

- *Ajout d'aléa limité* à l'ordre des colonnes et des octets appartenant à une même colonne permettant de garder le calcul de l'opération MixColumns en quatre cycles par colonne et de minimiser le surcoût en performance.
- *Ajout d'aléa complet* sur l'ordre des calculs des 16 octets de la matrice d'état engendrant un calcul en deux temps. Dans un premier temps, les opérations AddRoundKey, SubBytes et ShiftRows sont réalisées. Puis l'opération MixColumns est à son tour réalisée lorsque les 16 octets intermédiaires sont disponibles. La latence est alors multipliée par 2 lorsqu'un 1 cycle d'horloge par octet est nécessaire pour chacune de ces deux phases.

Dans ces travaux, ces deux approches ont été mises en œuvre et couplées avec une méthode d'ajout d'aléa sur le stockage. Le schéma de principe de nos architectures est illustré dans la Figure 32. Cette architecture se compose d'un module de brassage, un générateur de nombre aléatoire (RNG), un contrôleur et un chemin de données. Le module de brassage de données (*module shuffling*) est chargé de générer les permutations. Ce module est alimenté par une valeur pseudo-aléatoire (PRNG) générée à partir d'une graine, et un compteur cpt délivré par le contrôleur à chaque

cycle d'horloge, il est réinitialisé à chaque ronde. La permutation de l'ordre de traitement des 16 octets P correspondante à l'itération courante du compteur est ainsi délivrée en sortie. Le chemin de données de 8 bits est conçu et optimisé pour un aléa dans l'ordre de calcul des octets. Il reçoit en entrée, la clé initiale $K0$ et le texte clair D_{in} ainsi que les adresses de mémoires délivrées par le contrôleur et génère en sortie le texte chiffré D_{out}. Le contrôleur fournit la valeur du compteur cpt au module de brassage, génère la table de permutation inverse P^{-1} et de fournit les signaux de contrôle et les adresses correspondantes à l'opération courante en tenant compte de la permutation. Enfin, l'implémentation globale inclut l'interface et l'UART en charge de la communication PC-FPGA.

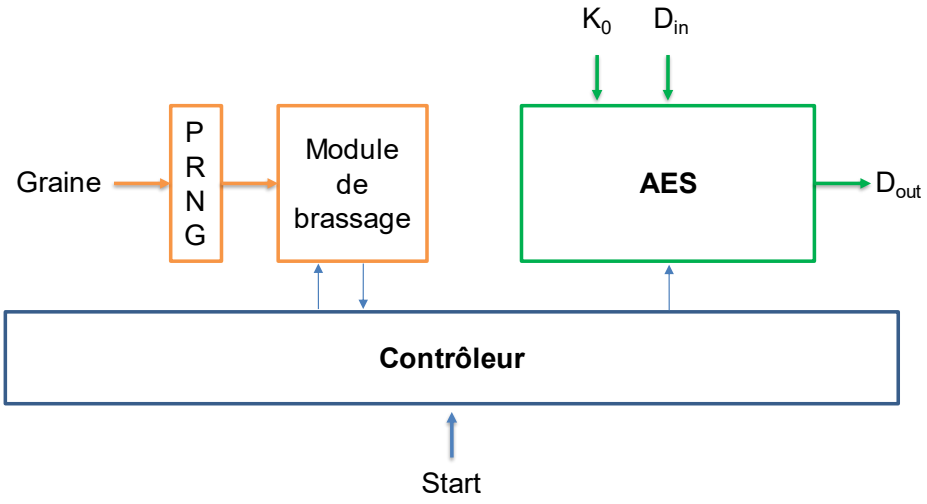


Figure 32 : Schéma de principe de notre modèle d'architecture

Dans les sections suivantes, nous détaillons ces différents modules et nous montrons comment les exploiter pour explorer l'espace de conception avec différents réseaux de brassage de données, différents types d'informations brassées et différentes technologies mémoires. Dans un premier temps nous présentons le générateur pseudo-aléatoire implémenté dans nos architectures.

1. Générateur pseudo-aléatoire (PRNG)

L'algorithme Trivium [79] est utilisé comme générateur de nombres aléatoires. Il s'agit d'un algorithme de chiffrement synchrone en flux, orienté vers les implémentations matérielles et capable de générer jusqu'à 2^{64} bits de clé (*keystream*).

Le processus est constitué en deux phases : la première est une phase d'initialisation dans laquelle les 288 bits d'état interne sont affectés avec un vecteur d'initialisation IV

de 80 bits et d'une clé K de 80 bits ; dans la seconde phase les bits du keystream sont générés en mettant à jour les bits d'état interne de façon répétitive.

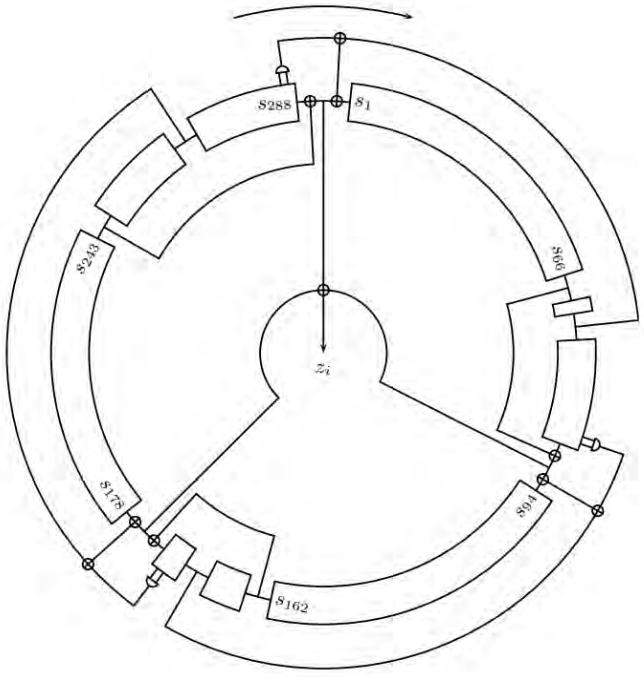


Figure 33: Architecture Trivium

Dans la phase d'initialisation, un registre noté *S (S₁ … S₂₈₈)* est affecté par une première clé et le vecteur IV. Les bits restants sont initialisés par des '0' à l'exception des trois derniers bits *S₂₈₆, S₂₈₇, S₂₈₈* :

$$(s_1, s_2 \dots s_{93}) \longleftarrow (K_1 \dots K_{80}, 0, \dots 0)$$
$$(s_{94}, s_{95} \dots s_{177}) \longleftarrow (IV_1 \dots IV_{80}, 0, \dots 0)$$
$$(s_{178}, s_{279} \dots s_{288}) \longleftarrow (0, \dots, 0, 1, 1, 1)$$

S'en suivent quatre rotations du cycle entier (288 itérations) sur l'état avant le début des calculs :

$$\boldsymbol{for}\ i = 14 \times 288\ \boldsymbol{do}$$
$$t_1 \longleftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$$
$$t_2 \longleftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$$
$$t_3 \longleftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$$
$$(s_1, s_2 \dots, s_{93}) \longleftarrow (t_3, s_1, \dots, s_{82})$$
$$(s_{94}, s_{95} \dots, s_{177}) \longleftarrow (t_1, s_{94}, \dots, s_{176})$$
$$(s_{178}, s_{279} \dots, s_{288}) \longleftarrow (t_2, s_{178}, \dots, s_{287})$$
$$\boldsymbol{end\ for}$$

La génération des bits de keystream (cf. *Figure 33*) peut commencer avec un calcul itératif où 15 bits internes spécifiques parmi les 288 bits de l'état interne sont utilisés

pour mettre à jour 3 bits ainsi que pour générer 1 bit du keystream. Puis une rotation sur les bits d'état est effectuée. Le processus est itéré jusqu'à la génération des $N<2^{64}$ bits requis du keystream.

$$
\begin{aligned}
&\boldsymbol{for}\ i = 1N\ \boldsymbol{do}\\
&\quad t_1 \longleftarrow s_{66} + s_{93}\\
&\quad t_2 \longleftarrow s_{162} + s_{177}\\
&\quad t_3 \longleftarrow s_{243} + s_{288}\\
&\quad z_i \longleftarrow t_1 + t_2 + t_3\\
&\quad t_1 \longleftarrow t_1 + s_{91} \times s_{92} + s_{171}\\
&\quad t_2 \longleftarrow t_2 + s_{175} \times s_{176} + s_{264}\\
&\quad t_3 \longleftarrow t_3 + s_{286} \times s_{287} + s_{69}\\
&\quad (s_1, s_2 \ldots, s_{93}) \longleftarrow (t_3, s_1, \ldots, s_{82})\\
&\quad (s_{94}, s_{95} \ldots, s_{177}) \longleftarrow (t_1, s_{94}, \ldots, s_{176})\\
&\quad (s_{178}, s_{279} \ldots, s_{288}) \longleftarrow (t_2, s_{178}, \ldots, s_{287})\\
&\boldsymbol{end\ for}
\end{aligned}
$$

L'algorithme est utilisé comme un générateur de nombres aléatoires, la graine servant de clé et de vecteur d'initialisation (vecteur IV). On a alors une sortie d'un bit aléatoire à chaque cycle d'horloge. Ce dernier est concaténé aux précédents dans un registre dont la taille dépend du vecteur de nombre aléatoire requis par le module de brassage de données.

Dans le but d'assurer la diversité de la génération des nombres aléatoires, l'algorithme doit être périodiquement réinitialisé avec un nouveau vecteur IV avant la fin de la période de la séquence du trivium. Celle-ci est difficile à déterminer, en raison de la non-linéarité du trivium. Cependant en utilisant n'importe quelle clé ou vecteur d'initialisation IV, il est possible de générer au moins une séquence de $2^{96-3}-1$ bits. Le vecteur d'initialisation peut être généré par un TRNG. Le nombre aléatoire généré par ce module est alors utilisé pour piloter le module de brassage dont le principe général de fonctionnement est présenté dans la section suivante.

2. Génération des permutations (module de brassage)

L'ordre de traitement des octets est défini par une permutation de l'ordre itératif initial. Les permutations sont générées à base d'un réseau de permutation constitué de commutateurs. Chaque commutateur se compose de deux entrées, deux sorties et un bit de contrôle. Quand le bit de contrôle est à '0', les entrées sont directement

propagées vers la sortie sans permutation (Figure 34.a). Quand il est à '1', les entrées sont permutées comme illustré dans la Figure *34*.

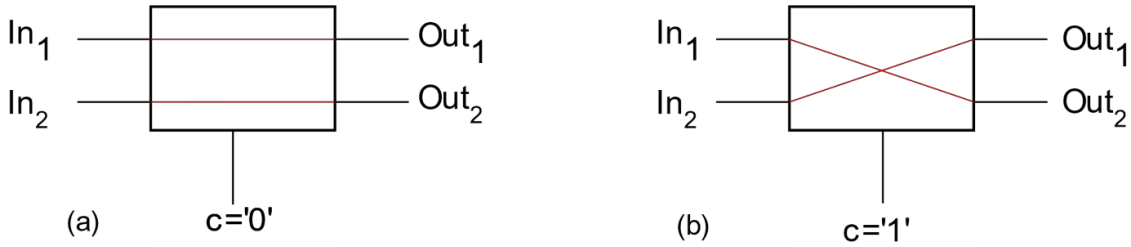


Figure 34 : Commutateur (a) entrée non permutée, (b) entrée permutée

Les commutateurs sont arrangés en colonnes appelées *étages,* et entre chaque étage, des connexions sont mises en place pour générer différentes topologies de réseaux de permutations. Il existe trois types de réseaux : les réseaux bloquants qui ne sont pas capables de réaliser toutes les permutations ; les réseaux non bloquants qui sont capables d'établir un lien de n'importe quelle entrée vers n'importe quelle sortie, sans prendre en compte les interconnexions déjà établies ; les réseaux non bloquants ré-arrangeables, qui sont capables d'établir un lien de n'importe quelle entrée vers n'importe quelle sotie, en prenant en compte les interconnexions existantes.

Les réseaux *Banyan* [80] sont des réseaux bloquants qui ont pour particularité d'offrir un chemin unique de n'importe quelle entrée vers n'importe quelle sortie. On peut citer dans cette famille les réseaux *Omega* [81], *Delta* [82] ou *Indirect Binary n-cube* [83] dont les topologies sont équivalentes. Ils sont constitués de $log_2 N$ étages offrant ainsi un maximum de $2^{\frac{N \times log_2 N}{2}}$ permutations.

Les réseaux *Clos* [84] quant à eux sont des réseaux non bloquants ré-arrangeables conçus, à l'origine, pour satisfaire les besoins des opérateurs téléphoniques. On distingue notamment le réseau de Benes [85] (un cas particulier des réseaux Clos). Ce réseau est capable d'offrir $N!$ permutations dans lesquelles chaque entrée et capable d'accéder à n'importe quelle sortie sans aucun conflit. Le nombre de commutateurs nécessaires est alors $\frac{N}{2} \times (2 \times log_2 N - 1)$.

Dans un contexte de sécurité, afin de créer un maximum d'aléa, le réseau de Benes semble une bonne option (Figure *35*.a). Ce dernier permet de générer toutes les permutations possibles (soit $N!$ permutations). Toutefois, son coût en surface est important au regard du nombre de multiplexeurs qu'il requiert. En effet, chaque commutateur est lui-même composé de deux multiplexeurs (cf. Figure 36) de n bits.

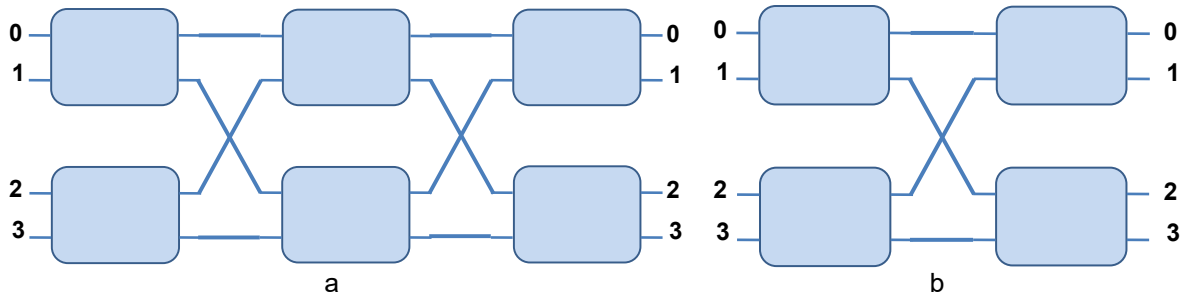


Figure 35 : Réseau 4x4 (a) Benes à 6 commutateurs (b) Omega à 4 commutateurs

Dans une architecture AES avec un chemin de données de 8 bits, les 16 octets de la matrice d'état sont traités octet par octet. Afin d'ajouter un maximum d'aléa sur l'ordre de traitement de ces octets, un réseau de Benes 16x16, peut être utilisé et obtenir ainsi 16! permutations. Un tel réseau de Benes se compose de sept étages dont chacun nécessite 8 commutateurs. Au total, le réseau se compose de 56 commutateurs avec des entrées sorties de 4 bits (112 multiplexeurs 2 vers 1 de 4 bits). Ce réseau peut être substitué avec un réseau banyan (ex. Omega, Figure 35.b), dont le coût est plus raisonnable tout en permettant un nombre important de permutations. Un réseau d'une taille 16x16 de ce type se compose de quatre étages dont chacun est composé de 8 commutateurs, avec au total 32 commutateurs avec des entrées/sorties de 4 bits (64 multiplexeurs 2 vers 1 de 4 bits). Ce qui nous permet d'obtenir un nombre de permutations égal à 2^{32}.

Dans le cas d'une architecture, où l'ajout d'aléa se limite à l'ordre de traitement des quatre colonnes et aux octets d'une même colonne dans un algorithme AES (quatre colonnes de quatre octets), les permutations peuvent être générées par deux réseaux de Benes 4x4. La permutation P des 16 octets peut se composer à partir de cinq permutations. Une première permutation permet de définir l'ordre de traitement des quatre colonnes. Les 4 permutations qui suivent permettent de définir l'ordre de traitement des quatre octets de chaque colonne, de sorte que chaque colonne puisse être calculée en quatre cycles indépendamment de l'aléa généré. Ainsi on peut obtenir un aléa sans dégrader les performances. De plus, le surcoût en surface est fortement minimisé. En effet un réseau de Benes 4x4, se compose de 3 étages dont chacun se compose de deux commutateurs (Figure 35.a). Au total 6 commutateurs sont utilisés (12 multiplexeurs 2 vers 1 de 2 bits).

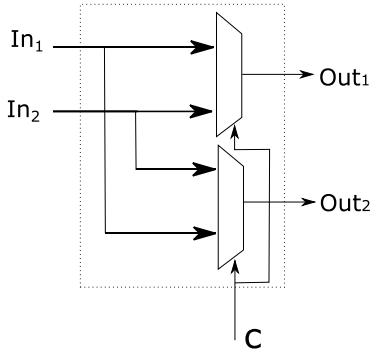


Figure 36 : Architecture d'un commutateur

Ce principe est mis en œuvre dans le module de brassage qui sera détaillé dans chaque architecture. La permutation P est représentée sous la forme d'un vecteur déterminant l'ordre de traitement des données. Lors des traitements, il est possible de ne pas inverser cette permutation durant le stockage d'un résultat dans un élément mémoire. On obtient alors non seulement un ordre aléatoire pour les traitements, mais également pour le stockage. La section suivante présente la gestion de cet aléa dans le cadre de ces travaux de thèse.

3. Gestion de l'aléa dans le stockage

Dans nos travaux, l'aléa est ajouté non seulement dans l'ordre des calculs, mais également pour le stockage des données. À chaque chiffrement, une nouvelle permutation P des 16 octets de la matrice d'état est générée par le module Shuffling. Afin d'assurer la justesse des calculs, les positions en mémoire des valeurs intermédiaires pour chaque octet sont stockées. Elles sont ensuite récupérées à l'aide d'une table de permutation inverse P^{-1} construite à partir de la table de permutation P.

Dans nos mises en œuvre, le compteur cpt s'incrémente à chaque cycle d'horloge et se réinitialise au début du calcul de chaque ronde. Ce compteur est utilisé comme indice dans la table de permutation et permet de construire la table de permutation inverse. Au début du chiffrement, la table de permutation inverse P^{-1} est construite pendant les 16 premières itérations du compteur, en inversant les indices et les valeurs de la permutation P. L'itération courante du compteur cpt est stockée à l'indice correspondant à la valeur de la permutation à l'itération courante : $P^{-1}\left[P\left[cpt[3:0]\right]\right] = cpt[3:0]$.

Le Tableau 3 illustre un exemple de permutation permettant de traiter les octets de la matrice d'état AES dans un ordre aléatoire. Dans ce tableau, l'indice correspond à la

valeur du compteur cpt et la valeur est la permutation $P[cpt[3:0]]$ correspondante définissant l'octet à traiter. Une fois traité, cet octet sera stocké en mémoire à l'adresse correspondante à la valeur de cpt.

Tableau 3 : Exemple d'une table de permutation P

Indice (cpt)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$P[cpt]$	**10**	**12**	**6**	**13**	**1**	**15**	**0**	**5**	**9**	**8**	**3**	**11**	**2**	**4**	**7**	**14**

Le Tableau 4 illustre la table de permutation inverse pour la permutation présentée dans la Tableau 3. Ici, la première ligne du tableau représente l'indice d'un octet au sein de la matrice d'état AES. La seconde ligne représente l'adresse à laquelle ce dernier est stocké en mémoire.

Tableau 4 : Exemple de la table de permutation inverse P^{-1}

Indice (cpt)	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
Valeur	**6**	**4**	**12**	**10**	**13**	**7**	**2**	**14**	**9**	**8**	**0**	**11**	**1**	**3**	**15**	**5**

Dans toutes les architectures qui ont été conçues durant ces travaux de thèse, la table de permutation inverse est stockée au sein de mémoires distribuées. Elle est utilisée à l'exécution pour assurer le bon fonctionnement des architectures proposées.

4. Architecture du module d'expansion de la clé

Dans l'algorithme AES, une clé de ronde est utilisée dans l'opération *AddRoundKey* à chaque ronde. Un module d'expansion est chargé de calculer ces clés de rondes à partir de la clé initiale. Dans notre architecture, les clés de ronde sont calculées à la volée comme c'est le cas dans l'architecture de référence [26] sur laquelle nous nous basons. En raison de l'ajout d'aléa dans l'ordre de calcul pendant le chiffrement, la clé de ronde courante doit être calculée et stockée à l'avance, durant la ronde précédente. L'architecture de l'algorithme d'expansion de la clé a alors été optimisée dans ce sens.

On note $K0$ la clé initiale et KX une clé de ronde avec $1 \leq X \leq 10$. De plus, on note $K0_j$, le *j*^{ième} octet de la clé initiale et KX_j le *j*^{ième} octet d'une clé de ronde. Chaque clé est représentée par une matrice d'octets de taille 4x4. Le calcul des clés de ronde est effectué selon l'algorithme suivant :

On note W une matrice $4x4$ représentant les colonnes des 11 clés de ronde (4 colonnes de 4 octets par clé) incluant la clé initiale $K0$ et W_i la ieme colonne de la matrice contenant quatre octets d'une clé $0 \leq i \leq 4 \times R$ et et $1 \leq R \leq 11$. Les quatre premières colonnes définissent la clé initiale :

$W_0 = [K0_0 K0_1 K0_2 k0_3]$,
$W_1 = [K0_4 K0_5 K0_6 K0_7]$,
$W_2 = [K0_8 K0_9 K0_{10} K0_{11}]$,
$W_3 = [K0_{12} K0_{13} K0_{14} K0_{15}]$.

La fonction ROT est une rotation des quatre octets d'une colonne

W_i: $ROT([k_0 k_1 k_2 k_3]) = [k_1 k_2 k_3 k_0]$.

La fonction $SubWord$ est une application de l'opération de SubBytes de l'algorithme AES sur chaque octet d'une colonne,

$$SubWord([k_0 k_1 k_2 k_3]) = [sBox(k_0) sBox(k_1) sBox(k_2) sBox(k_3)].$$

Dans le but de déterminer les éléments de W, il est nécessaire de mettre en œuvre les calculs suivants :

Si $i < 4$: $W_i = K0_i$ (*Équation 2)*

Sinon si $i \bmod 4 = 0$ et $i \geq 4$:

$$W_i = SubWord\big(ROT(W_{i-1})\big) \oplus rcon \oplus W_{i-4} \quad (\textit{Équation 3})$$

Sinon $W_i = W_{i-1} \oplus W_{i-4}$ *(Équation 4)*

Où le vecteur $rcon$ est défini par $rcon = [rci, 00,00,00]$ avec rci la constante de la ronde courante (cf. Tableau 5).

Tableau 5 : Constante de la ronde

X	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
rci	01	02	04	08	10	20	40	80	1B	36

On remarque que chaque colonne de la matrice d'états est dépendante d'une colonne de la ronde courante ainsi que d'une colonne de la ronde précédente. Dès lors, la possibilité d'ajout d'aléa se retrouve restreinte dans l'ordre de calcul des quatre octets d'une même colonne. Par conséquent, l'ajout d'aléa dans ce module n'a pas un intérêt suffisant au regard du surcoût qu'il pourrait engendrer. Dans nos travaux, le module d'expansion de la clé n'est donc pas protégé.

L'architecture de l'expansion de la clé est illustrée dans la Figure 37. La SRL_0 est utilisé dans le calcul de la clé de ronde.

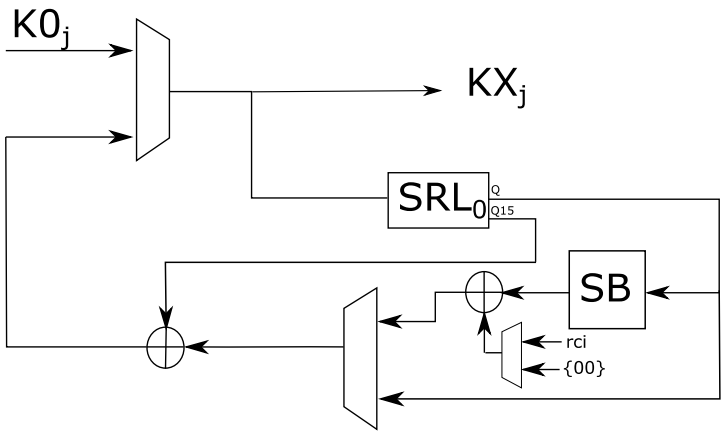


Figure 37: Architecture de l'algorithme d'expansion de la clé chargé de calculer les octets de clé de ronde KX_j à partir de la clé initiale K0

Dans un premier temps, les octets de la clé initiale $K0_j$ sont stockés dans la $SRL0$, en 16 cycles, au signal *start (* $W_i{=}K0_i$ (Équation *2)*. Le calcul de la première ronde peut alors commencer. L'octet lu dans la sortie Q de la $SRL0$, correspond à la valeur stockée dans Qi, accédée par l'adresse AD. Durant les quatre premiers cycles du calcul d'une ronde X (équation 3), une rotation est effectuée sur la troisième colonne de la clé par le biais de la $SRL0$. Une opération de *SubBytes* est effectuée sur les octets à la sortie de Q, suivi d'un XOR avec la constante R_{con} dans le premier cycle. Durant les trois cycles suivants, une autre opération de XOR est effectuée entre résultat de l'opération du *SubBytes* et $\{00\}$; puis l'opération est suivie d'un XOR avec un octet de la colonne précédente obtenu via la sortie $Q15$ de la SRL_0. A partir du quatrième cycle (Équation 4), la sortie Q (octets de la colonne précédente) subit un XOR avec la sortie $Q15$ (octets de la ronde précédente). La Tableau 4 illustre le contenu de la $SRL0$, l'adressage et les sorties cycle par cycle. On note K_j les octets de la clé de ronde courante X et $k_j$les octets de la clé de ronde précédente $X-1$.

Tableau 6 : Adressage et stockage des clé dans la SRL_0 durant le calcul d'une clé de ronde KX

cpt	$Q0$	$Q1$	$Q2$	$Q3$	$Q4$	$Q5$	$Q6$	$Q7$	$Q8$	$Q9$	$Q10$	$Q11$	$Q12$	$Q13$	$Q14$	$Q15$	Q	AD
0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_0	k_{13}	2
1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	k_3	k_2	k_1	k_{14}	2

2	K_1	K_0	k_{15}	K_{14}	K_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	k_3	k_2	k_{15}	2
3	K_2	K_1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	k_3	k_{12}	6
4	K_3	K_2	K_1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	k_4	K_0	3
5	K_4	K_3	K_2	K_1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	k_5	K_1	3
6	K_5	K_4	K_3	K_2	K_1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	k_6	K_2	3
7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	k_{15}	k_{14}	k_{13}	k_{12}	k_{11}	k_{10}	k_9	k_8	k_7	K_3	3
8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_{12}	K_{11}	K_{10}	K_9	K_8	K_4	3
9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_{12}	K_{11}	K_{10}	K_9	K_5	3
10	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_{12}	K_{11}	K_{10}	K_6	3
11	K_{10}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_{12}	K_{11}	K_7	3
12	K_{11}	K_{10}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_{12}	K_8	3
13	K_{12}	K_{11}	K_{10}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{13}	K_9	3
14	K_{13}	K_{12}	K_{11}	K_{11}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{14}	K_{10}	3
15	K_{14}	K_{13}	K_{12}	K_{11}	K_{10}	K_9	K_8	K_7	K_6	K_5	K_4	K_3	K_2	K_1	K_0	K_{15}	K_{11}	3

Cette section conclut la présentation des éléments communs aux architectures proposées. Dans la prochaine section, nous présentons les variantes architecturales étudiées durant nos travaux.

II. Variantes architecturales

1. Architecture AES avec aléa limité

Dans l'architecture AES avec aléa limité, les quatre colonnes de la matrice d'état sont traitées individuellement, dans un ordre aléatoire. L'aléa est ajouté dans l'ordre de traitement des quatre colonnes de la matrice d'état ainsi que des quatre octets au sein de chaque colonne. Cette approche offre un compromis entre la surface, les performances et le nombre de permutations possibles. En limitant les permutations aux octets d'une même colonne, le surcoût en latence est réduit. Pour générer ces permutations, un réseau adéquat a été conçu. Ce dernier étant plus compact, le surcoût en surface est lui aussi réduit. La génération d'une permutation s'appuie sur deux réseaux de Benes 4x4 capables de générer chacun *4!* permutations. Nous

représenterons ce réseau qui se basent sur l'utilisation de SRL sous l'acronyme *2xBen-4_SRL*.

a. Module de brassage

L'architecture du module de brassage est illustrée dans la *Figure 38*. Deux réseaux de Benes sont utilisés pour construire la permutation *P*. Le réseau de Benes 2 permet de générer la permutation de l'ordre de traitement des colonnes *c* de la matrice d'état. Le réseau de Benes 1 est quant à lui utilisé pour générer les permutations de l'ordre de traitement des quatre octets de chaque colonne. Chaque réseau est piloté par un nombre aléatoire codés sur 6 bits *cmd1* et par *cmd2* délivré par le PRNG.

La permutation des quatre colonnes *Pc* est générée par le réseau de Benes 2 piloté par *cmd2* puis stockée dans le registre REG. Ce registre est partitionné en quatre blocs de deux bits.

Le réseau de Benes 1 génère quatre permutations Pc_i pilotées par quatre mots aléatoires successifs de 6 bits *cmd1*. L'ensemble de ces permutations Pc_0, Pc_1, Pc_2, Pc_3 définit l'ordre de calcul des quatre octets des colonnes correspondantes c_0, c_1, c_2, c_3. Ces permutations sont stockées dans les quatre premières adresses de la SRL, en quatre cycles. Les permutations des colonnes *Pc* et des octets Pc_i sont générées et stockées pendant quatre cycles suivant le signal *start* et sont utilisées durant tout le chiffrement.

Dans cette architecture, chaque ronde est calculée en 20 cycles d'horloge. Le signal *cpt*, issu du contrôleur, permet de sélectionner à chaque cycle, via deux multiplexeurs, l'octet à calculer parmi les 16 octets de la matrice d'état de l'AES.

La colonne à calculer c_i est sélectionnée à l'aide d'un multiplexeur MUX2 situé à la sortie du registre REG (voir *Figure 38*). Ce multiplexeur est piloté par les deux bits de l'itération du compteur *cpt[3:2]* (section du signal composé des bits 3 à 2 inclus), itéré tous les quatre cycles. La sortie du multiplexeur MUX2 représente donc, l'indice de la colonne de la matrice d'état sélectionné :

$$c_i = P[cpt[3:2]][3:2] = Pc[cpt[3:2]]$$

La SRL est pilotée par l'indice de la colonne sélectionnée c_i. On obtient alors l'ordre de traitement des quatre octets de la colonne sélectionnée défini par Pc_i à la sortie de la SRL. Le multiplexeur MUX1, piloté par deux bits du compteur cpt[1:0] itéré à chaque

cycle, permet alors de parcourir cette permutation. La sortie du multiplexeur MUX1 représente l'indice de l'octet de la colonne $b_j=Pc_i[cpt[1:0]]$.

La permutation *P* est la concaténation des deux bits de poids fort à la sortie du MUX1 (la colonne), ainsi que des deux bits de poids faible à la sortie du MUX2 (l'octet). On note $P[cpt[3:0]] = Pc[cpt[3:2]] \| Pc_i[cpt[1:0]]$ l'indice de l'octet calculé selon la permutation P où Pc[cpt[3:2]] représente les quatre bits de point fort (l'indice de la colonne courante) et Pc_i[cpt[1:0]] représente les quatre bits de poids faible (l'indice de l'octet à calculer).

Le module de brassage délivre en sortie la permutation courante correspondante à l'itération du compteur *cpt* qui définit l'octet à calculer ainsi que la permutation de la colonne courante Pc_i, utilisée dans le calcul de l'opération du MixColumns. En effet, l'opération du MixColumns calcule les quatre octets de la colonne courante en parallèle c_i et *a* alors besoin de connaître l'ordre des permutations de celle-ci à l'avance.

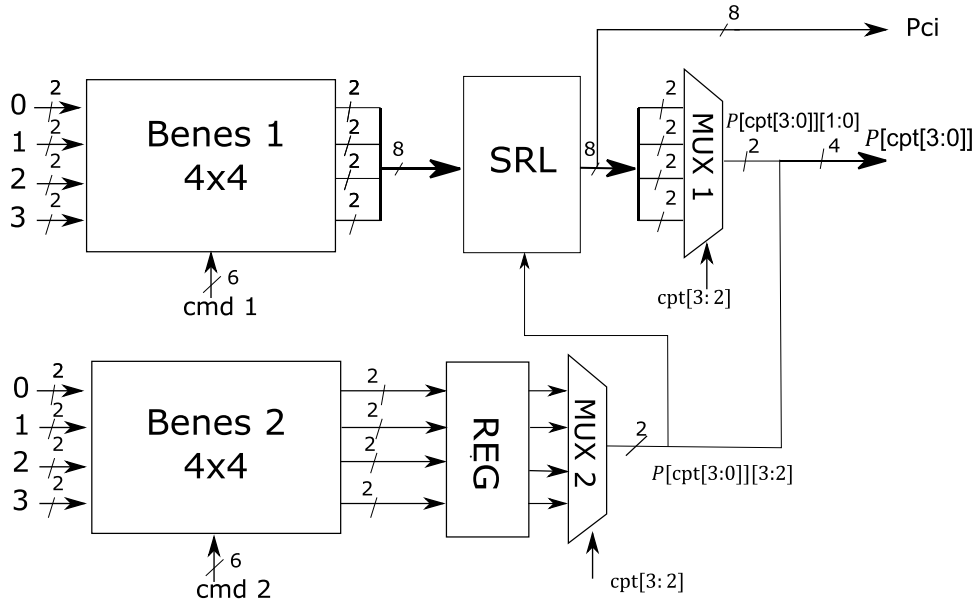


Figure 38: Architecture du module de brassage pour l'architecture 2xBen-4_SRL.

Exemple :

Une première permutation Pc est utilisée pour définir l'ordre de traitement des colonnes :

> La permutation Pc=[3,1,2,0] est générée par le réseau de Benes 2 piloté par le mot aléatoire cmd2. La permutation est stockée dans le registre REG.

Quatre permutations successives sont ensuite générées et stockées dans la mémoire SRL :

La permutation de la colonne 0 : Pc_0=[0,2,3,1], cmd1=$(010100)_2$ est stockée à l'adresse 0 de la SRL.

La permutation de la colonne 1 : Pc_1=[0,3,1,2], cmd1=$(101000)_2$ est stockée à l'adresse 1 de la SRL.

La permutation de la colonne 2 : Pc_2=[0,1,2,3], cmd1=$(010001)_2$ est stockée à l'adresse 2 de la SRL.

La permutation de la colonne 3 : Pc_3=[0,2,3,1], cmd1=$(001010)_2$ est stockée à l'adresse 3 de la SRL.

On obtient alors une permutation P des 16 octets de la matrice d'état intermédiaire. La Tableau 8 illustre le résultat des permutations correspondant à chacune des 16 itérations du compteur cpt[3:0].

Tableau 8 : Résultat de la permutation P

$\boldsymbol{cpt[3:0]}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$\boldsymbol{P[3:0]}$	12	14	15	13	4	7	5	6	8	9	10	12	0	2	3	1

Avant de décrire le chemin de données de l'architecture, nous présentons l'ordonnancement des opérations dans la section suivante.

b. Ordonnancement des opérations

L'ordonnancement des opérations de l'AES est présenté en Figure 39. Les traits pointillés noirs verticaux délimitent une période de 16 cycles et se situent entre les itérations 0 et 15 du compteur cpt. Les traits pointillés bleus verticaux se situent entre l'itération 3 et 5 du compteur et les traits pointillés rouges verticaux délimitent une période de 20 itérations et se situent entre l'itération 0 et 19 du compteur. On note w_{Din} l'opération stockage du texte clair, wKX l'opération de stockage de la clé de ronde.

Au signal *start,* le texte clair et la clé initiale ($wD, wK0$) sont stockées en 16 cycles (première partie de la phase d'initialisation). Chaque clé de ronde wKX est calculée et stockée une ronde à l'avance $X-1$.

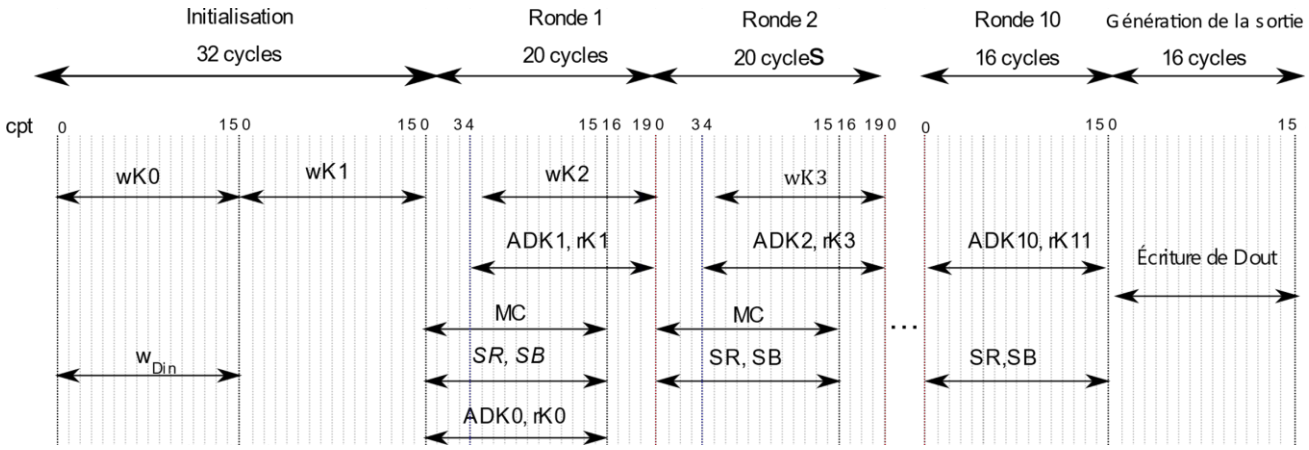


Figure 39 : Ordonnancement des opérations de l'AES pour l'architecture 2xBen-4_SRL

Les opérations ShiftRows (SR), SubBytes (SB) et MixColumns (MC) de chaque ronde sont réalisés en 16 cycles (c.à.d. 1 cycle par octet de la matrice d'état). L'opération *key-whitening* $ADK0$ est effectuée en parallèle de ces opérations durant la première ronde. Chaque colonne de la matrice d'état est calculée en quatre cycles en accumulant à chaque cycle la contribution de l'octet défini par Pci. Ensuite, l'opération $ADK1$ est effectuée en utilisant la clé de ronde $K1$ durant les quatre cycles qui suivent. Par conséquent, 16 cycles sont nécessaires pour réaliser les opérations SR, SB et MC et 4 cycles supplémentaires sont nécessaires pour terminer l'opération d'AddRoundKey ($ADK1$) sur la dernière colonne de la matrice d'état. Lors de la dernière ronde, l'opération du MixColumns est omise : le calcul est alors effectué en 16 cycles.

En raison de l'aléa dans l'ordre de traitement des colonnes, le calcul de la ronde suivante ne peut commencer que lorsque tous les octets de la ronde courante ont été traités.

Une fois le chiffrement terminé, une phase de génération de la sortie permet d'inverser la permutation afin d'écrire le résultat en sortie ($Dout$). Au total, la latence globale incluant l'initialisation, le chiffrement et l'écriture de la sortie est de 244 cycles d'horloge. Dans la section suivante, nous présentons l'architecture du chemin de données.

c. Chemin de données

Le chemin de données est illustré dans la Figure 40. $SRL2$ et $SRL1$ sont utilisées comme des mémoires échangeant leurs rôles en ping-pong à chaque ronde via le

pilotage du MUX2. Lorsqu'une mémoire est utilisée pour le stockage du résultat de la ronde courante, la seconde est lue pour la réalisation des opérations de la ronde courante. Le bon adressage de cette dernière assure l'opération de ShiftRows.

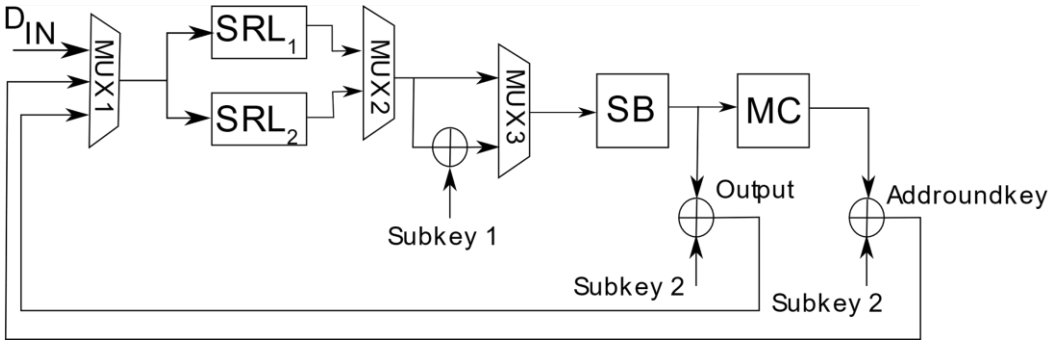


Figure 40: Chemin de données de l'architecture 2xBen-4_SRL

Dans un premier temps, le texte clair est stocké dans la mémoire $SRL1$ durant les 16 premiers cycles de la phase d'initialisation. Cette SRL sera alors utilisée en lecture durant la première ronde. Le pilotage du $MUX3$ permet de réaliser l'opération du *key-whitening* lors de la première ronde uniquement. Par la suite, l'opération du SubBytes (SB) est effectuée directement. Cette dernière est implémentée dans des mémoires distribuées nécessitant 8 Slices. S'en suit l'opération du MixColumns (MC) présentée en détail dans la section suivante. Le calcul de la ronde se termine par l'opération d'AddRoundKey $ADKX$ à l'aide d'un octet de clé de ronde KX délivré par le module d'expansion de la clé.

Lors de la dernière ronde, le MUX1 est piloté de manière à omettre l'opération MixColumns. Dans la phase de génération de la sortie, le résultat est stocké dans la mémoire et est écrit en sortie D_{OUT}. Enfin, il est important de noter que le bon adressage des $SRLs$ est assuré via les signaux générés par le module de brassage.

d. Module MixColumns « MC »

L'opération du *MixColumns* est le résultat d'une multiplication matricielle entre une colonne de la matrice d'état AES et une matrice prédéfinie (cf. Équation 5) :

Équation 5: Opération matricielle du MixColumns

$$\begin{bmatrix} B_{c,0} \\ B_{c,1} \\ B_{c,2} \\ B_{c,3} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Chaque octet de la colonne de sortie $[B_{c,0}B_{c,1}B_{c,2}B_{c,3}]$ est dépendant des quatre octets en entrée $[b_{c,0}b_{c,1}b_{c,2}b_{c,3}]$, on obtient alors les équations suivantes :

Équation 6: Calcul de $B_{c,0}$

$$B_{c,0} = 2b_{c,0} \oplus 3b_{c,1} \oplus b_{c,2} \oplus b_{c,3}$$

Équation 7 : Calcul de $B_{c,1}$

$$B_{c,1} = b_{c,1} \oplus 2b_{c,1} \oplus 3b_{c,2} \oplus b_{c,3}$$

Équation 8 : Calcul de $B_{c,2}$

$$B_{c,2} = b_{c,0} \oplus b_{c,1} \oplus 2b_{c,2} \oplus 3b_{c,3}$$

Équation 9 : Calcul de $B_{c,3}$

$$B_{c,3} = 3b_{c,0} \oplus b_{c,1} \oplus b_{c,2} \oplus 2b_{c,3}$$

Dans une implémentation AES sur un chemin de données de 8 bits, les octets sont multipliés dans l'espace de Galois par les coefficients correspondants et accumulés dans quatre registres (R_0, R_1, R_2, R_3).

Le chemin de données de l'architecture est présenté dans Figure 41. Les quatre octets b_{cj} de la colonne courante ci sont acheminés dans ce module en quatre cycles dans l'ordre défini par la permutation P_{ci}. Par la suite, ils sont multipliés par les coefficients $\{03\}$ et $\{02\}$ et accumulés dans les quatre registres R_0, R_1, R_2 et R_3 contenant initialement une valeur nulle. La permutation P_{ci} est gérée via les multiplexeurs M_0, M_1, M_2 et M_3 orientant la contribution de l'octet courant vers le bon registre d'accumulation. Les multiplexeurs $M_{c,j}$ prennent en entrée le résultat des multiplications des octets ainsi que l'octet non multiplié $b_{c,j}$.

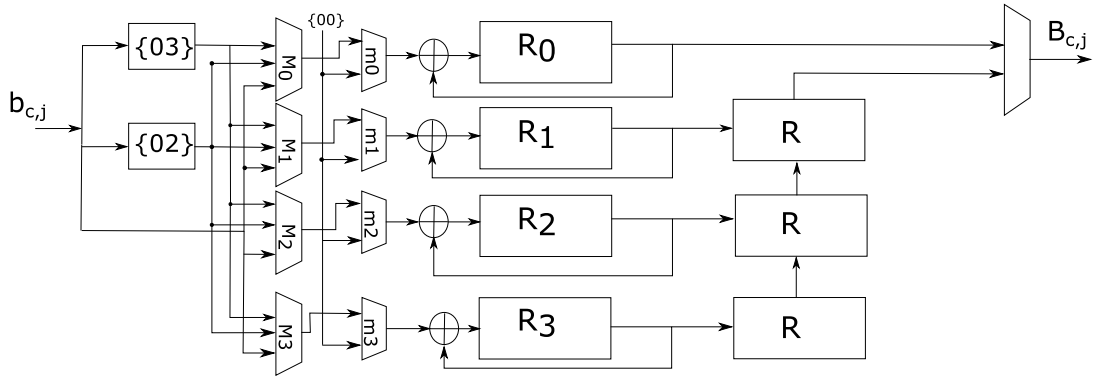


Figure 41 : Architecture du module MixColumns de l'architecture 2xBen-4_SRL

Le pilotage des multiplexeurs $M_{c,j}$ se base sur le lien entre les coefficients de multiplication et la valeur du signal $cpt[1:0]$. Dans le cas d'une architecture sans ajout d'aléa, le calcul est itératif comme illustré dans la Figure 42. Les octets "rouge" $b_{c,j}$ sont multipliés par deux (ce qui correspond à la matrice identité). Dans le cas de la multiplication par trois, la liaison entre le compteur cpt et les coefficients de multiplication est définie par une table de correspondance T présentée dans le Tableau 7 et est utilisée pour identifier l'indice de l'octet "bleu" concerné par cette multiplication.

Tableau 7: Table des dépendances de la multiplication par 3

$\boldsymbol{cpt[1:0]}$	**0**	**1**	**2**	**3**
$\boldsymbol{T}$	3	0	1	2

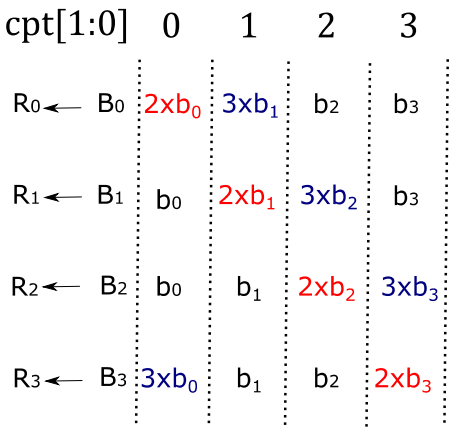


Figure 42 : Ordonnancement du calcul du MixColumns cycle par cycle

Dans le cas de l'architecture 2xBen-4_SRL, la sortie d'un multiplexeur M_{cj} correspond au résultat de la multiplication par $\{02\}$ si l'indice de l'octet assigné à ce multiplexeur, c.-à-d. $Pci[j]$, est égal à l'indice de l'octet courant à calculer (deux bits de poids faible de la permutation courante $P[cpt]$).

Sinon la valeur de sortie correspond au résultat de la multiplication par {3} si l'indice de l'octet assigné à ce multiplexeur $Pci[j]$ est égal à la valeur dans le tableau T correspondante à l'indice de l'octet courant $cpt[1:0]$.

Enfin, dans tous les autres cas, elle correspond aux données non multipliées $b_{c,j}$.

À la cinquième itération du compteur $cpt[1:0] = 0$, les résultats du calcul de la colonne, stockés dans les registres R_0, R_1, R_2, R_3 sont alors stockés dans les registres R. Les

données sont ensuite décalées en sortie via les registres R dans l'ordre de permutation Pci.

Exemple :

La permutation de la colonne courante Pci est illustrée dans le *Tableau 8* :

Tableau 8 : Exemple de permutation Pci

$\boldsymbol{Cpt[1:0]}$	**0**	**1**	**2**	**3**
$\boldsymbol{P_{ci}}$	2	0	1	3

Les résultats des multiplications des octets b_{cj} sont accumulés dans les registres R_0, R_1, R_2, R_3, de telle sorte à ce que : $R_0 \leftarrow B_{c,2}, R_1 \leftarrow B_{c,0}, R_2 \leftarrow B_{c,1}, R_3 \leftarrow B_{c,3}$.

Au premier cycle $cpt[1:0] = 0$**,** $P[cpt][1:0] = 2, T[2] = 1$:

Multiplexeur M_0 :$Pci[0] = 2$:$R_2 = 2 \times b_{c,2} \oplus R_2$.

Multiplexeur M_1 :$Pci[1] = 0$, aucune égalité n'est satisfaite :$R_0 \leftarrow b_{c,2} \oplus R_0$.

Multiplexeur M_2 :$Pci[2] = 1$:$R_1 \leftarrow 3 \times b_{c,2} \oplus R_1$.

Multiplexeur M_3 :$Pci[3] = 3$, aucune égalité n'est satisfaite :$R_3 \leftarrow b_{c,2} \oplus R_3$.

Au deuxième cycle $cpt[1:0] = 1$**,** $P[cpt][1:0] = 0$**,**$T[0] = 3$**:**

Multiplexeur M_0 :$Pci[0] = 2$, aucune égalité n'est satisfaite :$R_2 \leftarrow b_{c,0} \oplus R_2$.

Multiplexeur M_1 :$Pci[1] = 0$:$R_0 \leftarrow 2 \times b_{c,0} \oplus R_0$.

Multiplexeur M_2 :$Pci[2] = 1$, aucune égalité n'est satisfaite :$R1 \leftarrow b_{c,0} \oplus R_1$.

Multiplexeur M_3 :$Pci[3] = 3$:$R_3 = 3 \times b_{c,0} \oplus R_3$.

Au troisième cycle $cpt[1:0] = 2$**,** $P[cpt][1:0] = 1$**,**$T[0] = 0$**:**

Multiplexeur M_0 :$Pci[0] = 2$, aucune égalité n'est satisfaite :$R0 \leftarrow b_{c,1} \oplus R_0$.

Multiplexeur M_1 :$Pci[1] = 0$:$R_1 \leftarrow 3 \times b_{c,1} \oplus R_1$.

Multiplexeur M_2 :$Pci[2] = 1, R_2 \leftarrow 2 \times b_{c,1} \oplus R_2$.

Multiplexeur M_3 :$Pci[3] = 3$, aucune égalité n'est satisfaite, $R_3 \leftarrow b_{c,1} \oplus R_3$.

Au quatrième cycle $cpt[1:0] = 3$**,** $P[cpt][1:0] = 3$**,**$T[0] = 2$**:**

Multiplexeur M_0 :$Pci[0] = 2$:$R_0 \leftarrow 3 \times b_{c,3} \oplus R_0$.

Multiplexeur M_1 :$Pci[1] = 0$, aucune égalité n'est satisfaite :$R_1 \leftarrow b_{c,3} \oplus R_1$.

Multiplexeur M_2 :$Pci[2] = 1$, aucune égalité n'est satisfaite :$R_2 \leftarrow b_{c,3} \oplus R_2$.

Multiplexeur M_3 :$Pci[3] = 3$:$R_3 \leftarrow 2 \times b_{c,3} \oplus R_3$.

Le contenue des registres est alors :

$R_0 \leftarrow B_{c,2}$ (Équation 8)

$R_1 \leftarrow B_{c,0}$ (Équation 6)

$R_3 \leftarrow B_{c,3}$ (Équation 9)

$R_2 \leftarrow B_{c,1}$(Équation 7)

e. Module d'expansion de la clé « KeyScheduling »

En raison de l'aléa introduit dans le calcul, tous les octets de la clé d'une ronde X doivent être calculés et stockés avant le début de cette dernière. Par conséquent, la clé de la ronde $X + 1$ est calculée en parallèle du calcul de la ronde X. Les résultats du calcul sont stockés dans l'une des deux mémoires SRL_1 et SRL_2 utilisées en *Ping-Pong* pour la lecture et l'écriture de la clé de ronde. La Figure 43 présente l'architecture du module d'expansion de la clé pour l'architecture 2xBen-4_SRL.

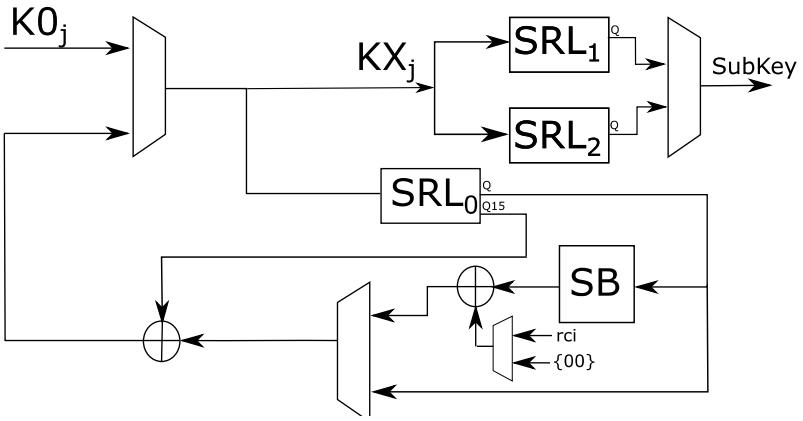


Figure 43 : Module KeyScheduling de architecture 2xBen-4_SRL

2. Architectures avec aléa complet

Dans cette approche, l'aléa est ajouté sur l'ordre de traitement des 16 octets de la matrice d'état. Le FPGA offre quatre alternatives de stockage : les registres, les SRLs, les mémoires distribuées et les blocs mémoires (BRAM). Dans ces travaux de thèse, nous avons utilisé deux de ces alternatives pour concevoir des variantes : la première utilise des SRLs et la seconde utilise des mémoires distribuées.

Dans une SRL, les 16 octets se décalent au sein des éléments de stockage durant les 16 cycles pendant lesquels ils sont stockés. L'adresse de chaque octet dépend alors de l'ordre dans lequel ce dernier a été inséré dans la SRL. Dans une architecture où l'aléa est ajouté dans l'ordre de traitement des données, l'adressage d'un octet sera donc dépendant de la permutation utilisée lors du traitement.

Les mémoires distribuées quant à elles permettent l'ajout (ou non) de l'aléa dans le stockage. La première méthode consiste à stocker le résultat d'un octet B_j à l'adresse j indépendamment de son ordre de traitement (ou *adresse fixe*). La seconde consiste à choisir une adresse aléatoire. Cette dernière peut par exemple correspondre à son ordre de traitement défini par la permutation P généré à chaque chiffrement.

Dans ces travaux, trois alternatives ont été conçues :

- la première utilise des SRLs résultant ainsi en un aléa dans le stockage que nous représenterons avec le suffixe *_SRL* ;
- la deuxième utilise des mémoires distribuées avec des adresses fixes, que nous représenterons avec le suffixe *_Mem* ;
- la dernière utilise des mémoires distribuées avec aléa sur les adresses, que nous représenterons avec le suffixe *_MemBrass*.

De plus, pour chaque alternative deux versions ont été conçues : la première utilise un réseau de Benes 16x16 permettant d'obtenir toutes les permutations possibles (16 !) nommée *Ben16* ; la deuxième utilise un réseau Omega 16x16, qui permet d'offrir 2^{32} nommée *Omeg16.*

La Figure 44, illustre la classification de ces six versions.

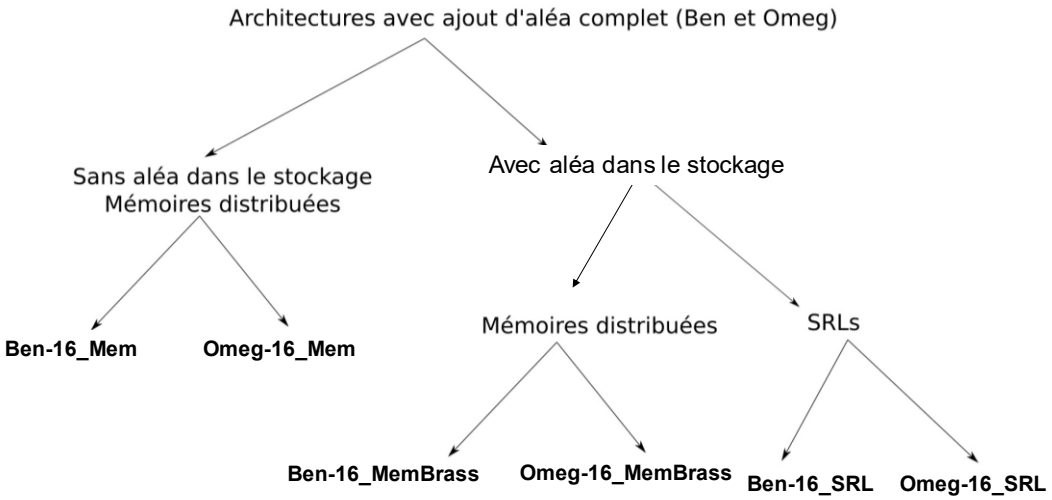


Figure 44 : Classification des architectures mono bloc

a. Module de brassage

L'architecture du module de brassage est illustrée dans la Figure 45. Elle est commune aux versions basées sur un réseau de permutation 16x16. Ce dernier permet de générer la permutation P en un seul cycle d'horloge au début de chaque chiffrement. Cette permutation est alors parcourue à l'aide d'un multiplexeur MUX, piloté par l'itération $cpt[3:0]$ généré par le contrôleur. La sortie du multiplexeur est l'indice de l'octet à calculer $P\left[cpt[0:3]\right]$.

Dans la version Ben-16, le réseau implémenté est un réseau de Benes 16x16, composé de 56 commutateurs (112 multiplexeurs 2 vers 1 de 4 bits) et contrôlé par un mot de commande cmd de 56 bits. Pour la version Omeg-16, le réseau implémenté est un réseau Omega 16X16 composé de 32 commutateurs (64 multiplexeurs 2 vers 1 de 4 bits) et contrôlé par un mot de commande cmd de 32 bits.

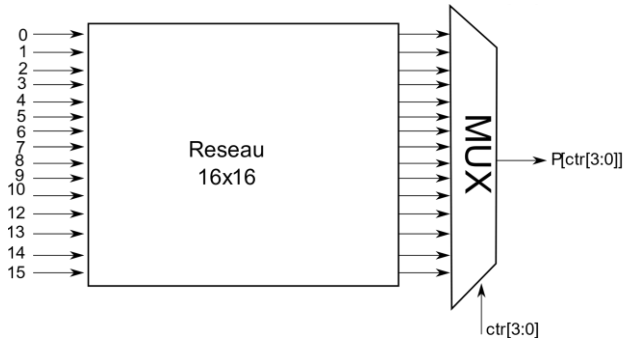


Figure 45 : Architecture du module Shuffling des architecture Ben-16 et Omeg-16

Dans la section suivante nous présentons l'ordonnancement des opérations relatif à cette approche.

b. Ordonnancement des opérations

L'ordonnancement des opérations de l'AES est présenté en *Figure 46*. Les traits pointillés noirs verticaux délimitent une période de 16 cycles et se situent entre les itérations 0 et 15 du compteur cpt. Au signal *start,* le texte clair et la clé initiale ($wD, wK0$) sont stockés en 16 cycles (première partie de la phase d'initialisation). Chaque clé de ronde wKX est calculée et stockée dans la première phase de la ronde courante X à l'exception de la première et de la dernière ronde.

L'ajout d'aléa dans les opérations implique une gestion particulière des calculs lors du MixColumns. En effet, celui-ci ne peut s'effectuer qu'après la fin du calcul des opérations indépendantes sur les seize octets de la matrice d'état. Le calcul de chaque ronde est alors effectué en deux phases :

- dans la première phase, les opérations ShiftRows (SR), SubBytes (SB) et la première partie du MixColumns ($MC[1/2]$) de chaque ronde sont réalisés en 16 cycles (c.-à-d. 1 cycle par octet de la matrice d'état). En parallèle, la clé de ronde est calculée wKX. L'opération key-whitening $ADK0$ est effectuée en parallèle de ces opérations durant la première ronde ;
- dans la seconde phase, le calcul de la ronde est complété par la seconde partie du MixColumns ($MC[2/2]$ et l'opération d'AddRoundKey ($ADKX$). Lors de la dernière ronde, l'opération du MixColumns est omise, le calcul est alors effectué en 16 cycles.

Une fois le chiffrement terminé, une phase de génération de la sortie permet d'inverser la permutation afin d'écrire le résultat en sortie ($Dout$). Au total, la latence globale incluant l'initialisation, le chiffrement et l'écriture de la sortie est de 336 cycles d'horloge. Dans la section suivante, nous présentons l'architecture du chemin de données.

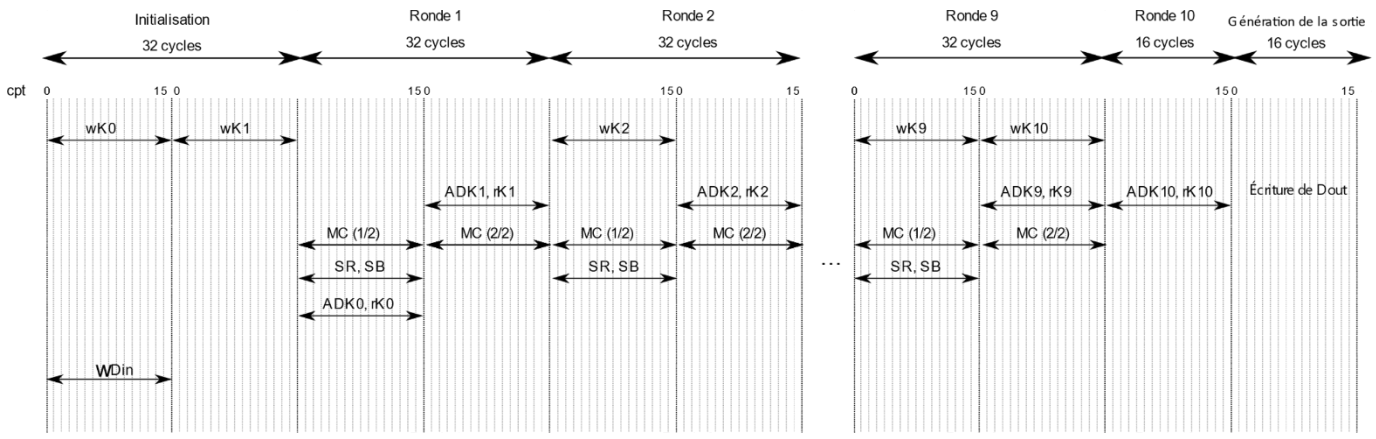


Figure 46 : Ordonnancement des opérations des architectures Ben-16 et Omeg-16

c. Chemin de données

Le chemin de données est illustré dans la *Figure 47*. La mémoire MEM_i est utilisée dans le stockage des données intermédiaires. L'opération de ShiftRows est assurée par le bon adressage de cette dernière. La mémoire MEM_o est utilisée dans le stockage du résultat de la dernière ronde à la fin du chiffrement. Les mémoires MEM_0 et MEM_i sont implémentée par des SRL dans les architectures Ben-16_SRL et Omeg-16_SRL et avec des mémoires distribuées dans les architectures Ben-16_Mem, Ben-16_MemBrass, Omeg-16_Mem, Omeg-16_MemBrass.

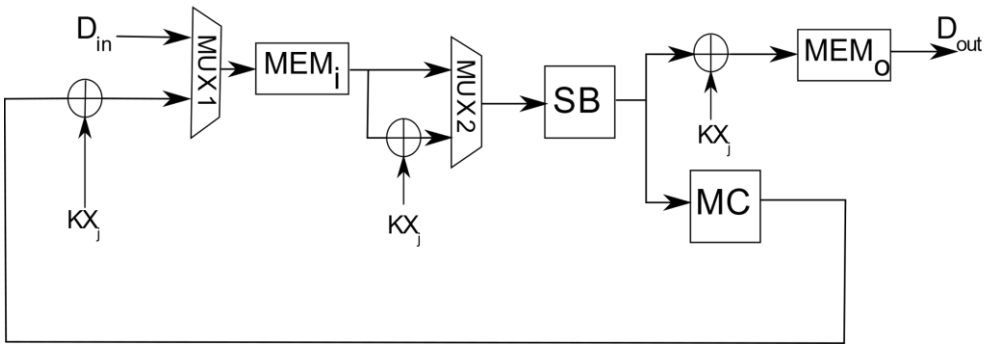


Figure 47 : Chemin de données des architectures Ben-16 et Omeg-16

Dans un premier temps, le texte clair est stocké dans MEM_i durant les 16 premiers cycles de la phase d'initialisation. Le pilotage du $MUX2$ permet de réaliser l'opération du key-whitening lors de la première ronde uniquement. Par la suite, l'opération du SubBytes (SB) est effectuée directement. Cette dernière est implémentée dans des mémoires distribuées nécessitant 8 slices. S'en suit l'opération du MixColumns présentée en détail dans la section suivante. Le calcul de la ronde se termine par l'opération d'AddRoundKey $ADKX$ qui consiste en une opération de XOR avec l'octet

de clé de ronde KX_j délivré par le module d'expansion de la clé. Lors de la dernière ronde, le résultat de l'opération d'AddRoundKey $ADKX$ qui suit celle du SubBytes est stocké dans MEM_0. Dans la phase de génération de la sortie, les octets stockés dans la mémoire sont écrits en sortie D_{OUT}.

L'ajout ou non d'aléa dans le stockage dépend des adresses délivrées par le contrôleur quand les mémoires distribuées sont implémentées. Quand l'aléa est ajouté dans le stockage, les données sont mémorisées aux adresses relatives à l'ordre de traitement permuté et les données sont retrouvées en utilisant la table de permutation inverse P^{-1}. Dans le cas contraire, les octets sont stockés aux adresses correspondantes à leur indice dans la matrice d'état. La table de permutation inverse n'est alors pas requise.

d. Module MixColumn

Dans l'opération du MixColumns l'opération se fait en deux temps : Dans un premier temps, les résultats de la multiplication dans $GF(2^8)$ des 16 octets de la matrice d'état intermédiaire sont acheminés dans l'ordre de la permutation P et sont ensuite stockés dans les mémoires MEM_0,MEM_1,MEM_2 et MEM_3 implémentées par des SRLs ou des mémoires distribuées selon l'architecture (cf. *Figure 48*).

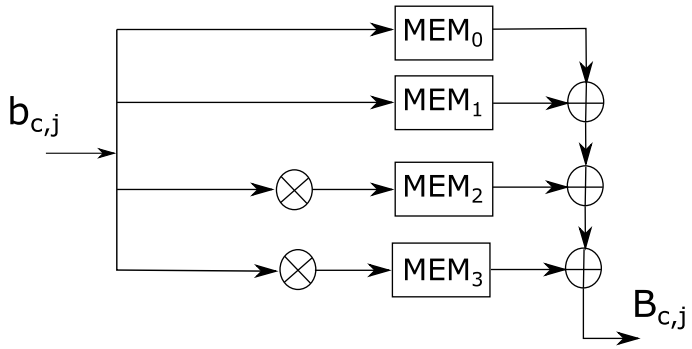


Figure 48 : Architecture du MixColumns des architectures Ben-16 et Omeg-16

Dans un second temps, chaque octets $B_{c,j}$ est calculé via une opération XOR sur quatre octets de la même colonne ci définie par les deux bits de poids fort de l'indice de l'octet courant de la matrice. Le bon calcul est assuré par le bon adressage des SRLs ou des mémoires distribuées contenant les résultats des multiplications dans $GF(2^8)$.

Les adresses de lecture Adi dans les quatre mémoires MEM_0, MEM_1, MEM_2et MEM_3 sont retrouvées à l'aide de quatre tableaux $A0, A1, A2, A4$. Ces tableaux stockent les dépendances entre les octets à calculer $B_{c,j}$ et les quatre octets $b_{c,j}$ de la colonne courante c_i sont stockés.

Ainsi les deux bits représentant l'indice de l'octet de la colonne à calculer $B_{c,j}$ sont utilisés comme indice dans les quatre tableaux. La valeur *A3* du *Tableau 12* correspond à l'octet $b_{c,j}$ (multiplication par $\{03\}$ stockée dans MEM_3). La valeur $A2$ du *Tableau 11* correspond à l'octet b_{cj} (multiplication par $\{02\}$ stockée dans MEM_2). Les valeurs $A0$ du Tableau 9 et $A1$ du *Tableau 10* correspondent aux deux octets restant non multipliés et stockées dans MEM_0 et MEM_1.

L'adresse de lecture dans la mémoire MEM_i est définie par l'indice de la colonne à calculer $P\left[cpt[3:2]\right]$ concaténé avec l'indice de l'octet correspondant retrouvé dans le tableau A_i : $Ad_i = P\left[cpt[3:2]\right] \parallel Ai\left[P\left[cpt[1:0]\right]\right]$.

Tableau 9 : Tableau des dépendances des octets utilisée dans MEM_0

$\boldsymbol{P\left[cpt[1:0]\right]}$	**0**	**1**	**2**	**3**
A0	2	0	0	1

Tableau 10 : Tableau des dépendances des octets utilisée dans MEM_1

$\boldsymbol{P\left[cpt[1:0]\right]}$	**0**	**1**	**2**	**3**
A1	3	3	1	2

Tableau 11 : Tableau des dépendances des octets utilisée dans MEM_2

$\boldsymbol{P\left[cpt[1:0]\right]}$	**0**	**1**	**2**	**3**
A2	0	1	2	3

Tableau 12 : Tableau des dépendances des octets utilisée dans MEM_3

$\boldsymbol{P\left[cpt[1:0]\right]}$	**0**	**1**	**2**	**3**
A3	1	2	3	0

Exemple :

Considérons le troisième octet de la quatrième colonne $B_{3,2}$, ($P\left[cpt[3:0]\right] = 14$), l'équation correspondante est alors :$b_{3,0} \oplus b_{3,1} \oplus 2b_{3,2} \oplus 3b_{3,3}$ (cf. *Équation 8*).

- $P\left[cpt[3:2]\right] = 2$, $A0[2] = 0$, $Ad_0 = 12$: la valeur en sortie de MEM_0 est $b_{3,0}$.
- $P\left[cpt[3:2]\right] = 2$, $A1[2] = 1$, $Ad_1 = 13$: la valeur en sortie de MEM_1 est $b_{3,1}$.
- $P\left[cpt[3:2]\right] = 2$, $A0[2] = 2$, $Ad_2 = 14$: la valeur en sortie de MEM_2 est $2 \times b_{3,2}$.
- $P\left[cpt[3:2]\right] = 2$, $A1[2] = 3$, $Ad_3 = 15$: la valeur en sortie de MEM_3 est $3 \times b_{3,3}$.

e. Module d'expansion de la clé « Key Scheduling »

En raison de l'aléa dans le calcul, tous les octets de la clé d'une ronde X doivent être calculés et stockés avant le début du calcul de cette dernière. La clé de ronde est utilisée dans la deuxième partie du calcul de chaque ronde, à l'exception de la dernière. Chaque clé de ronde est stockée dans les deux SRLs SRL_0 et SRL_1. La SRL_1 est utilisée dans la lecture la clé initiale $K0$ et les clés de rondes $K2$ à $K9$. La SRL_0 est essentiellement utilisées dans le calcul des clés de ronde.

Les clés calculées KX sont stockées dans cette SRL après chaque opération de calcul wKX. Cette dernière est utilisée temporairement dans les opérations de lecture dans le cas où il y ait deux opérations successives de lecture, comme c'est le cas des couples d'opérations $rK0, rK1$ et $rK9, rK10$. De plus, la lecture de la clé $K9$ est réalisée en parallèle du calcul de la clé de ronde $K10$. Par conséquent, la SRL_1 ne peut être utilisée pour stocker la clé $K10$.

La *Figure 49* présente l'architecture du module d'expansion de la clé pour les architectures de type Ben-16 et Omeg-16.

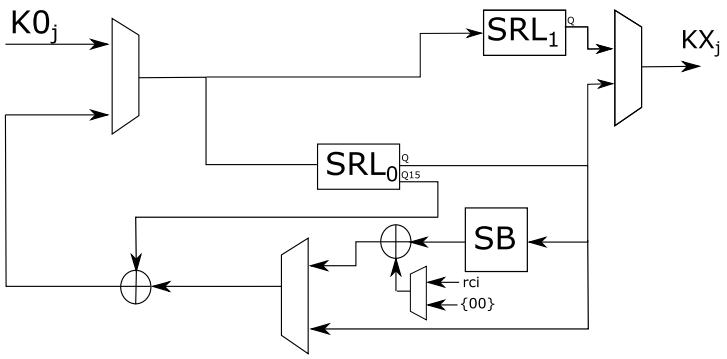


Figure 49: Module d'expansion de la clé "Key scheduling" des architefture Ben-16 et Omeg-16

III. Résultat d'implantation des architectures AES-ECB

Dans le but d'évaluer nos contre-mesures, l'architecture de référence a été mise en œuvre sur une cible FPGA. Dans nos architectures, la bibliothèque d'opérateurs [86] est utilisée, notamment pour les multiplieurs dans $GF(2^8)$.

Les architectures proposées ont été synthétisées sur un circuit Xilinx SPARTAN-6 (xc6slx9 et xc6slx4). Le *Tableau 13* compare les résultats de l'architecture de référence présentée dans [26] implémentée sur une carte Xilinx xc6slx4 et ceux de notre architecture synthétisée sur la même cible ainsi que sur une cible xc6slx9 utilisée dans nos évaluations de sécurité.

Tableau 13: Résultat d'implantation des architectures non protégées

Architecture	**[26]**	**[26] Notre Travail**	**[26] Notre Travail**
Cible FPGA	**xc6slx4**	**xc6slx4**	**xc6slx9**
Fréquence (Mhz)	73	114,81	114,09
Latence	160	160	160
Débit (Mbit/s)	58,13	91,848	91,272
Slice	80	81	81
Slice LUT	n/a	218	218
Slice registres	n/a	97	97
BRAM	n/a	0	0
DSP	n/a	0	0
Débit /Surface	**0,72**	**1,13**	**1,12**

Les résultats obtenus pour ces deux cibles sont quasi similaires pour nos propres implémentations de [26]. Cependant, on remarque que la fréquence est plus élevée dans nos implémentations par rapport à celle présentée dans [26]. Ceci est probablement lié aux améliorations apportées par les versions plus récentes des outils de conception Xilinx utilisés. Dans [26], la version utilisée est XILINX ISE 11.1, tandis que nos architectures ont été implémentées en utilisant XILINX ISE 14.7.

Notre implémentation de l'architecture [26] sur une cible xc6slx9 peut donc être utilisée comme référence pour comparer les architectures proposées dans la suite de ce manuscrit.

Le noyau cryptographique a été réalisé en incluant le réseau de permutation. Le générateur de nombres aléatoires quant à lui n'est pas inclus (mis en œuvre à l'extérieur du noyau cryptographique). Les résultats de synthèse sont présentés dans le *Tableau 14*. La version limitée 2xBen-4_SRL utilise 107 slices ce qui représente un surcoût de 32 % en surface par rapport à l'architecture de référence. Les trois alternatives architecturales[2] avec ajout d'aléa complet utilisant le même réseau Ben-16/Omeg-16 présentent des résultats similaires en surface et performances. On remarque que les architectures présentant le meilleur rapport débit/surface sont celles utilisant des SRLs.

Dans une architecture utilisant des SRLs, le surcoût en termes de slices est de 77 % dans la version Ben-16 et de 30 % dans la version Omeg-16. Les résultats montrent une chute de débit par un facteur de 2,23 et par un facteur de 2.01 dans la version Omeg-16 par rapport à l'architecture de référence.

Dans une architecture utilisant des mémoires distribuées avec aléa sur le stockage (*MemBrass*), le surcoût en termes de slices est de 88 % dans la version Ben-16 et de 40 % dans la version Omeg-16. Les résultats montrent une chute de débit par un facteur de 2,1 et par un facteur de 2,16 dans la version Omeg-16 par rapport à l'architecture de référence.

Enfin dans une architecture utilisant des mémoires distribuées sans aléa sur le stockage (*Mem*), le surcoût en termes de slices est de 76 % dans la version Ben-16 et de 39.56 % dans la version OnC. Les résultats montrent une chute de débit par un facteur de 2,28 dans les deux versions Ben-16 et Omeg-16 par rapport à l'architecture de référence. On remarque que l'ajout d'aléa dans le stockage n'a pas de surcoût majeur en surface et en performances.

[2] Ben-16_SRL, Ben-16_Mem, Ben-16_MemBrass, Omeg-16_SRL, Omeg-16_Mem et Omeg-16_MemBrass

Tableau 14 : Résultat de synthèse des architectures

Architectures (cible FPGA)	[CB12] (xc6slx9)	Ben-16			Omeg-16			2xBen-4_SRL
		(xc6slx9)						
		Ben-16_SRL	Ben-16_MemBrass	Ben-16_Mem	Omeg-16_SRL	Omeg-16_MemBrass	Omeg-16_Mem	2xBen-4_SRL
Fréquence (Mhz)	114,09	107,27	98	104,73	118,63	110,9	104.97	114
Latence	160	336	336	336	336	336	336	228
Débit (Mbit/s)	91,272	40,86	37,33	39,89	45,19	42,24	39,9	64
Slice	81	144	153	143	106	114	109	107
Slice LUT	218	365	365	344	294	293	272	268
Slice Register	97	45	50	49	45	50	49	119
BRAM	0	0	0	0	0	0	0	0
DSP	0	0	0	0	0	0	0	0
Débit/Surface	**1,12**	**0.28**	**0.24**	**0.27**	**0.42**	**0,37**	**0,36**	**0,59**

Le

Tableau 15 détaille le nombre de slices pour chaque module implanté individuellement sur une cible xc6slx9. Le surcoût en surface est principalement causé par le réseau (Benes), le contrôleur (doublé en moyenne) et le MixColumns (triplé en moyenne).

Tableau 15 : Résultats de synthèse par module (en slices)

	[CB12]	SRL	MemBrass	Mem	2xBen-4_SRL
Chemin de donnée	60	79	73	72	72
Key Scheduling	24	28	28	28	14
SubBytes	8	8	8	8	8
MixColumns	8	29	22	19	14
Contrôleur	12	28	28	25	28
Réseau Benes	-	38	38	38	10*
Réseau Omega	-	15	15	15	-

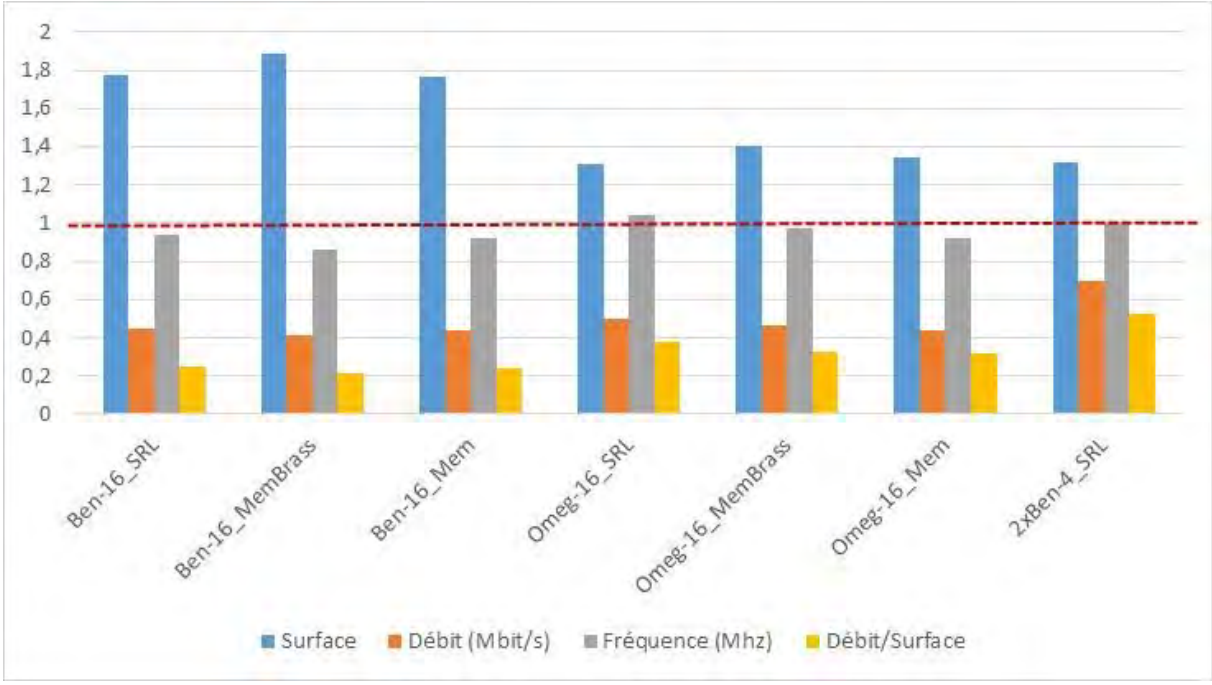


Figure 50 : Coût des architectures mono-bloc normalisé par rapport à [26] *en termes de surface et de performance*

La Figure 50 présente les caractéristiques des architectures mono-bloc protégées en terme de surface, de performances et d'efficacité, normalisées par rapport à l'architecture de référence *[26]* représentée par le trait rouge en pointillé. Les résultats en performances sont similaires pour les architectures Ben-16 et Omeg-16. Ces architectures atteignent 44 % du débit et 90 % de la fréquence de l'architecture de référence. Les résultats en terme de surface sont globalement similaires dans les architectures utilisant un même réseau : un facteur 1,8 pour les architectures Ben-16 et 1,35 pour les architectures Omeg-16. L'architecture la plus efficace (rapport débit/surface) est l'architecture 2xBen-4_SRL avec un facteur 0,52. Les architectures Omeg-16 ont une efficacité moyenne de 0,34 et les architectures Ben-16 ont une efficacité moyenne de 0,23.

IV. Conclusion

Dans ce chapitre nous avons détaillé les architectures avec aléa sur un bloc de chiffrement AES en mode ECB. En raison des dépendances de données, la latence se retrouve alors assez fortement impacté. Cette dégradation des performances se retrouve diminuée dans l'approche avec ajout d'aléa limité (Architecture 2xBen-4_SRL).

Les architectures les plus efficaces parmi les architectures avec aléa complet sont celles utilisant une SRL. Les choix de conception ont un impact sur la surface et la transition des données et donc sur la consommation de puissance et la fuite d'information via ce canal.

Dans la section suivante nous allons évaluer la robustesse de ces implémentations face aux attaques par observation de la consommation de puissance.

Chapitre 4 Etudes de la robustesse des modèles d’architectures proposés

Dans ce chapitre, nous évaluons la robustesse de nos architectures contre des attaques par analyse de la consommation. En premier lieu, nous présentons le banc de test utilisé durant nos expérimentations, ainsi que le résultat de l'attaque sur l'implémentation non protégée de référence. Puis, nous présentons les résultats obtenus lors d'attaques ciblant nos architectures protégées. Enfin, nous étudions l'impact des options de synthèse sur la robustesse des architectures proposées face aux attaques considérées.

Les architectures présentées dans le chapitre précédent ont fait l'objet d'une étude intensive qui ne sera pas présentée ici. L'objet était d'une part d'évaluer le fonctionnement de nos architectures, et d'autres part de réaliser des premières évaluations de sécurité [97].

I. Description de notre cadre expérimental

Afin de mettre en place nos attaques, nous avons utilisé le kit CW1200 (ChipWhisperer Pro) pour capturer les traces de consommation de puissance. Ce kit inclut un Convertisseur Analogique-Numérique (CAN) de 10 bits et 105 MS/s (Mega-Samples par seconde), avec une taille de buffer de 98 119 échantillons. Le kit inclut également une carte cible FPGA Spartan 6 S6LX9 avec un package TQFP. La capture des traces et les attaques ont été réalisées en utilisant la version 5 de la chaîne d'outils ChipWhisperer. Pour évaluer nos contre-mesures, nous avons opté pour la métrique MTD - *Measurement To Discosure* pour nos attaques de type CPA - *Correlation Power Analysis*. Le modèle choisi est celui du poids de Hamming.

La *Figure 51* illustre le banc de test utilisé. Un PC est relié au ChipWhisperer via un câble USB. Le ChipWhisperer est quant à lui connecté à la cible FPGA, elle-même embarquée sur une carte mère (CW308) dédiée aux attaques par canaux cachés. Cette carte mère permet d'interfacer la carte cible et le ChipWhisperer. L'interface est principalement constituée d'un bus de 20 broches permettant l'échange d'information et un connecteur SMA (Vco) permettant la mesure.

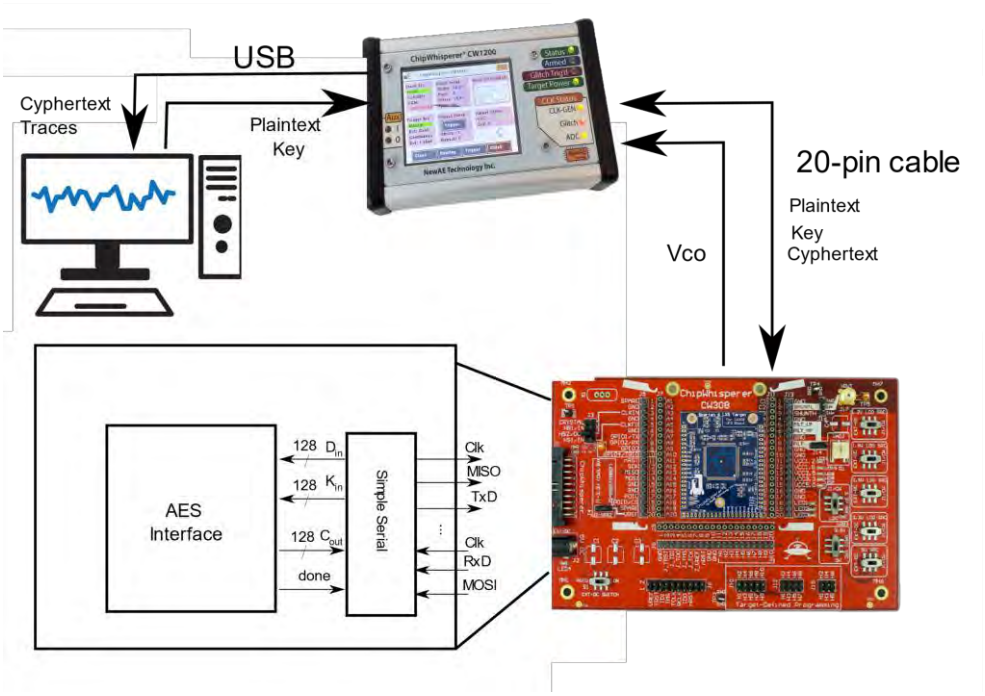


Figure 51 : Banc de test ChipWhisperer

Après la validation du fonctionnement de l'architecture, les traces de consommation de puissance sont collectées par l'ordinateur. Les principaux paramètres de l'oscilloscope du ChipWhisperer pour nos expérimentations sont les suivants : la fréquence de la cible est fixée à 7 384 KHz et celle de l'ADC est fixée à 29 538 KHz. La fréquence d'échantillonnage est donc 4 fois plus grande que la fréquence de la cible (FPGA). Ainsi, pour chaque cycle, on obtient 4 points de mesure. De plus, le gain est fixé à 30dB. Le nombre de traces collectées est configuré pour chaque campagne de mesure et les textes clairs sont générés aléatoirement. La clé quant à elle est fixe et générée aléatoirement. Les attaques CPA sont réalisées en ciblant la dernière ronde de l'algorithme AES, on l'occurrence la 10^{ième} ronde.

II. Architecture non protégée (NP)

Dans cette section, les architectures sont mises en œuvre sur la cible en utilisant les paramètres d'optimisation Xilinx ISE par défaut : optimisation en vitesse avec un effort normal, sans mise à plat de l'architecture, choix d'utilisation des mémoires (BRAM/mémoire distribuée) et des DSP laissé à l'outil. Pour rappel, dans cette version un chiffrement complet est effectué en 160 cycles. En ajoutant les 12 cycles de mémorisation des octets du texte clair dans les registres avant le début du calcul, la latence globale est de 172 cycles. Compte tenu de la fréquence d'échantillonnage, le

nombre de points minimum à capturer est de 688 (quatre fois le nombre de cycles). Cependant, nous avons choisi d'en capturer 800 afin de mieux délimiter les opérations de chiffrement.

La *Figure 52* illustre une trace de consommation résultante d'un tel chiffrement. L'axe des abscisses représente le numéro de l'échantillon et l'axe des ordonnées l'image de la puissance consommée. Les dix rondes du chiffrement sont indiquées par des flèches, le chiffrement commence vers le point 50 et se termine vers le point 700. Les fuites ciblées se situent entre les cycles 600 et 750 (le calcul des rondes 9 et 10).

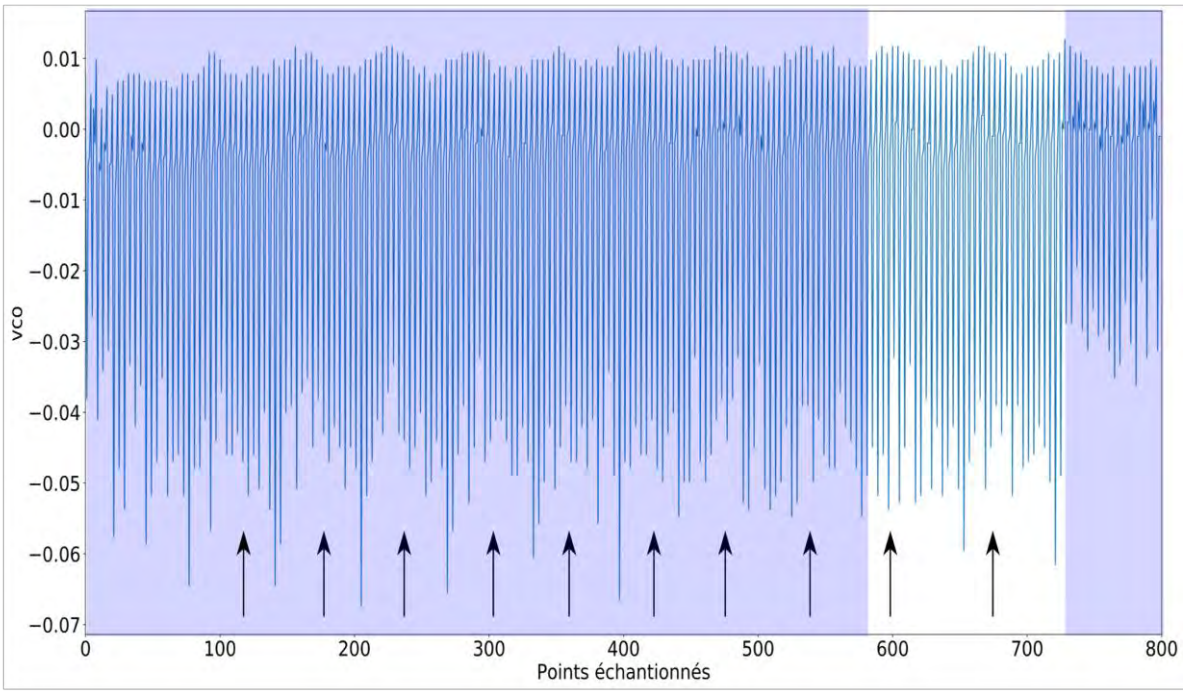


Figure 52 : Traces de consommation de puissance d'un chiffrement sur l'architecture NP

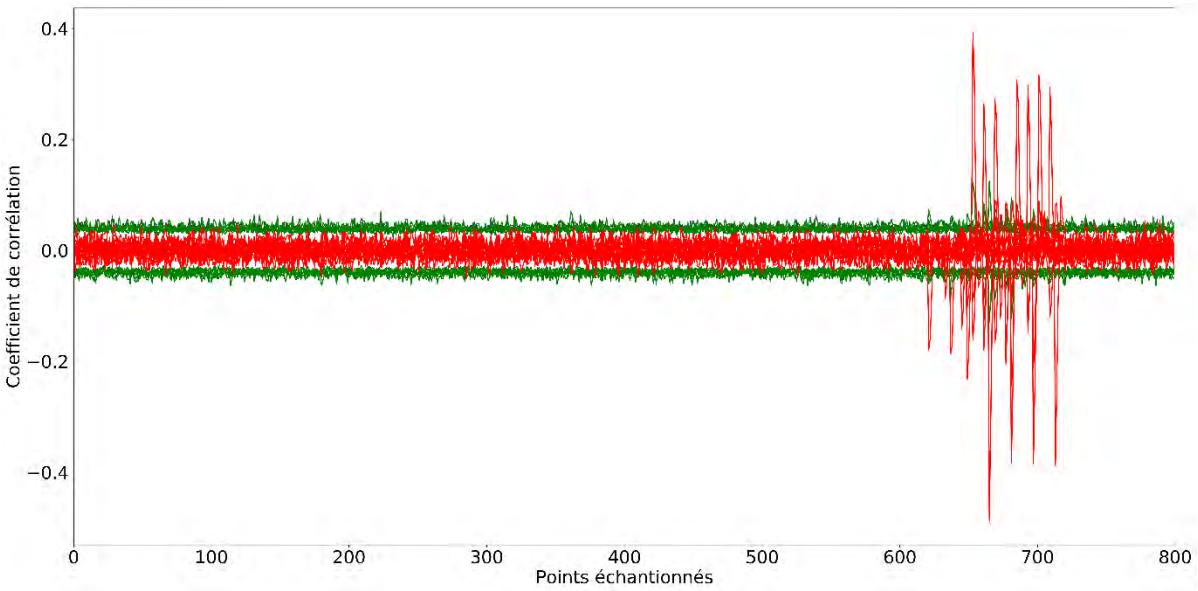


Figure 53 : Courbe de l'attaque CPA pour l'architecture NP

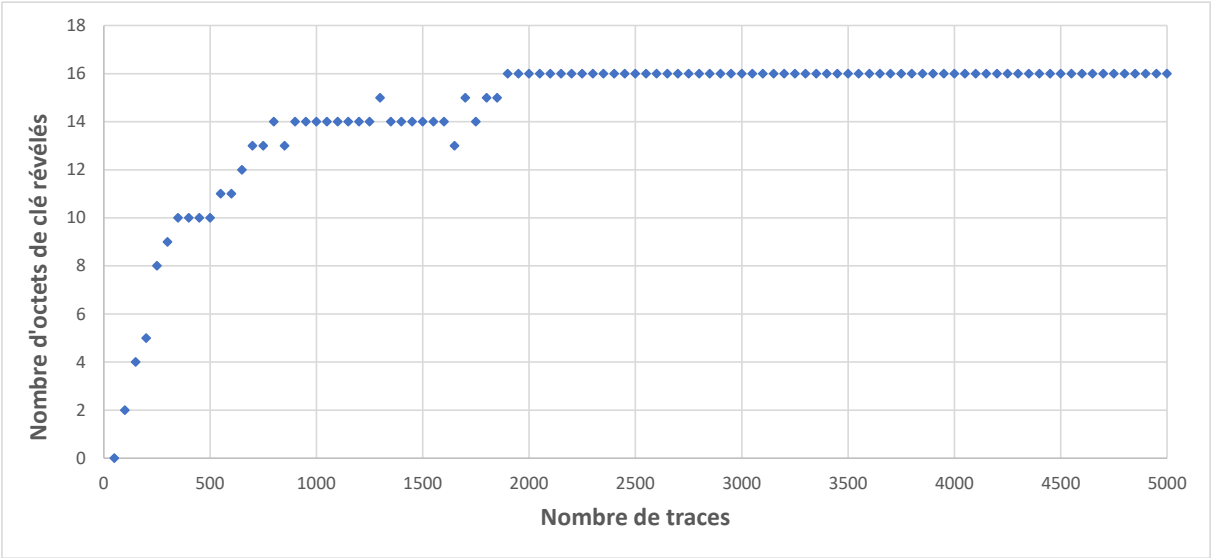


Figure 54 : Résultats de l'attaque CPA sur l'architecture NP (Nombre d'octets révélés en fonction du nombre de traces)

Une première attaque CPA a été réalisée, en utilisant 5000 traces, une clé fixe « 95 24 05 8F DE DB A9 9A B1 8F 16 FF 11 5C 32 72 » et des textes clairs aléatoires. L'entrée de la *Sbox* de la ronde *10* est alors ciblée, et le modèle choisi est celui du poids de Hamming.

La *Figure 53* illustre le résultat de l'attaque CPA : l'axe des abscisses représente le nombre d'échantillons et l'axe des ordonnées représente les coefficients de corrélation

correspondant. Les courbes rouges sont les courbes CPA des 16 octets des bonnes hypothèses de clé, les courbes vertes sont les courbes CPA correspondantes aux mauvaises hypothèses. On remarque des pics avec des coefficients de corrélation dont la valeur absolue est comprise entre 0,2 et 0,4. Ces pics se situent entre les points 600 et 730. Cet intervalle correspond aux traitements réalisés entre la deuxième phase de la neuvième ronde (les octets de la neuvième ronde sont stockés dans la mémoire après l'opération d'AddRoundKey) et la première phase de la dixième ronde (les octets de la neuvième ronde sont lus dans l'opération du ShiftRows). Le nombre d'octets collectés en fonction du nombre de traces est illustré dans la *Figure 54*. L'axe des abscisses représente le nombre de traces utilisées dans l'attaque et l'axe des ordonnées représente le nombre d'octets révélés.

On constate que tous les octets de la clé pour l'architecture non protégée ont été révélés avec moins de 2000 traces. Cette première expérience a permis de valider notre banc de test avec les paramètres qui ont été choisis. Le résultat obtenu est utilisé comme référence dans l'évaluation de rapport de sécurité de nos architectures protégées.

Dans la section suivante, nous présentons les résultats de l'attaque CPA sur l'architecture 2xBen-4_SRL dans laquelle l'aléa et les permutations sont limités afin d'offrir un compromis entre les performances et le nombre de permutation.

III. Etude de la robustesse de nos modèles

1. Architecture 2xBen-4_SRL

Pour rappel, au sein de l'architecture 2xBen-4_SRL, deux réseaux de Benes 4X4 sont utilisés pour générer l'ordre de traitement des colonnes, ainsi que ceux des octets au sein de chaque colonne de la matrice d'état. Ainsi, une colonne est sélectionnée tous les quatre cycles selon la permutation générée par le premier réseau de Benes. Durant ces quatre cycles, les octets au sein de cette colonne sont traités dans l'ordre de traitement défini par la permutation générée par le second réseau de Benes.

La *Figure 55* illustre une trace de consommation de puissance contenant 1200 échantillons. On remarque que le chiffrement est effectué entre les points 100 et 1000, intervalle dans lequel on peut distinguer les 10 rondes de l'AES. La latence globale est de 244 cycles, le nombre de points échantillonnés et analysés sur cet

intervalle est de 976. Une attaque CPA a été mise en place en utilisant 30 000 de ces traces.

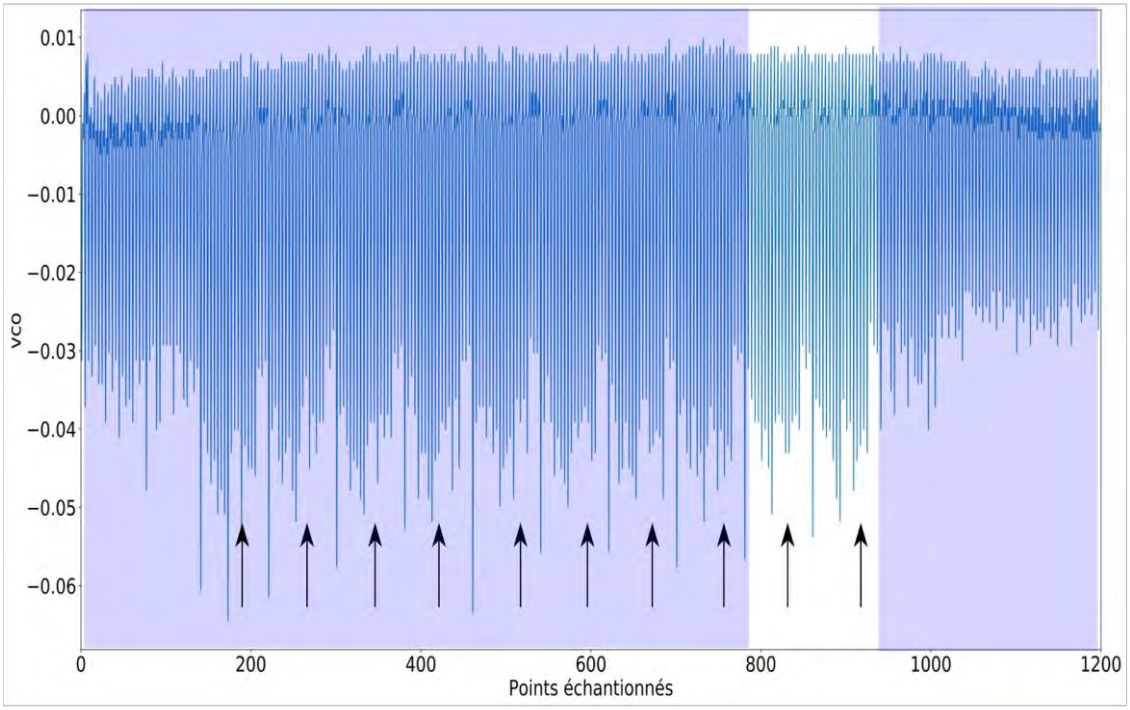


Figure 55 : Traces de consommation de puissance d'un chiffrement sur l'architecture 2xBen-4_SRL.

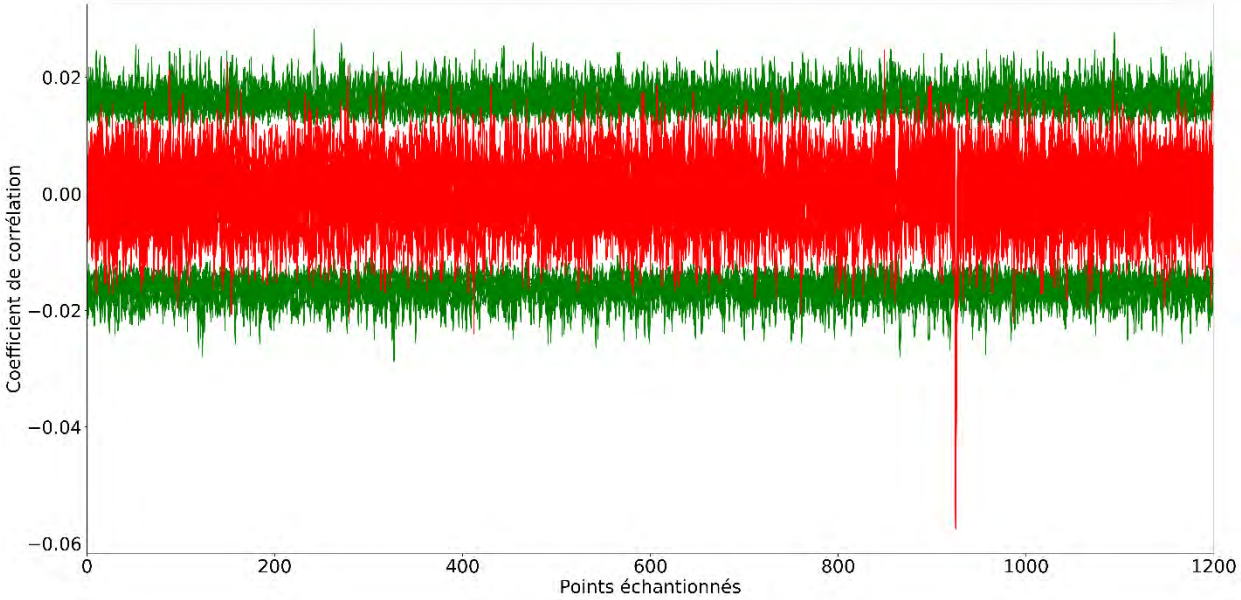


Figure 56 : Courbe CPA pour l'architecture 2xBen-4_SRL

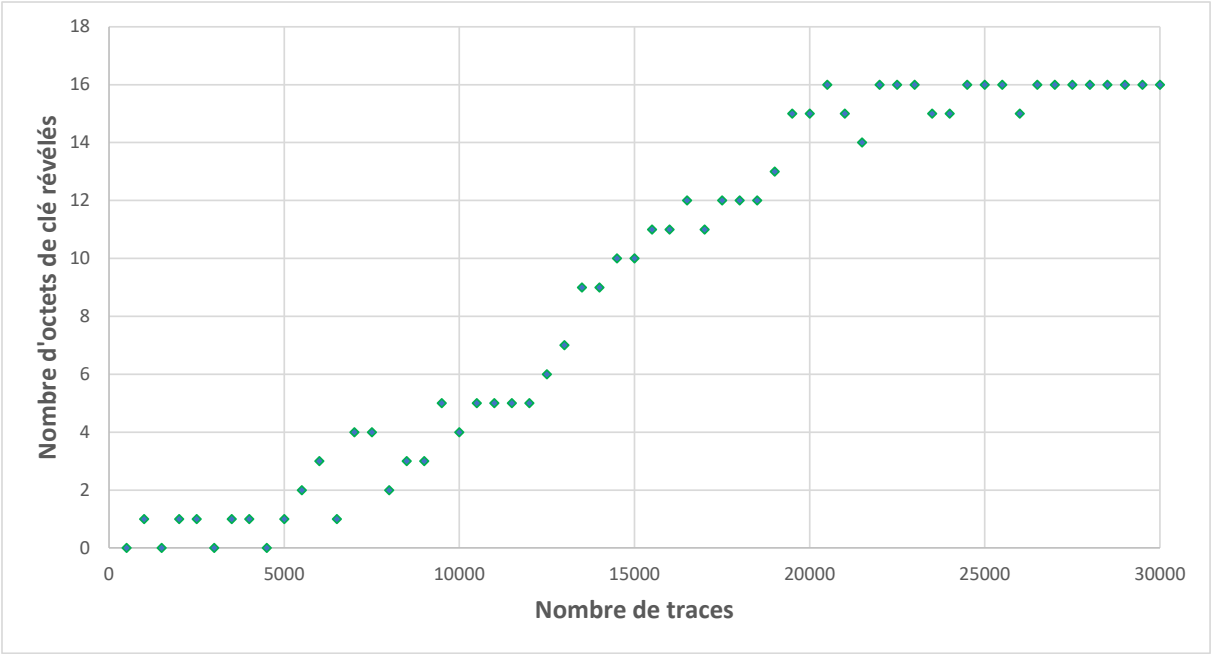


Figure 57 : Résultats de l'attaque CPA sur l'architecture 2xBen-4_SRL

Le résultat de l'attaque est illustré dans la *Figure 56*. On remarque que la fuite exploitée se situe dans un faible intervalle avec des coefficients de corrélation entre 0,048 et 0,031 qui sont 6 à 8 fois inférieurs aux coefficients obtenus dans l'attaque sur l'architecture de référence. Plus particulièrement, la fuite se situe entre l'opération AddRoundKey de la neuvième ronde (16 cycles) et le calcul de la dixième ronde (16 cycles). Le nombre d'octets révélés en fonction du nombre de traces est illustré dans la *Figure 57* : les 16 octets de la clé sont révélés avec 30 000 traces. On a alors un coefficient de sécurité de 15. Ce coefficient représente le rapport entre le MTD obtenu avec l'implémentation protégée et le MTD de l'implémentation non protégée.

Bien que l'apport en sécurité soit insuffisant, l'approche engendre un surcoût en surface modéré (32%) ainsi qu'une faible baisse des performances temporelle (x1.2) par rapport à l'architecture de référence. Dans la section suivante, nous présentons les résultats de l'attaque CPA sur les architectures Ben-16/Omeg-16 pour lesquelles l'ajout d'aléa est maximisé.

2. Architectures Ben-16/Omeg-16

Dans cette section, nous présentons les résultats de l'attaque CPA sur les architectures avec ajout d'aléa complet, en utilisant les réseaux de Benes et Omega (architectures Ben-16 et Omeg-16) pour générer les permutations dans

chacune des variantes : stockage sur des mémoires SRL avec aléa sur les adresses, stockage sur des mémoires distribuées avec aléa sur les adresses et stockage dans des mémoires distribuées sans aléa sur les adresses.

Les traces de consommation de puissance pour chacune des variantes Ben-16 et Omeg-16 sont illustrées dans les *Figure 58*, *Figure 59* et *Figure 60*. La *Figure 58* illustre les traces pour un chiffrement utilisant des SRLs pour le stockage et avec un aléa sur le stockage (Ben-16_SRL et Omeg-16_SRL). La *Figure 59* illustre les traces de consumation pour les architectures avec mémoires distribuées et aléa sur les adresses (Ben-16_MemBrass et Omeg-16_MemBrass). Enfin, la *Figure 60* illustre les traces de consumation pour les deux architectures avec mémoire distribuées sans aléa sur les adresses (Ben-16_Mem et Omeg-16_Mem). Chacune de ces traces contient 1500 échantillons. La latence du chiffrement pour ces architectures est de 336 cycles. Par conséquent, le nombre d'échantillons nécessaires pour capturer un chiffrement complet est estimé à 1344. Les 10 rondes sont distinguables pour les deux architectures utilisant des mémoires distribuées entre les points 50 et 1400 (illustrées par des flèches dans les *Figure 59* et *Figure 60*), tandis que dans l'architecture SRL la consommation est plus faible et les rondes ne sont pas facilement distinguables à l'œil nu. La consommation des deux variantes est par ailleurs similaire.

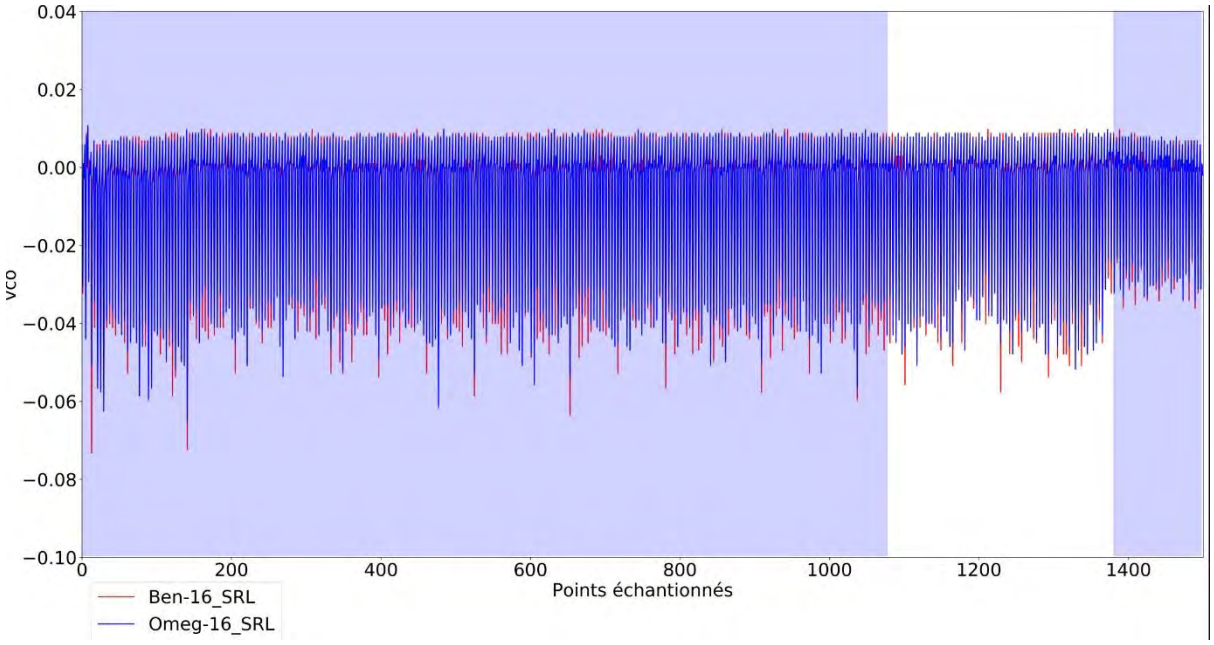


Figure 58 : Traces de consommation de puissances des architectures Ben-16_SRL et Omeg-16_SRL

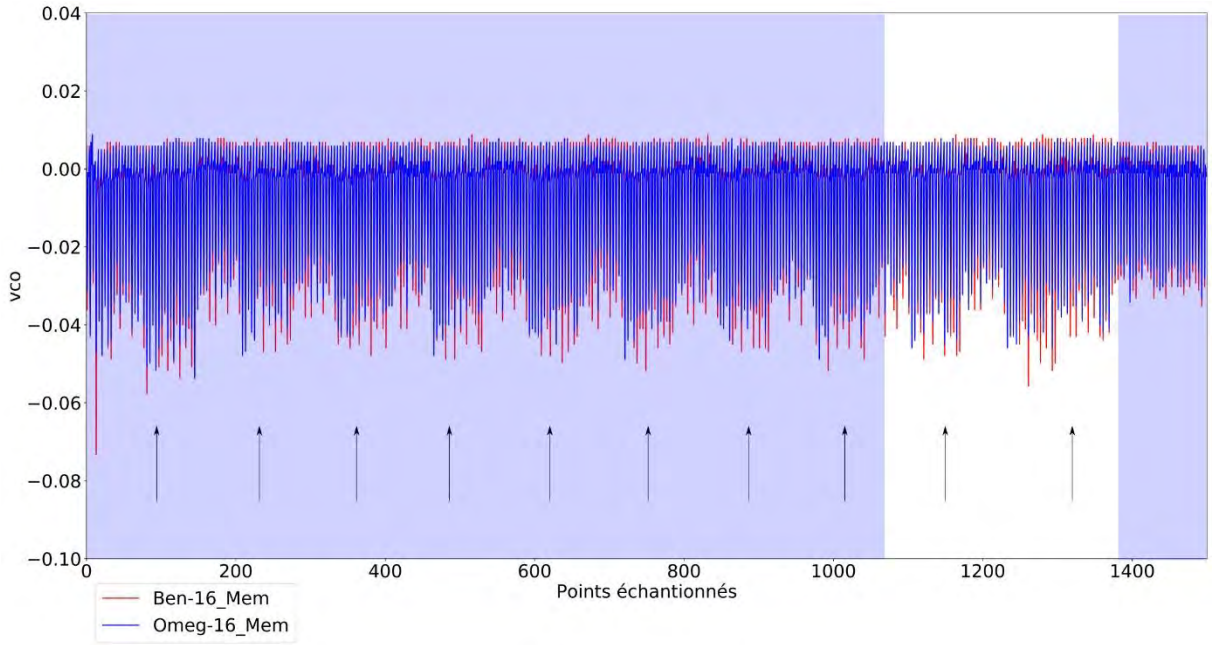


Figure 59 : Traces de consommation de puissance des architectures Ben-16_Mem et Omeg-16_Mem

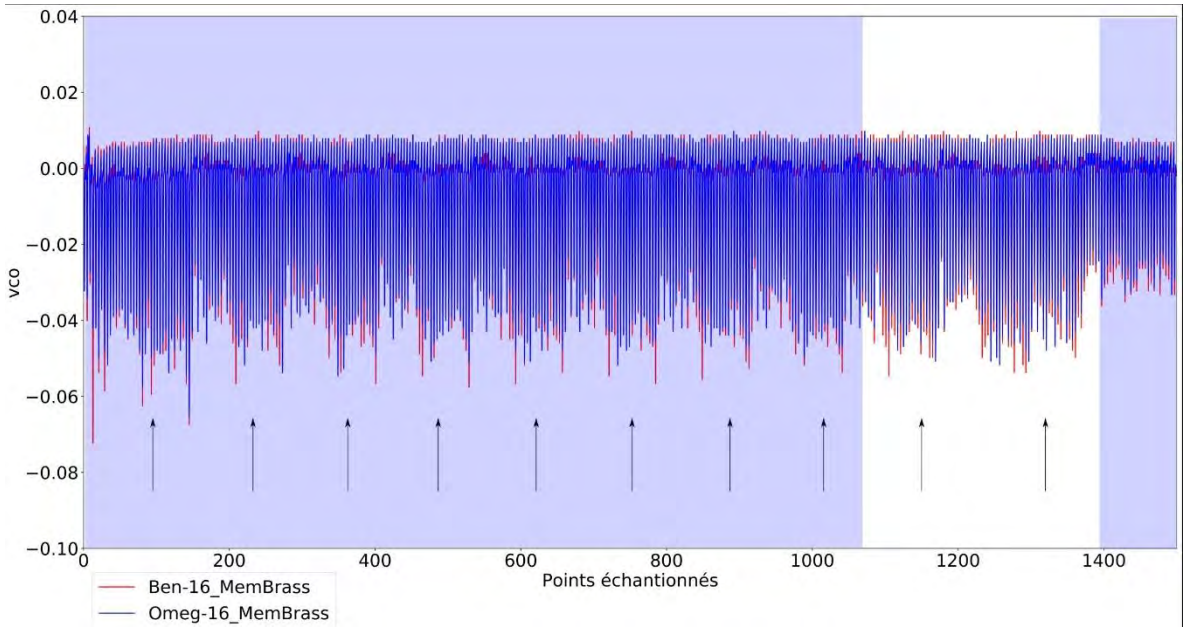


Figure 60 : Traces de consommation de puissance des architectures Ben-16_MemBrass et Omeg-16_MemBrass

Nous présentons dans les prochaines sections les résultats des attaques CPA sur les trois variantes à base de réseaux de Benes (Ben-16).

a. Attaque CPA sur les architectures Ben-16

Les *Figure 61*, *Figure 62* et *Figure 63*, illustrent le résultat de l'attaque CPA avec un million de traces sur les trois variantes utilisant un réseau de Benes. On remarque que

les fuites se situent entre les points 1200 et 1400 où le calcul de la neuvième ronde se situe. Les coefficients de corrélation maximaux des hypothèses correctes varient entre 0,0041 et 0,074 dans l'architecture avec SRL (cf. *Figure 61*) ; entre 0,0035 et 0,18 dans l'architecture Ben-16_Mem (cf. *Figure 62*) ; et entre 0,0033 et 0,0074 dans l'architecture Ben-16_MemBrass (cf. *Figure 63*).

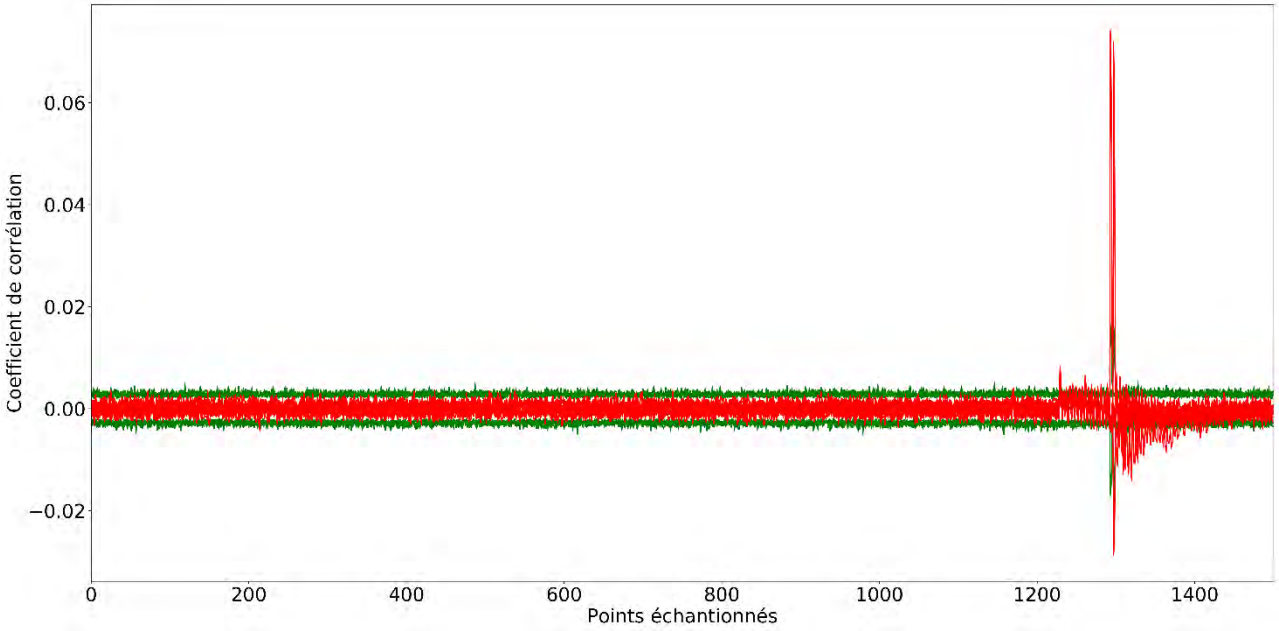


Figure 61 : Courbe CPA pour l'architecture Ben-16_SRL

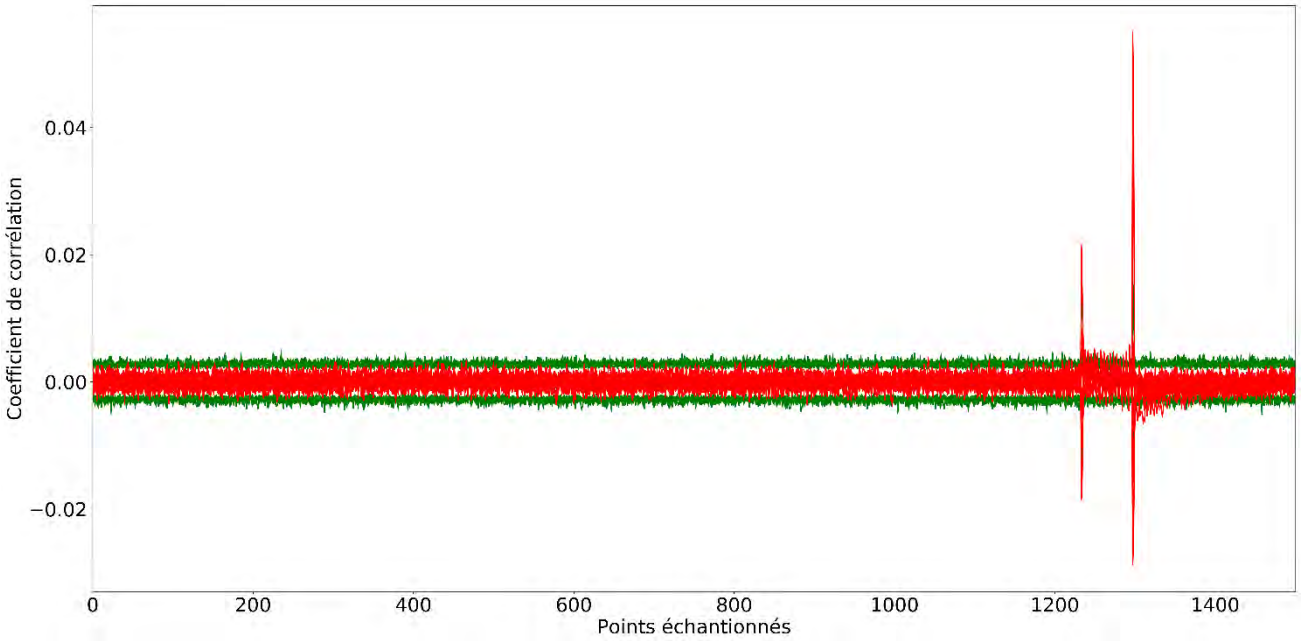


Figure 62 : Courbe CPA pour l'architecture Ben-16_Mem

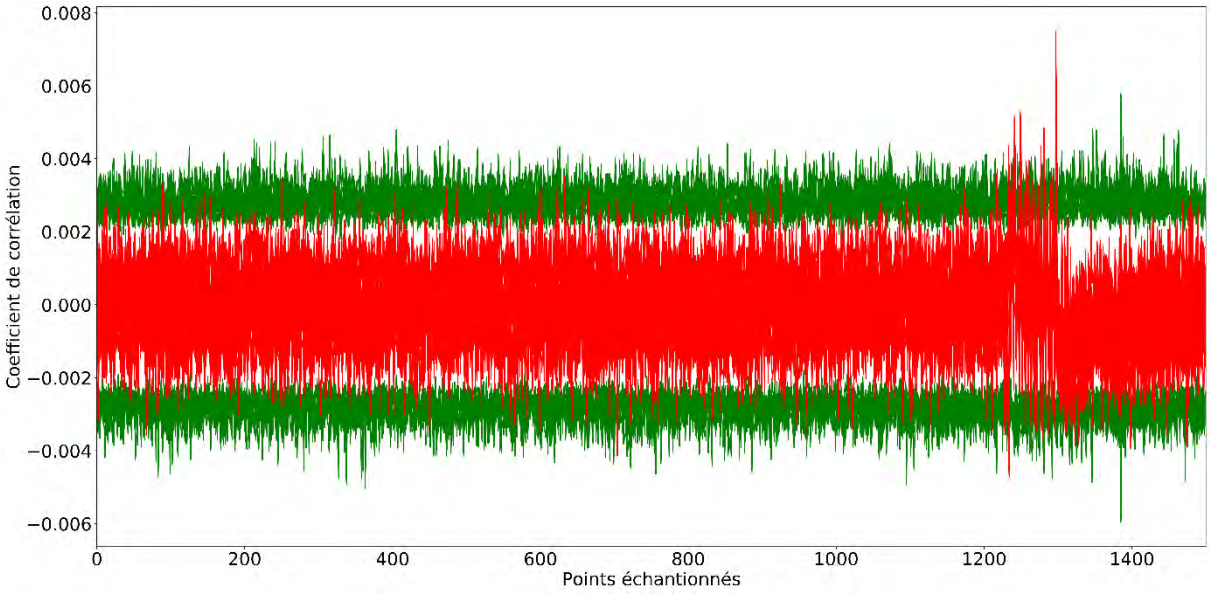


Figure 63 : Courbe CPA pour l'architecture Ben-16_MemBrass

La *Figure 64* illustre les résultats des attaques CPA sur les trois variantes Ben-16 en indiquant le nombre d'octets révélés en fonction du nombre de traces collectées. L'architecture la plus robuste est celle utilisant des mémoires distribuées avec aléa sur les adresses. Avec un million de traces, 6 octets de la clé ont été retrouvés pour l'architecture Ben-16_MemBrass. Les architecture Ben-16_SRL et Ben-16_Mem, quant à eux présentent des résultats similaires. En effet, une attaque sur l'architecture Ben-16_SRL avec un million de traces permet de révéler 13 octets de la clé dont 50 % ont été retrouvés avec environ 500 000 traces.

Dans la variante pour laquelle le stockage est basé sur des mémoires distribuées sans aléa sur les adresses (Ben-16_Mem), il a fallu un million de traces pour retrouver 13 octets de la clé dont 50 % ont été retrouvés avec environ 400 000 traces.

Les 16 octets de la clé n'ont pas été révélés avec un million de traces, cependant nous considèrons arbitrairement que l'attaque est réussie quand 50% de 16 octets de la clé sont retrouvés (c.-à-d. 8 octets) puisque les octets restants peuvent être retrouvés en utilisant une attaque par force brute en un temps raisonnable. Par ailleurs, 50% des octets de la clé dans l'architecture non protégée sont révélés avec environ 250 traces. Le coefficient de sécurité est alors de 1600 dans l'architecture Ben-16_Mem et de 2000 dans l'architecture Ben-16_SRL. Dans l'architecture Ben-16_MemBrass le facteur est supérieur à 4000.

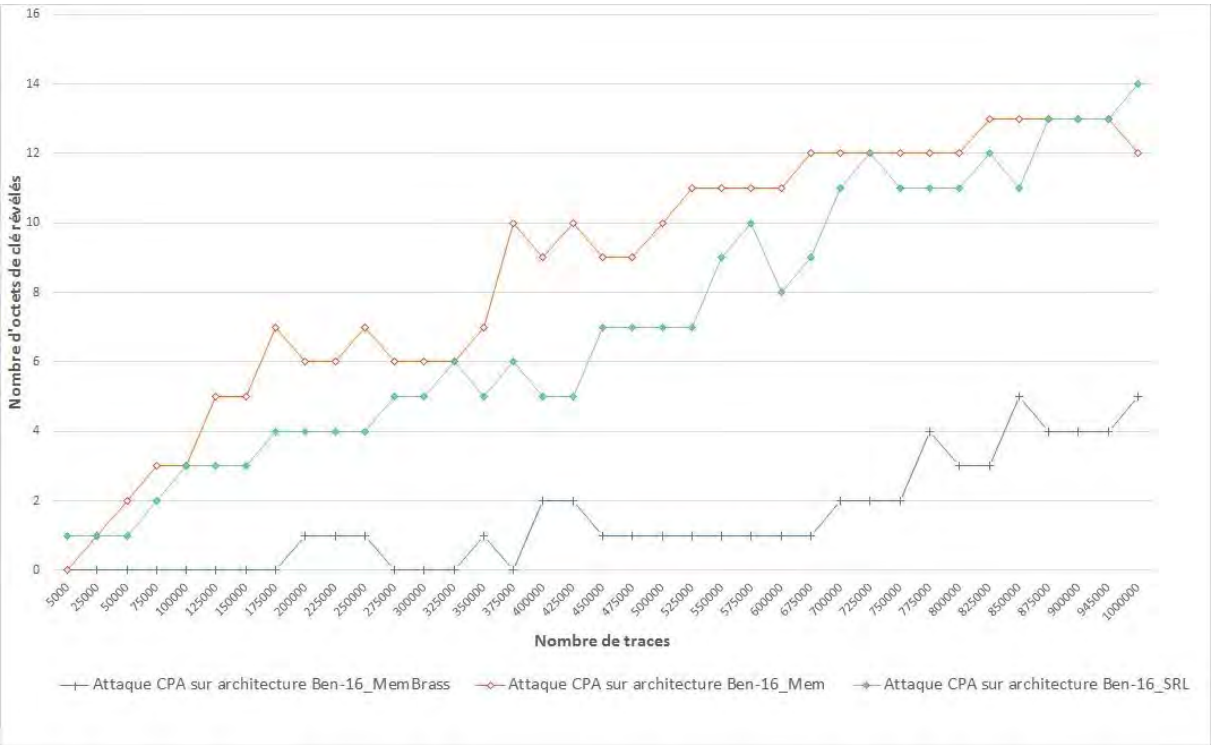


Figure 64 : Résultats de l'attaque CPA des architectures trois variantes Ben-16 en fonction du nombre d'octets révélés

Les résultats des attaques CPA sur les trois variantes Ben-16 ont montré que l'architecture la plus robuste est l'architecture Ben-16_MemBrass. La moitié des octets de la clé n'a pas pu être révélée. On considère donc que l'architecture est protégée contre une attaque CPA avec un million de traces.

Dans la section suivante, nous présentons les résultats des attaques CPA sur les trois variantes Omeg-16.

b. Attaque CPA sur les architectures Omeg-16

Les *Figure 65*, *Figure 66*, *Figure 67* illustrent les résultats de l'attaque CPA avec un million de traces sur les trois variantes utilisant un réseau Omega. On remarque que ces résultats sont similaires à ceux des architectures utilisant un réseau de Benes mais avec des coefficients de corrélation légèrement plus faibles. Dans l'architecture Omeg-16_SRL, (cf. *Figure 65*) les coefficients de corrélation des hypothèses correctes varient entre 0,0053 et 0,06. Dans l'architecture Omeg-16_Mem, les coefficients de corrélation des hypothèses correctes varient entre 0,0035 et 0,038 (cf. *Figure 66*). Dans l'architecture Omeg-16_Mem, les coefficients de corrélation maximale des hypothèses correctes varient entre 0,0031 et 0,0045 (cf. *Figure 67*).

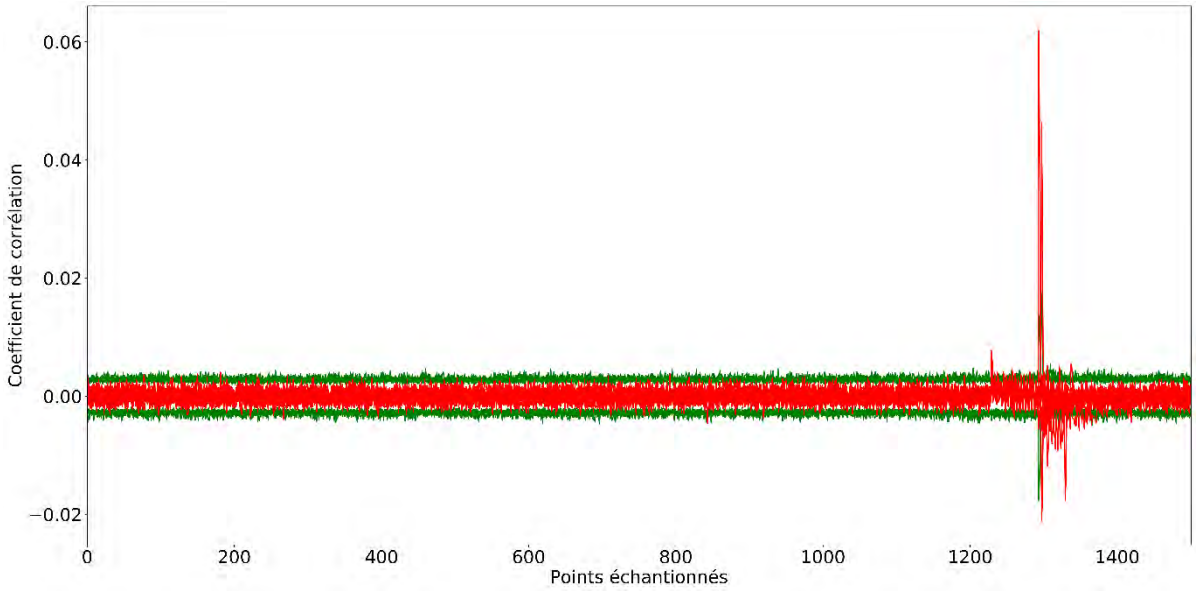


Figure 65 : Courbe CPA pour l'architecture Ben-16_SRL

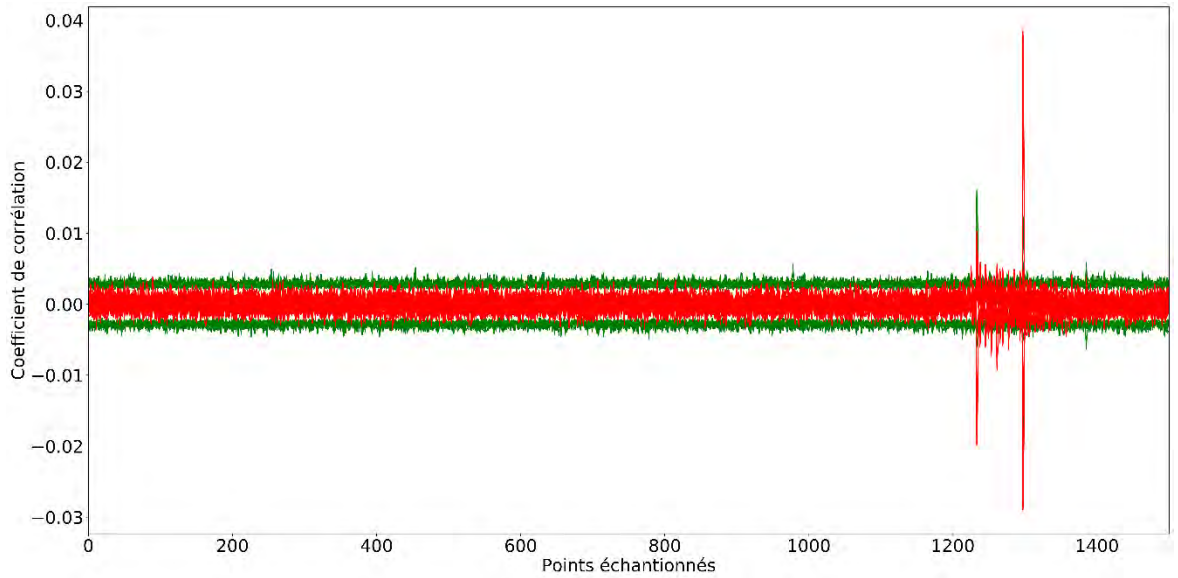


Figure 66 : Courbe CPA pour l'architecture Ben-16_Mem

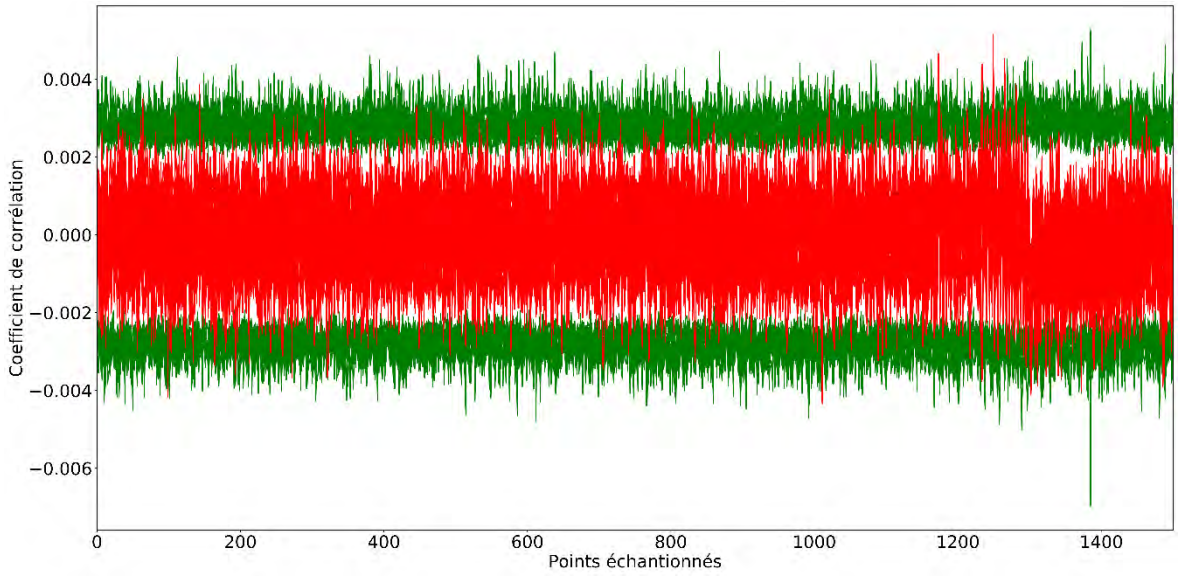


Figure 67 : Courbe CPA pour l'architecture Ben-16_MemBrass

La *Figure 68* illustre les résultats des attaques CPA sur les trois variantes Omeg-16 en indiquant le nombre d'octets révélés en fonction du nombre de traces utilisées dans l'attaque. Les résultats sont similaires à ceux obtenus avec les architectures Ben-16 dans le cas de l'architecture utilisant des SRL et de celle s'appuyant sur des mémoires distribuées sans aléa.

Dans la version avec un stockage basé sur des SRL (Omeg-16_SRL), 14 octets de la clé sont révélés avec un million de traces dont 50 % sont retrouvés avec environ 450 000 traces. Dans la variante pour laquelle le stockage est basé sur des mémoires distribuées sans aléa sur les adresses (Omeg-16_Mem), 11 octets de la clé sont révélés avec un million de traces dont 50 % sont retrouvés avec environ 550 000 traces. En revanche dans l'architecture avec aléa sur les adresses (Omeg-16_MemBrass) aucun octet de la clé n'est révélé avec un million de traces. Le coefficient de sécurité est alors de 1800 dans l'architecture Omeg-16_SRL et de 2200 dans l'architecture Omeg-16_MemBrass. Dans l'architecture Omeg-16_MemBrass il est supérieur à 4000.

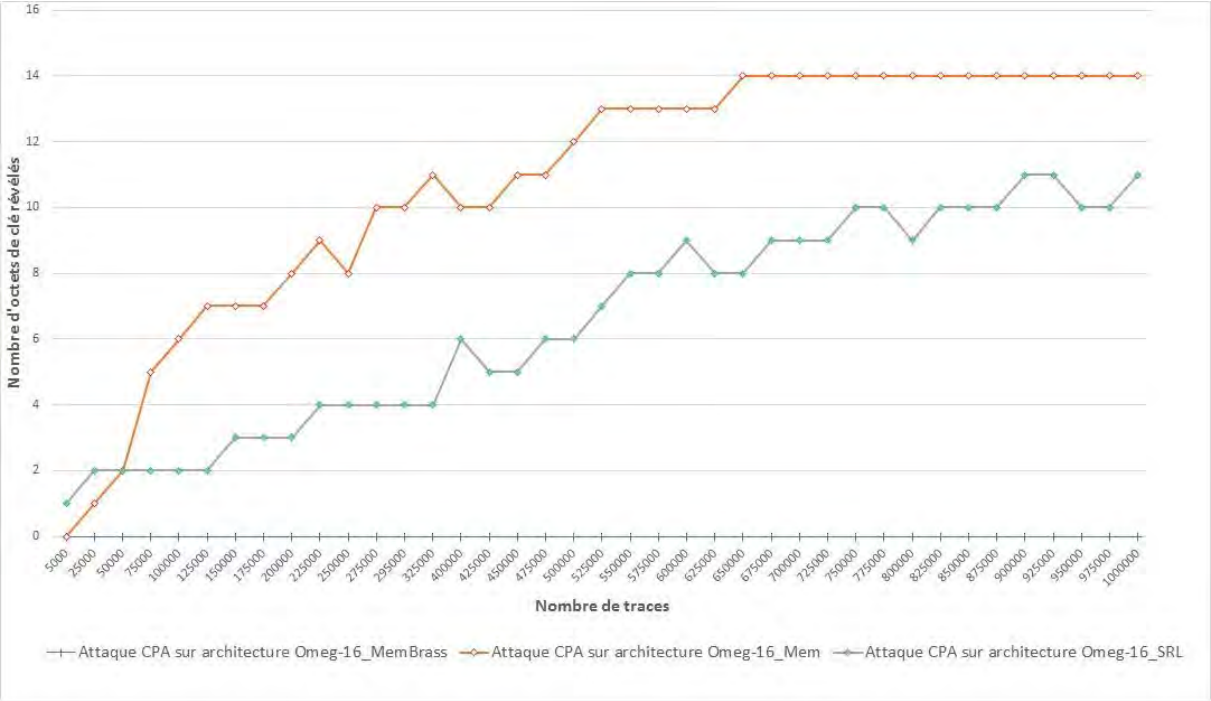


Figure 68 : Résultats de l'attaque CPA des trois variantes Omeg-16 en fonction du nombre d'octets révélés

Les résultats de l'attaque CPA sur les architectures Ben-16 et Omeg-16 présentent des résultats similaires, le réseau Omega offre donc assez de diversité pour la génération de permutation.

On peut également conclure de ces expériences que l'aléa dans le stockage présente un apport important en sécurité. Cependant, on constate que cet apport est moindre quand les SRLs sont utilisées. Ceci peut être expliqué par le mouvement des données dans les éléments mémoires à chaque cycle engendrant des transitions supplémentaires dépendantes des données manipulées.

Théoriquement, le coefficient de sécurité apporté par le brassage est quadratique par rapport aux nombres de points de mesures t dans lesquels la fuite d'information est dispersée [19]. Le coefficient de corrélation est alors réduit par un facteur t. Ce facteur peut être réduit à $\sqrt{t}$ en additionnant les contributions des t points de fuite. On réalise alors une CPA dite intégrée. Quand une trace est intégrée dans un intervalle de temps de temps donné, le signal ainsi que le bruit sont accumulés, ayant pour effet d'impacter le rapport signal sur bruit (SNR). Le SNR peut alors augmenter ou diminuer après intégration selon la taille de la fenêtre utilisée. Par conséquence, cette technique

nécessite une connaissance plus fine de l'implémentation afin de réduire l'impact dû à l'accumulation du bruit lors des mesures.

Dans la section suivante, nous présentons les résultats de l'attaque CPA avec intégration des points.

3. Attaques par CPA avec intégration

Nous avons réalisé une attaque CPA intégrée s'appuyant sur une fenêtre glissante de taille k introduite dans CLAVIER et coll. [87]. La taille de cette fenêtre doit englober l'ensemble des échantillons supposés participer à la fuite d'information.

Selon les résultats de nos précédentes attaques CPA, on remarque que les pics de corrélations s'étendent sur 200 points de mesure. Ces points se situent dans l'intervalle [1200 : 1400] dans les architectures Ben-16 et Omeg-16 :

L'intervalle [1200 : 1300] représente les fuites d'information durant la seconde partie de la 9^{ième} ronde.

L'intervalle [1300 : 1400], quant à lui, représente le calcul de la 10^{ième} ronde.

La fuite d'information du calcul des 16 octets se situent alors dans les deux intervalles. Durant nos expérimentations les trois intervalles dans la trace ont été sélectionnés [1200 :1400] [1200 : 1300] et [1300 : 1400].

Dans le premier intervalle [1200 : 1400] une fenêtre glissante de 200 points a été utilisée. Cependant l'attaque n'a pas apporté d'amélioration.

Dans les deux intervalles [1200 : 1300] et [1300 : 1400] une fenêtre de 100 points a été utilisée. Cette fois-ci, l'attaque a été plus efficace.

Il a été observé cependant que dans l'architecture sans aléas sur les adresses, les résultats optimaux sont obtenus dans l'intervalle [1300 : 1400]. En revanche, dans les architectures avec aléa sur le stockage, les résultats optimaux sont obtenus dans l'intervalle [1200 : 1300] avec l'architecture sans aléa sur le stockage.

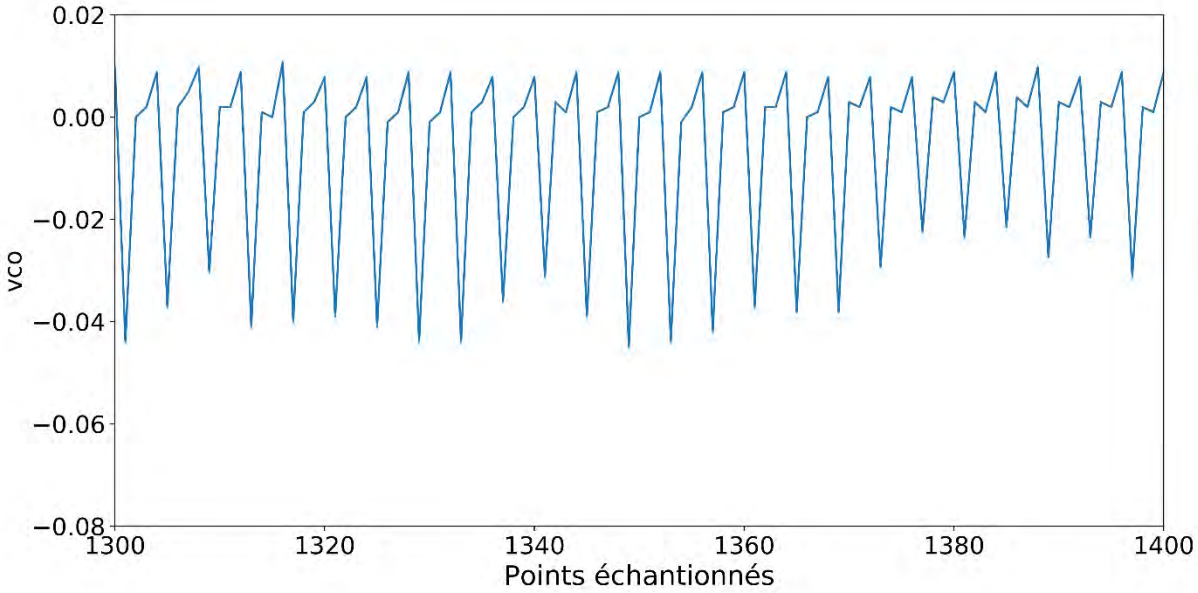


Figure 69: Traces de consommation de puissance entre l'intervalle [1300 : 1400] de l'architecture Ben-16_MemBrass

La *Figure 69* illustre la trace de consommation contenant les points de fuite dans l'intervalle [1300 : 1400] pour l'architecture Ben-16_MemBrass. Dans l'attaque CPA intégrée, un prétraitement de la trace, dans lequel une fenêtre glissante de taille $k = 100$, est appliqué sur la trace présentée en *Figure 69*. On obtient une nouvelle trace présentée en *Figure 70*. L'attaque CPA est alors réalisée sur cette trace.

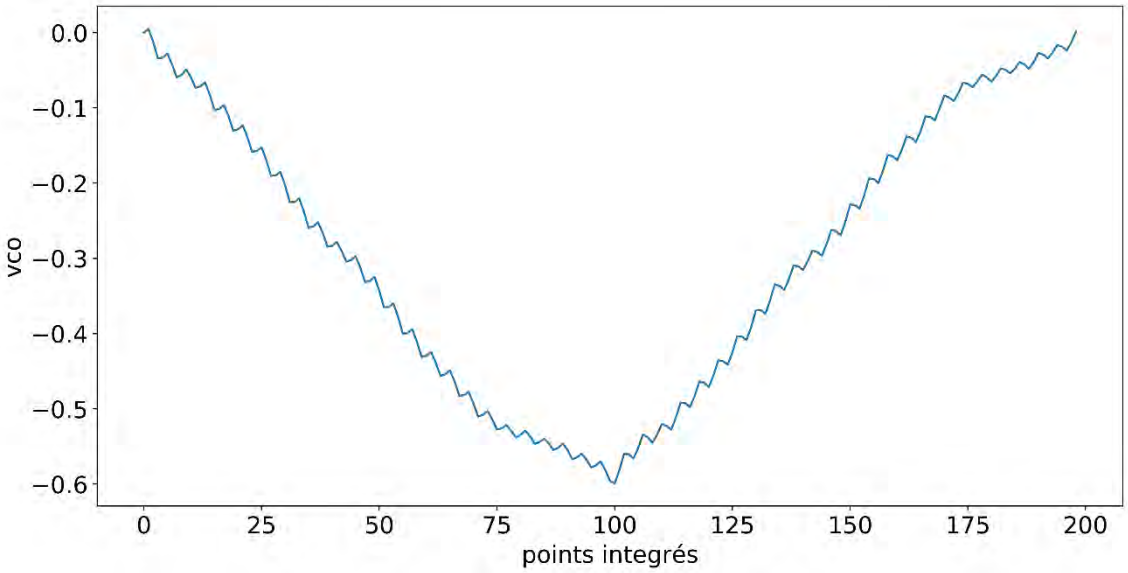


Figure 70 : Résultats de l'intégration

L'attaque CPA intégrée a été expérimentée sur les différentes variantes architecturales décrites précédemment. Les architectures avec aléa sur les adresses Ben-16_MemBrass/Omeg-16_MemBrass et Ben-16_SRL/Omeg-16_SRL ont été intégrées sur l'intervalle [1300 : 1400] et les architectures sans aléa dans le stockage Ben-16_Mem/Omeg-16_Mem ont été intégrée dans l'intervalle [1200 : 1300].

La *Figure 71* illustre les résultats des attaques CPA et CPA intégrée sur l'architecture Ben-16_SRL.

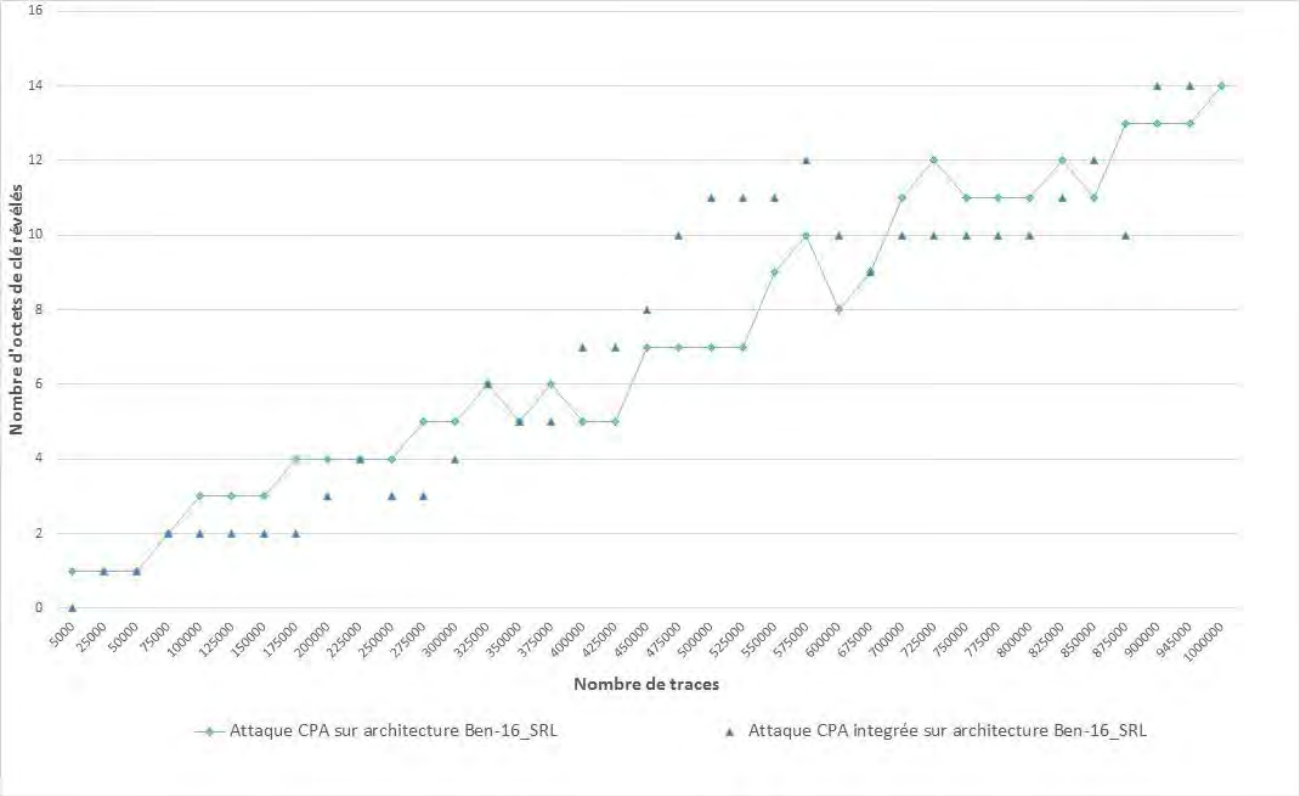


Figure 71: Résultat des attaques CPA et CPA intégrée sur les variantes Ben-16_SRL

Dans la version avec stockage basé sur des SRLs et en utilisant une attaque CPA, 14 octets de la clé ont été révélés avec un million de trace, dont 50% ont été retrouvés avec environ 550 000 traces. Tandis qu'on utilisant une attaque CPA intégrée cette architecture Ben-16_SRL, 13 octets de la clé ont étaient révélés avec un million de traces dont 50% sont révélés avec 450 000 traces.

La *Figure 72* illustre les résultats des attaques CPA et CPA intégrée sur les architectures Ben-16_Mem et Ben-16_MemBrass.

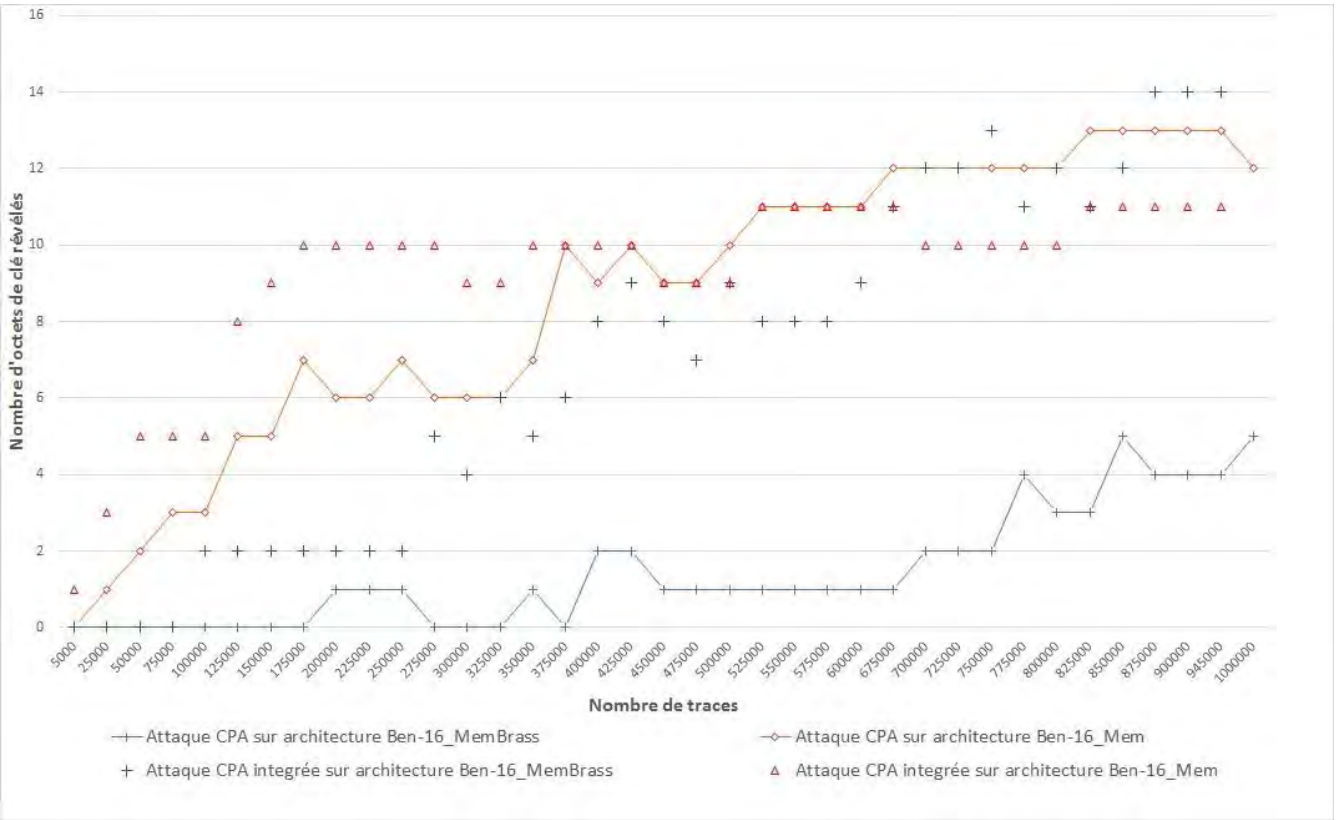


Figure 72 Résultats des attaques CPA et CPA intégrée sur les variantes Ben-16_Mem et Ben-16_MemBrass

Dans la version avec un stockage basé sur des mémoires distribuées sans aléa (Ben-16_Mem) en utilisant une attaque CPA, 12 octets de la clé ont été révélés avec un million de traces dont avec 50% des octets de la clé ont été révélés avec 400 000 traces. Toutefois, en utilisant une attaque CPA intégrée, 11 octets de la clé sont révélés dont 50% de la clé sont révélées avec environ 400 000 traces.

Dans la version avec un stockage basé sur des mémoires distribuées avec aléa (Ben-16_MemBrass) en utilisant une attaque CPA, 5 octets de la clé ont été révélés avec un million de traces dont 50% des octets de la clé ont été révélés avec 500 000 traces. Contrairement aux résultats des deux variantes précédentes, on constate un écart important de 6 octets en moyenne entre la courbe de l'attaque CPA et celle de l'attaque CPA intégrée.

La *Figure 73* illustre les résultats des attaques CPA et CPA intégrée sur l'architecture Omeg-16_SRL.

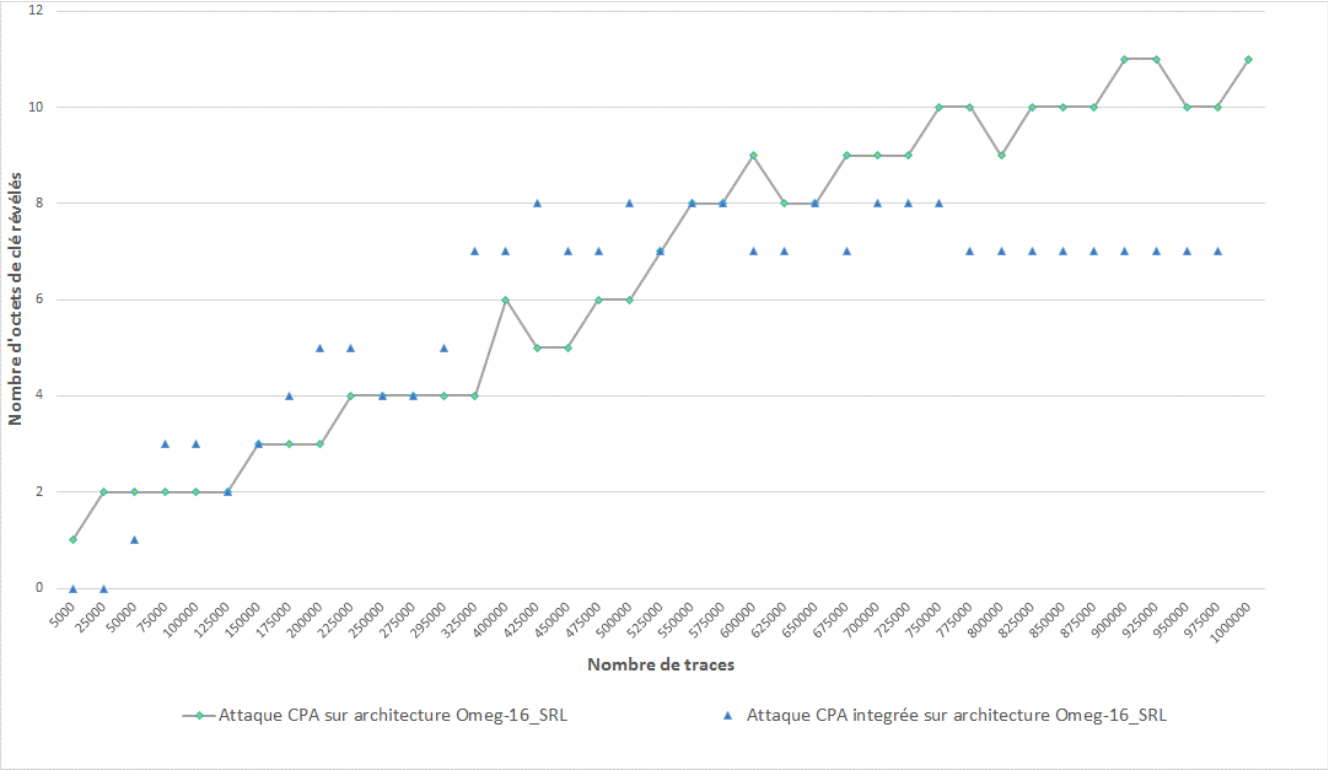


Figure 73: Résultat des attaques CPA et CPA intégrée sur les variantes Ben-16_SRL

Dans la version avec stockage basé sur des SRLs, en utilisant une attaque CPA,11 octets sont révélés avec un million de traces dont 50% des octets de la clé sont révélés avec environ 550 000 traces. En revanche, en utilisant l'attaque CPA intégrée, 7 octets de la clé sont révélés. Ici, l'intégration n'apporte aucune amélioration.

La *Figure 74* illustre les résultats des attaques CPA et CPA intégrée sur les architectures Omeg-16_Mem et Omeg-16_MemBrass.

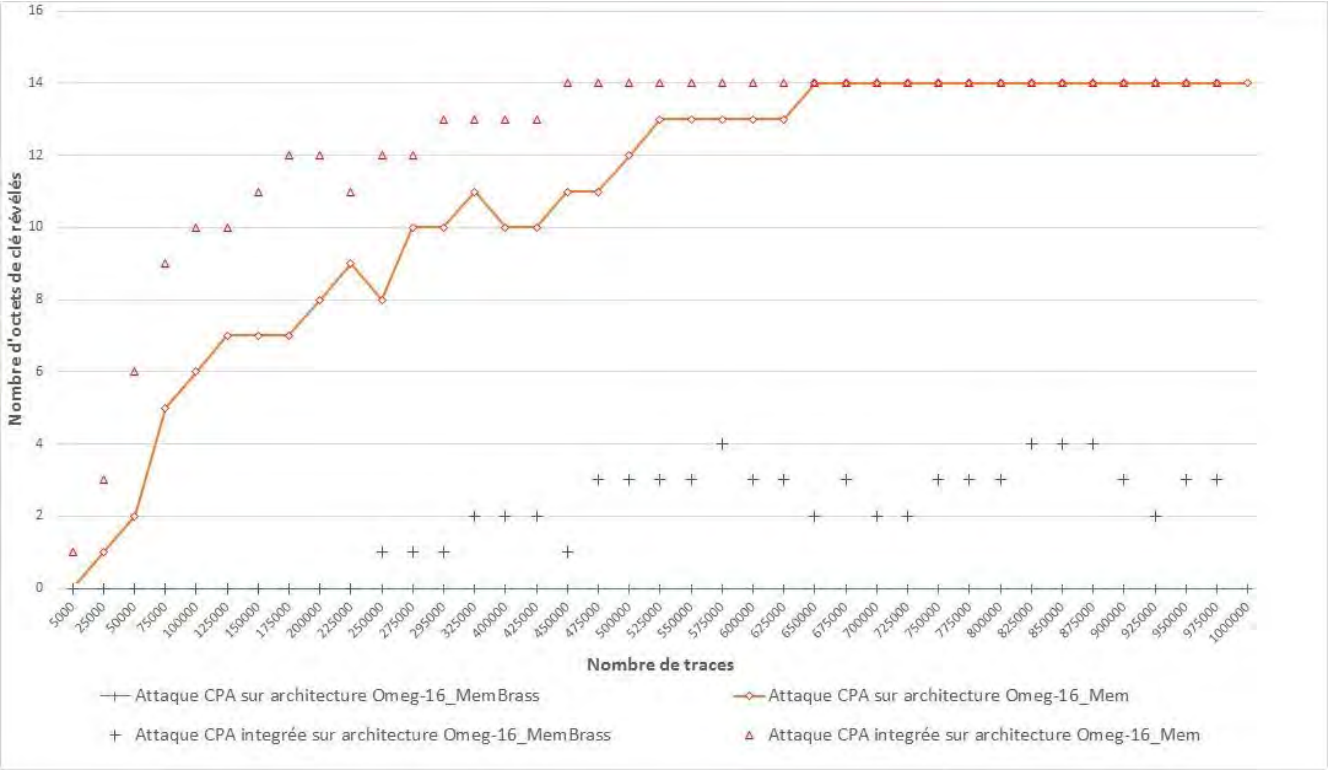


Figure 74 : Résultat des attaques CPA et CPA intégrées sur les variantes Omeg-16

Dans l'architecture Omeg-16_Mem, avec un million de traces, 14 octets sont révélés avec les deux attaques CPA et CPA intégrée. En revanche, 50 % des octets de la clé sont révélés avec 200 000 traces en considérant l'attaque CPA et avec 100 000 traces en considérant l'attaque CPA intégrée.

Dans l'architecture Omeg-16_MemBrass, aucun octet n'a été révélé avec un million de traces, en utilisant une attaque CPA et seulement 4 octets de la clé sont révélés avec une attaque CPA intégrée. Cette architecture est la plus robuste parmi les architectures étudiées jusqu'à maintenant.

On constate que l'intégration n'apporte pas d'amélioration nette dans les architectures Omeg-16. Cela pourrait être expliqué par le fait que les ressources réduites utilisées par ce réseau ont permis plus d'optimisation par l'outil, résultant en un signal utile plus faible. La faible puissance de ce dernier, couplé avec la forte diversité apportée par le réseau Omega 16x16 dont le nombre de permutation est 2^{32} permet d'apporter de la robustesse à l'architecture. En effet, l'intégration dégrade les résultats de l'attaque CPA dans le cas de

l'architecture ayant le meilleur rapport Débit/Surface, parmi les 6 architectures avec ajout d'aléa complet (c.à.d. l'architecture Omeg-16 utilisant des SRLs). On peut donc en conclure que le bruit est majoritaire par rapport au signal utile et que la diversité apportée par le réseau Omega est suffisante. Ce réseau représente donc une bonne alternative en vue de la réduction du surcoût en termes de surface et de performance (cf. chapitre 3). De plus, ces résultats montrent que la robustesse de l'architecture est fortement impactée par le choix architectural des éléments de stockage.

Durant les expérimentations présentées, les architectures ont été implémentées en utilisant les options de synthèse par défaut (optimisé en vitesse avec des efforts normaux sans la mise à plat de l'architecture). Par ailleurs, l'outil Xilinx ISE, permet d'offrir des optimisations importantes, selon les objectifs ciblés (optimisation en vitesse, optimisation en surface), résultant ainsi en des implémentations différentes selon les options choisies. Dans la section suivante, nous étudions donc l'influence des options de synthèse sur la robustesse des architectures Ben-16_MemBrass et Omeg-16_MemBrass face aux attaques CPA et CPA intégrée.

IV. Études de l'impact des options de synthèse

Afin d'évaluer l'influence des choix d'optimisation, les deux architectures Ben-16_MemBrass et Omeg-16_MemBrass ont été synthétisées avec 4 jeux d'options de synthèse (cf. *Tableau 16*) :

- Le premier est celui utilisé dans les sections précédentes de ce chapitre dans lesquelles les options de synthèse utilisée par l'outil Xilinx ISE durant la synthèse étaient celle par défaut, optimisé en vitesse avec un effort d'optimisation normal et en conservant la hiérarchie de l'architecture qu'on note *Opt_Vit_Hier*.
- Le second est similaire au précédent, mais avec la mise à plat de la hiérarchie de l'architecture afin de maximiser les optimisations (Keep hiearchy=no) qu'on note *Opt-Vit-Plat*.
- Le troisième, noté *Opt_Surf_Hier*, est optimisé en surface avec un effort d'optimisation élevé en conservant la hiérarchie de l'architecture (Keep hiearchy=yes). De plus, les mémoires distribués sont utilisées.
- Le quatrième est similaire au précédent, mais en mettant à plat la hiérarchie de l'architecture (Keep hiearchy=no), noté *Opt-Surf-Plat*.

Tableau 16 : détails des jeux d'options de synthèse utilisés pour les architectures Ben-16_MemBrass et Omeg-16_MemBrass dans cette étude

	Opt-Vit-Hier	**Opt-Vit-Plat**	**Opt-Surf-Hier**	**Opt-Surf-Plat**
Optimization goal	SPEED	SPEED	AREA	AREA
Optimization effort	normal	Normal	High	High
Keep hierarchy	yes	No	yes	no
RAM Style	Auto	Auto	Distributed	Distributed
ROM Style	Auto	Auto	Distributed	Distributed

Dans la sous-section suivante, nous présentons les résultats d'attaques CPA et CPA intégrée en utilisant les deux premiers jeux d'options de synthèse orientée vitesse : Opt-Vit-Hier et Opt-Vit-Plat.

1. Optimisation en vitesse

La *Figure 75* illustre les traces de consommation de puissance des architectures Ben-16 et la *Figure 76* illustre les traces de consommation de puissance des architectures Omeg-16 optimisées en vitesse.

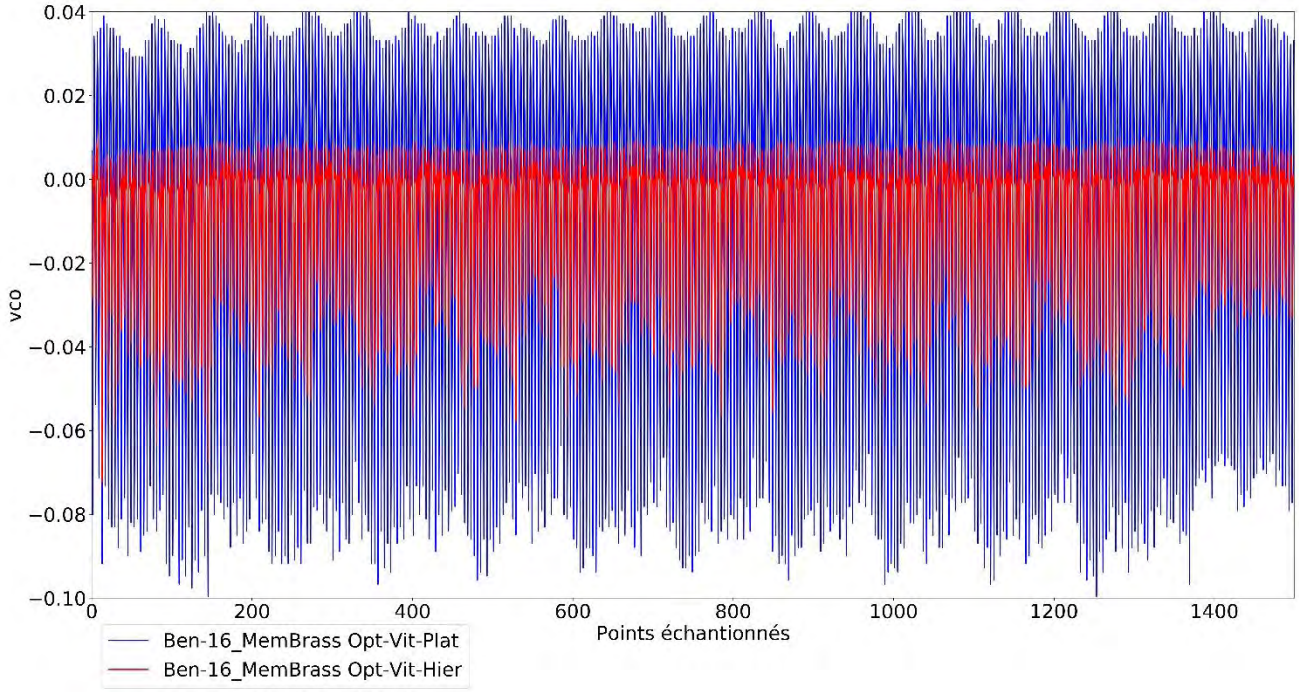


Figure 75 : Traces des architectures Ben-16_MemBrass pour les jeux d'options de synthèse Opt-Vit-Plat et Opt-Vit-Hier

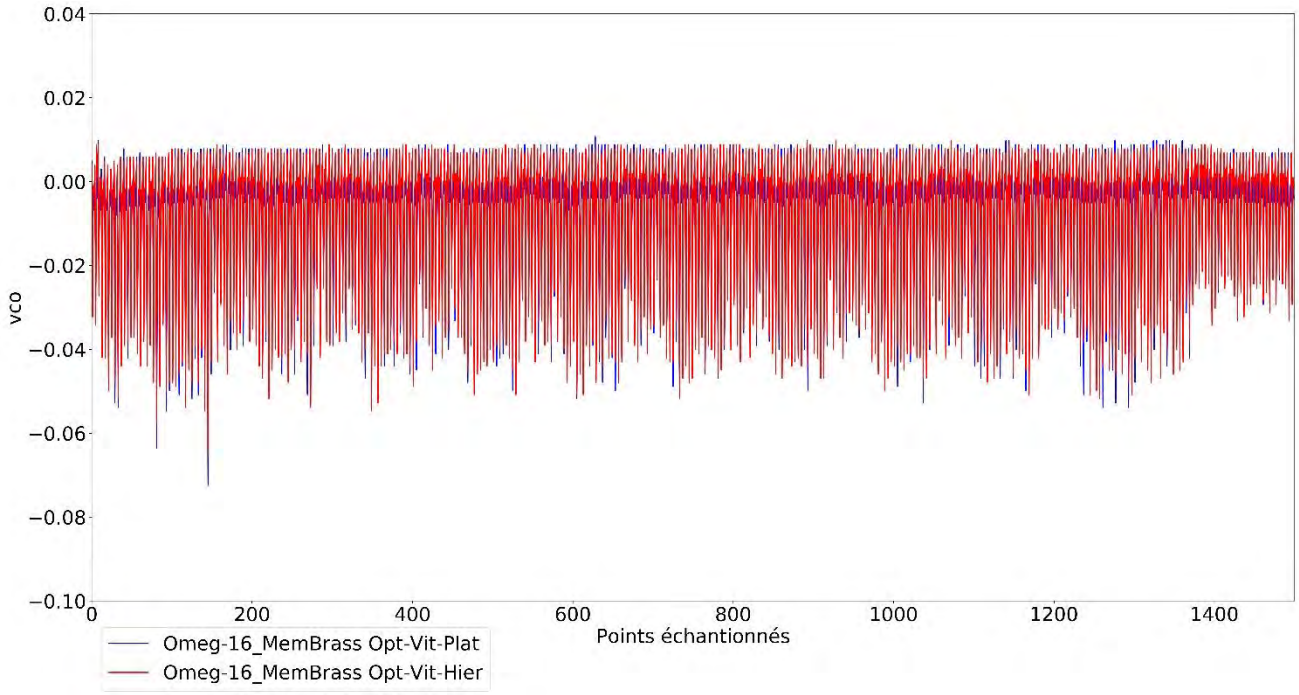


Figure 76 : Traces des architectures Omeg-16_MemBrass pour les jeux d'options de synthèse Opt-Vit-Plat et Opt-Vit-Hier

On remarque que dans le cas d'une architecture Ben-16, quand le jeu d'options de synthèse Opt-Vit-Hier est utilisé, l'implémentation consomme beaucoup moins que celle synthétisée

avec le jeu d'options de synthèse Opt-Vit-Plat. Tandis que dans le cas d'une architecture Omeg-16, la différence de consommation est moindre.

La *Figure 77* illustre les résultats CPA et CPA intégrée de l'architecture Ben-16_MemBrass en utilisant les deux jeux d'options de synthèse Opt-Vit-Hier et Opt-Vit-Plat.

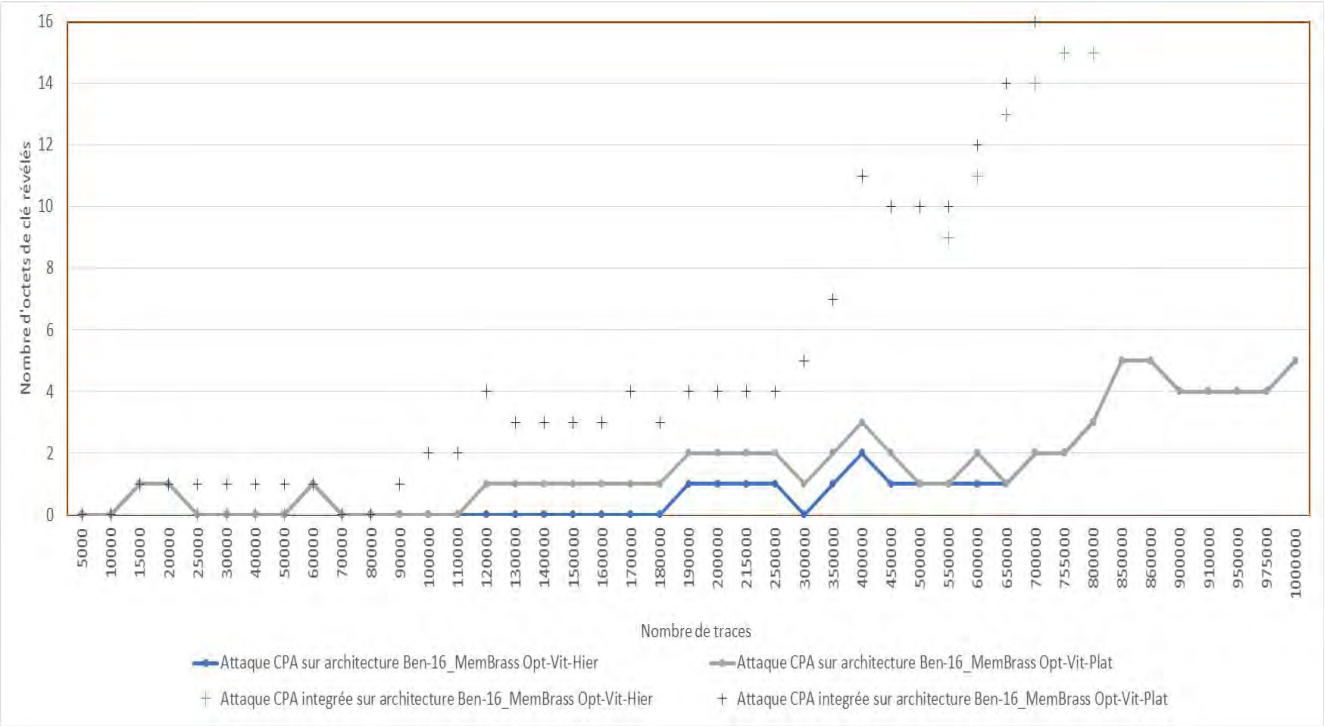


Figure 77 : Résultat des attaques CPA et CPA intégrées sur l'architecture Ben-16 synthétisée avec les jeux d'options de synthèse Opt-Vit-Hier et Opt-Vit-Plat

En utilisant une attaque CPA, dans le cas où la hiérarchie de l'architecture est conservée (Opt-Vit-Hier), 5 octets sont révélés avec un million de traces. En revanche, dans le cas où la hiérarchie de l'architecture est mise à plat (Opt-Vit-Plat), un seul octet est révélé avec un million de traces. Dans une attaque CPA intégrée, dans le cas où la hiérarchie de l'architecture est conservée (Opt-Vit-Hier), 15 octets de la clé sont révélés avec un million de traces dont 50% des octets de la clé sont révélés avec 450 000 traces. En revanche, dans le cas où l'architecture est mise à plat (Opt-Vit-Plat), seulement 5 octets sont révélés. Les résultats ont montré que la mise à plat de la hiérarchie offre donc plus de robustesse dans le cas où l'architecture Ben-16_MemBrass est optimisée en vitesse.

La *Figure 78* illustre les résultats des attaques CPA et CPA intégrée pour l'architecture Omeg-16_MemBrass en utilisant les deux jeux d'options de synthèses Opt-Vit-Hier et Opt-Vit-Plat.

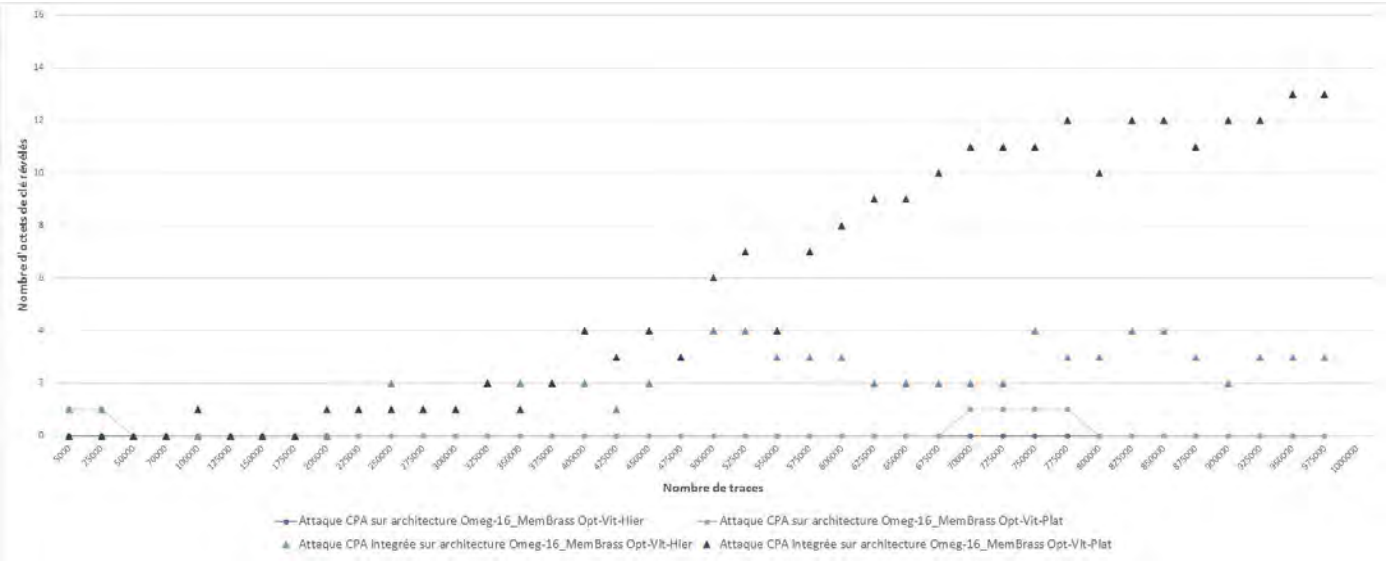


Figure 78 : Résultat des attaques CPA et CPA intégrées sur l'architecture Omeg-16 synthétisé avec les options de synthèse Opt-Vit avec efforts réduits et efforts maximums

En utilisant une attaque CPA, aucun octet de la clé n'est révélé avec un million de traces quand la hiérarchie de l'architecture est conservée (Opt-Vit-Hier). Quand la hiérarchie de l'architecture est mise à plat, un octet est révélé avec un million de traces. En utilisant une attaque CPA intégrée, 4 octets de la clé sont révélés quand la hiérarchie de l'architecture est conservée (Opt-Vit-Hier) avec un million de traces. Quand la hiérarchie de l'architecture est mise à plat (Opt-Vit-Plat), 12 octets sont révélés avec un million de traces dont 50% de la clé est révélée avec 600 000 traces en utilisant une attaque CPA intégrée. Contrairement au résultat précédent concernant l'architecture Ben-16_MemBrass, l'architecture Omeg-16_MemBrass synthétisée avec le jeu d'options de synthèse Opt-Vit-Hier est plus robuste.

Dans la partie suivante, nous allons étudier l'impact des deux jeux d'options de synthèse optimisé en surface, Opt-Surf-Hier et Opt-Surf-Plat L'ensemble des résultats est présenté dans le *Tableau 17* à venir.

2. Optimisation en surface

La *Figure 79* illustre les traces de consommation de puissance des architectures Ben-16 et la *Figure 80* illustre les traces de consommation de puissance des architectures Omeg-16 optimisées en surface.

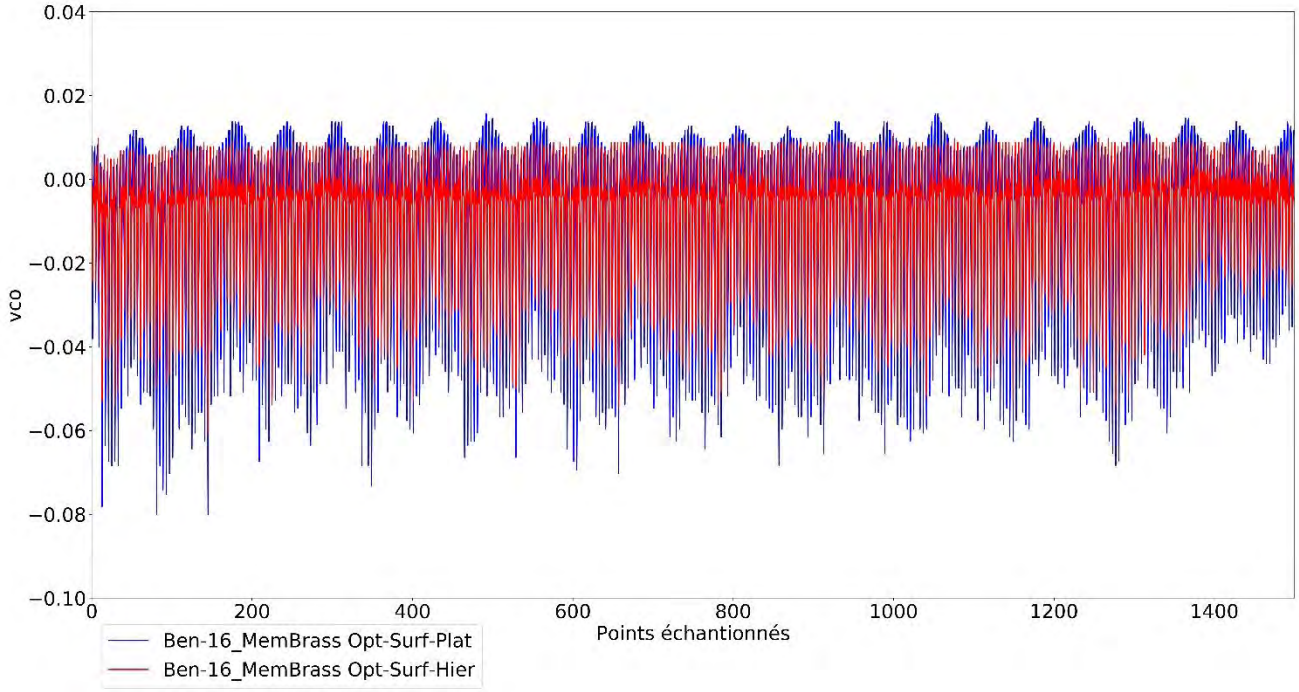


Figure 79 : Traces des architectures Ben-16_MemBrass pour les jeux d'options de synthèse Opt-Surf-Plat et Opt-Surf-Hier

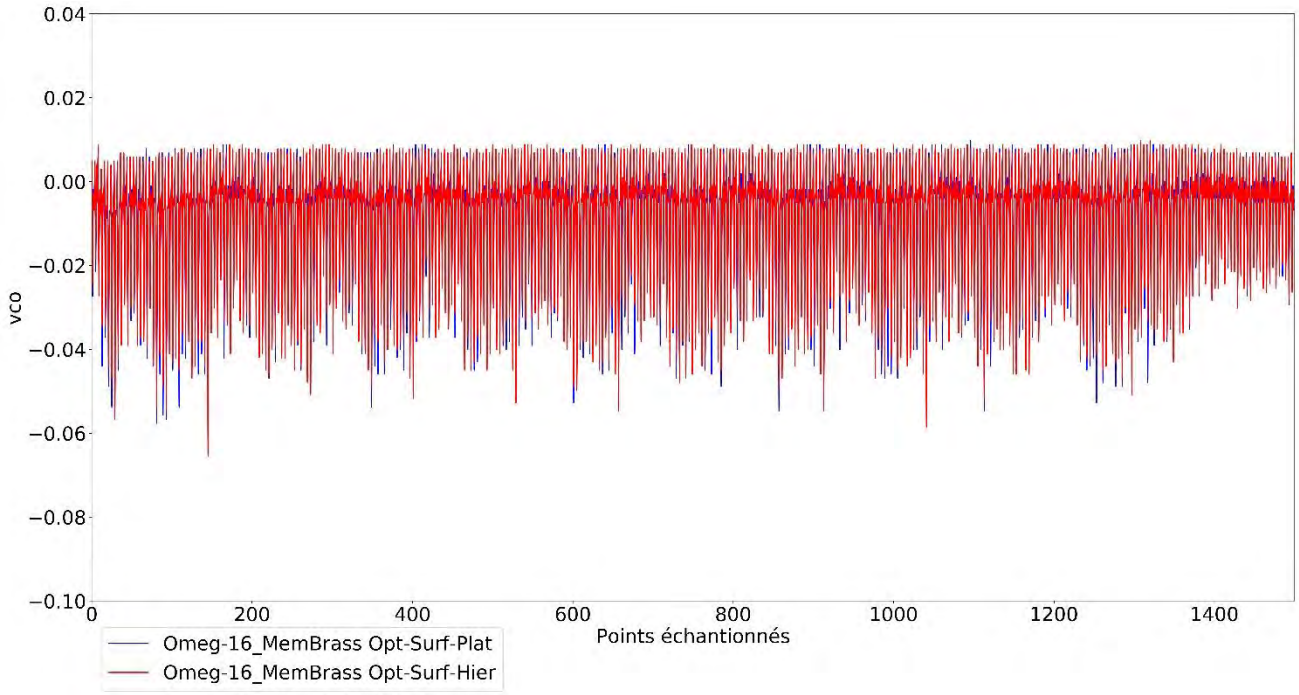


Figure 80 : Traces des architectures Omeg-16_MemBrass pour les jeux d'options de synthèse Opt-Surf-Plat et Opt-Surf-Hier

Contrairement aux variations observées avec les jeux d'options de synthèse Opt-Vit-Plat et Opt-Vit-Hier, on observe peu de différence de consommation pour l'architecture Ben-16_MemBrass avec les jeux d'options Opt-Surf-Plat et Opt-Surf-Hier.

La *Figure 81* illustre les résultats des attaques CPA et CPA intégrée pour l'architecture Ben-16_MemBrass. La *Figure 82* illustre ces mêmes résultats pour l'architecture Omeg-16_MemBrass. Dans ces deux figures, les deux jeux d'options de synthèse, optimisées en surface, Opt-Surf-Hier et Opt-Surf-Plat sont considérés.

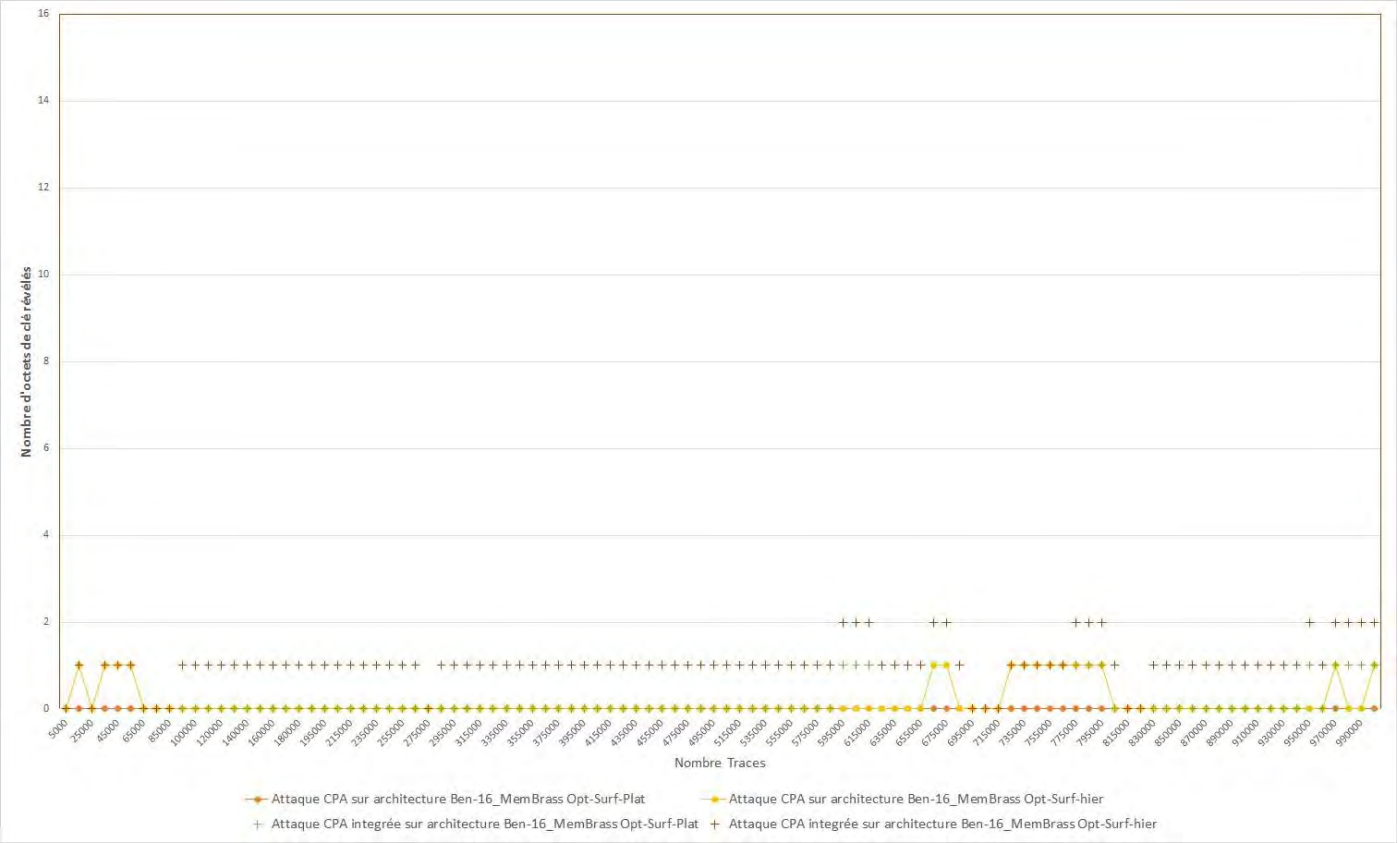


Figure 81 : Résultats des attaques CPA et CPA intégrée sur l'architecture Ben-16_MemBrass synthétisée avec les jeux d'options de synthèse Opt-Surf-Hier et Opt-Surf-Plat

En considérant une attaque CPA, dans le cas où la hiérarchie est conservée (Opt-Surf-Hier), un octet de la clé est révélé avec un million de traces. Dans le cas où la hiérarchie est mise à plat (Opt-Surf-Plat), aucun octet n'est révélé avec un million de traces. Quand la hiérarchie de l'architecture est conservée (Opt-Surf-Hier), l'attaque CPA intégrée n'a aucun apport : un seul octet est révélé avec un million de traces. De même quand la hiérarchie est mise à plat (Opt-Surf-Plat), l'attaque CPA intégrée : un seul octet de la clé est révélé.

La *Figure 82* illustre les résultats des attaques CPA et CPA intégrée de l'architecture Omeg-16_MemBrass en utilisant les deux jeux d'options de synthèse, optimisés en surface, (Opt-Surf-Hier) et (Opt-Surf-Plat).

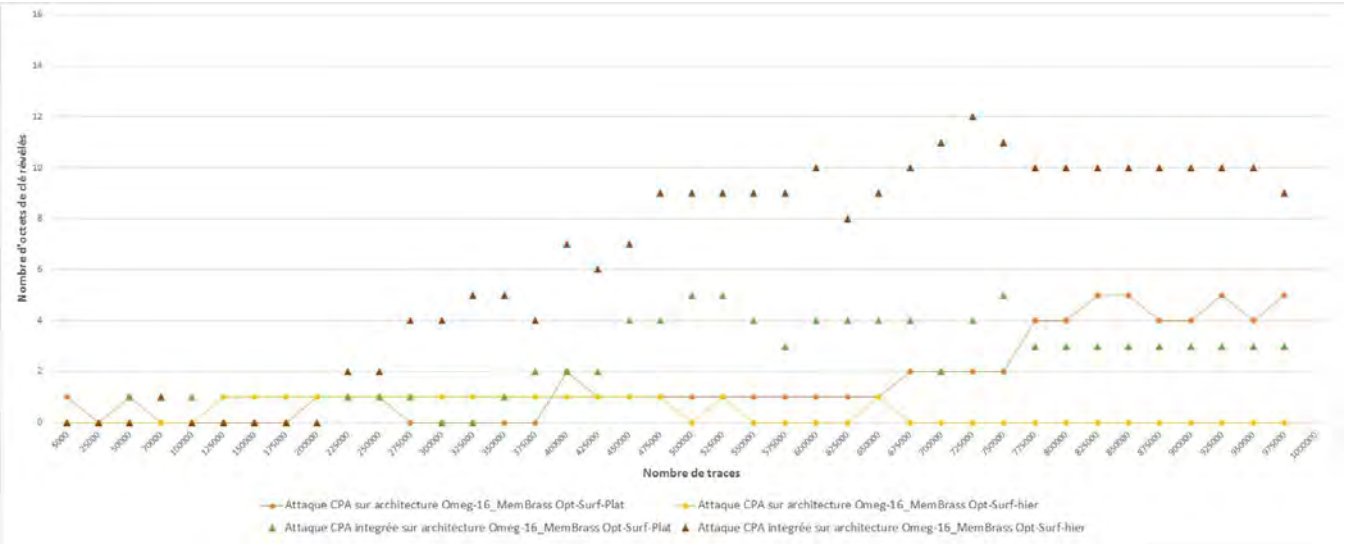


Figure 82 : Résultats des attaques CPA et CPA intégrée sur l'architecture Omeg-16_MemBrass synthétisée avec les jeux d'options de synthèse Opt-Surf-Hier et Opt-Surf-Plat

En considérant une attaque CPA, dans le cas où la hiérarchie est conservée (Opt-Surf-Hier), un seul octet de la clé est révélé avec un million de traces. Tandis dans le cas où la hiérarchie est mise à plat (Opt-Surf-Plat), 5 octets sont révélés avec un million de traces. En revanche, en considérant une attaque CPA intégrée, dans le cas où la hiérarchie est conservée (Opt-Surf-Hier), 10 octets sont révélés avec un million de traces ; 50% de la clé est révélée avec 500 000 traces. Dans le cas où la hiérarchie de l'architecture est mise à plat (Opt-Surf-Plat), 5 octets de la clé sont révélés avec un million de traces. L'architecture Omeg-16_MemBrass synthétisée avec le jeu d'options Opt-Surf-Plat est donc la plus robuste.

3. Bilan des résultats d'attaques

Le Tableau 17 illustre le bilan de l'étude de l'impact des options de synthèse. On constate que le jeu d'option de synthèse a un impact très fort sur les résultats. Dans le cas de l'architecture Ben-16_MemBrass, l'implémentation est sécurisée contre les attaques CPA et CPA intégrées. En revanche, la quasi-totalité de la clé (15 octets sur les 16) est obtenue quand le jeu d'option de synthèse Opt-Vit-Hier est utilisé. De plus, la clé peut être retrouvée avec 450 000 traces en utilisant l'attaque en *brut force*. La consommation de puissance la plus importante est obtenue quand le jeu de synthèse Opt-Vit-Plat est utilisé. En effet, ce jeu de synthèse permet d'optimiser les performances au maximum.

Dans le cas de l'architecture Omeg-16_MemBrass, les implémentations synthétisées avec les 4 jeux d'options de synthèse ont une consommation de puissance similaire. Pour cette

architecture, l'implémentation est protégée contre les attaques CPA et CPA intégrée quand les deux jeux d'options de synthèse Opt-Vit-Hier et Opt-Surf-Plat sont utilisés. En revanche, une attaque CPA intégrée permet de révéler la clé en utilisant l'attaque en brut force avec 500 000 traces quand l'architecture est synthétisée avec le jeu d'option de synthèse Opt-Vit-Plat et 600 000 traces quand l'architecture est synthétisée avec le jeu d'option de synthèse Opt-Surf-Hier.

Tableau 17 : Bilan des résultats d'attaques CPA et CPA intégrée sur les architectures Ben-16_MemBrass et Omeg-16_MemBrass synthétisées avec les quatre jeux d'options de synthèse Opt-Vit-Hier, Opt-Vit-Plat, Opt-Surf-Hier, Opt-Surf-Plat

Architecture	Ben-16_MemBrass				Omeg-16_MemBrass			
Options de synthèse	Opt-Vit-Hier	Opt-Vit-Plat	Opt-Surf-Hier	Opt-Surf-Plat	Opt-Vit-Hier	Opt-Vit-Plat	Opt-Surf-Hier	Opt-Surf-Plat
Nombre d'octets révélés (CPA)	6	1	1	0	0	1	1	5
Nombre d'octets révélés (CPA intégrée)	15	5	1	1	4	12	10	4
Nombre de traces pour révéler 50 % de la clé	450 000	>1 000 000	>1 000 000	>1 000 000	>1 000 000	600 000	500 000	>1 000000

On constate que la consommation globale des implémentations n'a pas d'impact direct sur le résultat des attaques. Le coefficient de corrélation est impacté par le rapport signal sur bruit. Le résultat de l'attaque est alors lié au nombre de points de fuite et à la puissance du signal utile ainsi qu'au nombre de points de bruit accumulés par l'attaque CPA. Le signal utile est lié à l'optimisation faite par l'outil Xilinx ISE. Les résultats ne nous ont pas permis de déterminer un lien direct entre les jeux d'options de synthèse utilisées et les résultats d'attaque. Néanmoins, on constate que globalement, les jeux d'option de synthèse optimisée en surface Opt-Surf-Plat apportent de la robustesse à l'architecture. Ce jeu d'option de synthèse permet d'apporter un maximum d'optimisation en surface, résultant ainsi on un signal faible.

L'architecture la plus robuste est alors l'architecture Ben-16_MemBrass optimisée en surface. En raison de l'aléa important apporté par le réseau de Benes (couplé avec une faible consommation de puissance), de l'ajout d'aléa dans le stockage et des optimisations effectuées par l'outil Xilinx.

Dans la section suivante, nous évaluons le surcoût en termes de surface et performance des architectures Ben-16_MemBrass et Omeg-16_MemBrass en utilisant ces mêmes quatre options de synthèse.

V. Comparaison des résultats en surface performance et sécurité

Le *Tableau 18* illustre les résultats en surface et en performances des architectures Ben-16 et Omeg-16 en utilisant les quatre jeux d'options de synthèse. La version Ben-16 la plus robuste est celle optimisée en surface en mettant à plat la hiérarchie de l'architecture (Opt-Surf-Plat) et celle ayant le meilleur rapport débit/surface. Cette architecture présente un surcoût de 72% en termes de surface et une chute de débit de 2,8 par rapport à l'architecture de référence optimisé en surface. La version Omeg-16 la plus robuste quant à elle est aussi celle ayant le meilleur rapport débit/surface. Cette architecture présente un surcoût de 40% et une chute de débit de 2,16 par rapport à l'architecture de référence.

Tableau 18: Comparaison des résultats de surface et performances sur les architectures Ben-16_MemBrass et Omeg-16_MemBrass avec les différentes options de synthèse

Architecture	***Ben-16_MemBrass***				***Omeg-16_MemBrass***			
Options de synthèse	**Opt-Vit-Hier**	**Opt-Vit-Plat**	**Opt-Surf-Hier**	**Opt-Surf-Plat**	**Opt-Vit-Hier**	**Opt-Vit-Plat**	**Opt-Surf-Hier**	**Opt-Surf-Plat**
Fréquence (Mhz)	104,58	110,05	94,87	98,04	102,82	104,31	99,9	110,91
Latence	336	336	336	336	336	336	336	336
Débit (Mbit/s)	39,84	41,92	36,1	37,35	39,17	39,74	38,05	42,25
Slice	296	140	294	153	198	114	182	114
Slice Register	67	67	67	50	67	69	67	50
Slice LUT	455	364	445	365	360	292	355	293
BRAM	0	0	0	0	0	0	0	0
DSP	0	0	0	0	0	0	0	0
Débit/Surface	0,13	**0,29**	0,12	0,24	0,19	0,34	0,29	**0,37**

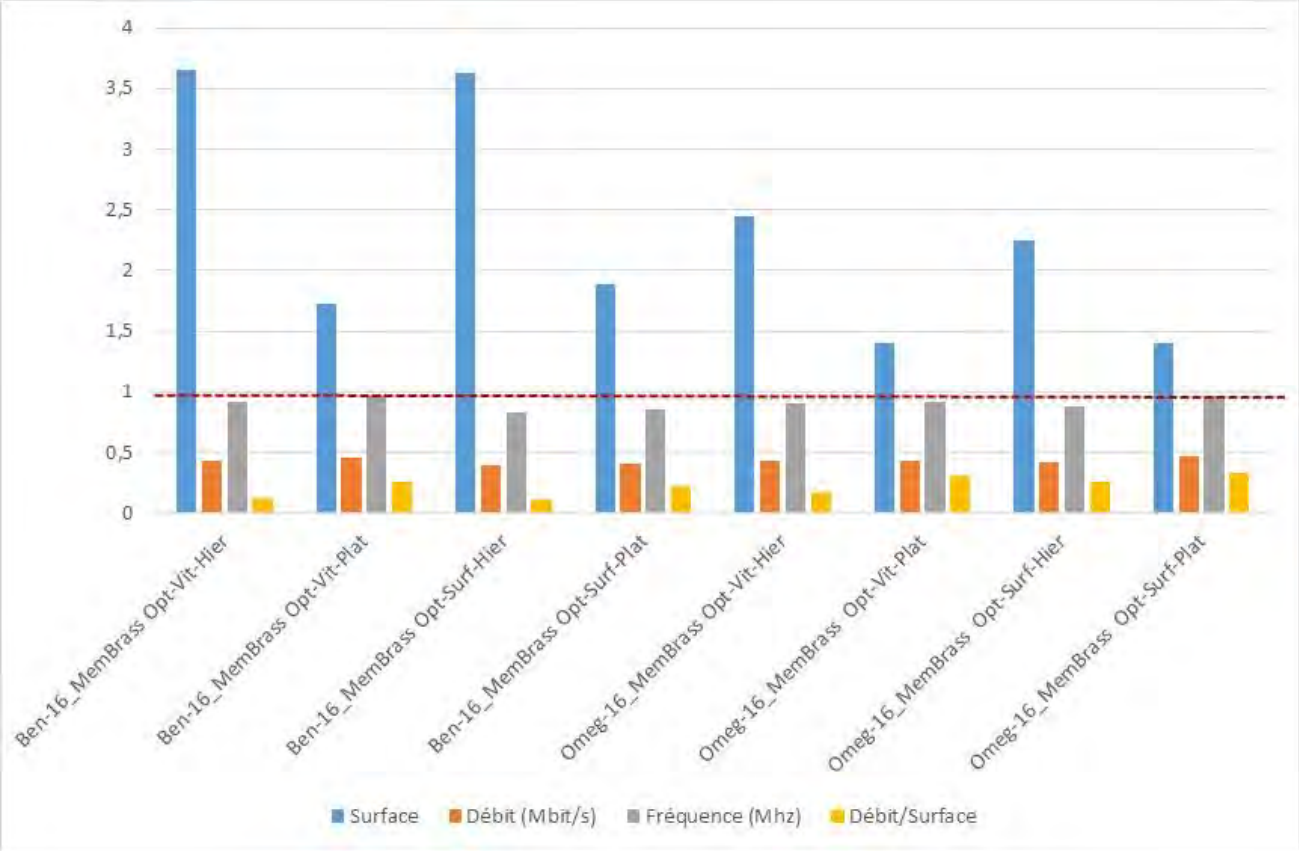


Figure 83 : Coût des architectures MemBrass normalisé par rapport à [26] en termes de surface optimisées avec les jeux d'options de synthèse Opt-Vit-Hier et Opt-Vit-Plat

La *Figure 83* présente les caractéristiques des architectures Ben-16_MemBrass et Omeg-16_MemBrass, en considérant les 4 jeux d'options de synthèse, normalisées par rapport à l'architecture de référence [26]. Les architectures Ben-16 optimisées en vitesse ont un surcoût en surface de 3,6 et 1,7 respectivement sans et avec mise à plat de la hiérarchie de l'architecture. Dans le cas où les architectures sont optimisées en surface, le surcoût est de 3,6 et 1,8 respectivement sans et avec mise à plat de la hiérarchie de l'architecture.

Les architectures Omeg-16 optimisées en vitesse ont un surcoût de 2,4 et 1,4 respectivement sans et avec mise à plat de la hiérarchie de l'architecture. Dans le cas où les architectures sont optimisées en surface, le surcoût est de 2,2 et de 1,4 respectivement sans et avec mise à plat de la hiérarchie de l'architecture.

Les résultats en performances sont globalement similaires. Les architectures proposées atteignent 40% du débit et 90% de la fréquence de l'architecture de référence. L'architecture la plus efficace au regard du rapport débit/surface est l'architecture Omeg-16_MemBrass (0,33). Enfin, l'architecture la plus robuste, l'architecture Ben-16_MemBrass, possède un rapport débit/surface de 0,25.

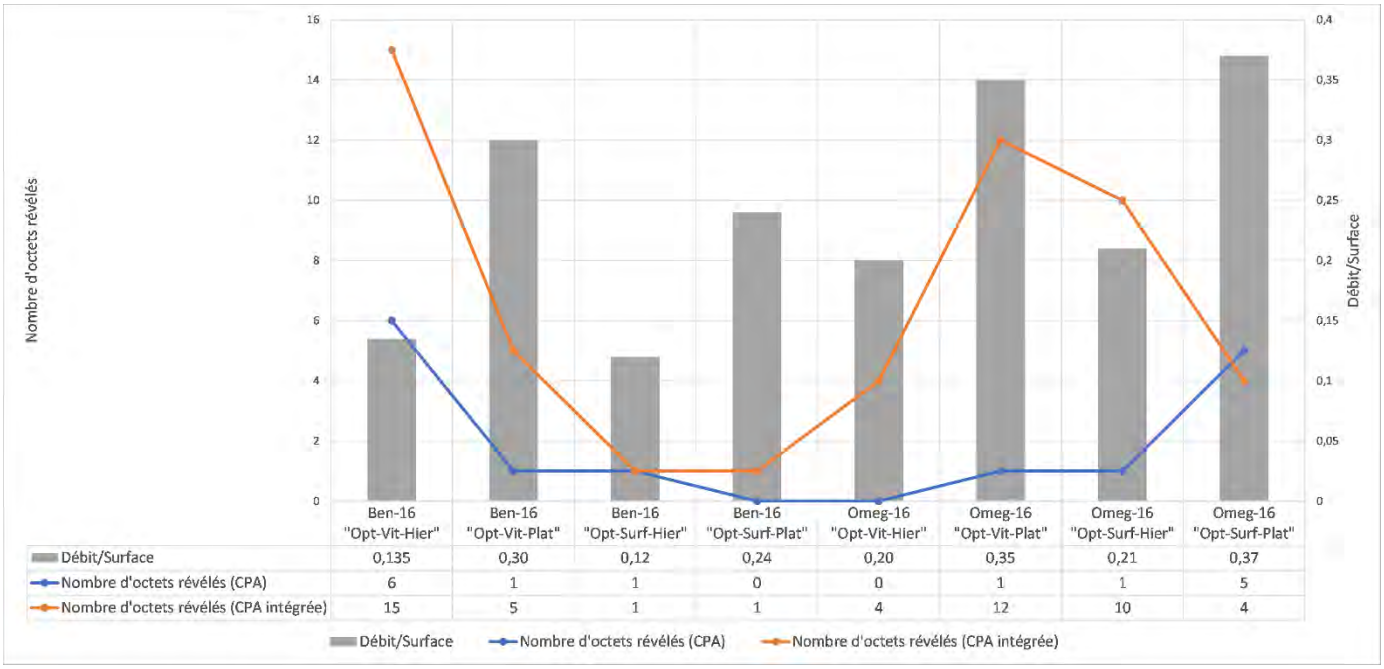


Figure 84 : Évaluation de l'efficacité (rapport Débit/Surface) et de la robustesse contre les attaques par observation (Nombre d'octets révélés)

La

Figure 84 représente une comparaison entre l'efficacité et l'apport en sécurité (nombre d'octets révélés avec les attaques CPA et CPA intégrée). Les architectures sécurisées contre les attaques CPA et CPA intégrée avec un million de traces sont les Ben-16(Opt-Vit-Plat), Ben-16(Opt-Surf-Hier), Ben-16(Opt-Surf-Plat), Omeg-16(Opt-Vit-Hier) et Omeg-16(Opt-Vit-Plat). Le meilleur compromis entre l'efficacité et la robustesse est obtenu avec l'architecture Omeg-16 optimisée en surface avec la hiérarchie mise à plat (Opt-Surf-Plat).

VI. Conclusion

L'architecture la plus robuste est la Ben-16_MemBrass optimisée en surface Ben-16 (Opt-Surf-Plat). On remarque que, comme l'on pouvait s'y attendre, les options de synthèses ont un impact important sur les résultats pour les architectures visées.

De manière global, les architectures optimisées en surface avec la mise à plat de la hiérarchie de l'architecture sont les plus robustes. En effet, une surface minimisée mène à un signal

affaibli et avec un rapport signal sur bruit plus faible. Ainsi l'attaque CPA intégrée n'apporte aucune contribution.

Dans le chapitre suivant nous présentons une architecture dans laquelle l'ajout d'aléa est étendu à plusieurs blocs. Son évaluation en sécurité et ses caractéristiques en termes de surface et de performances sont également présentés.

Chapitre 5 Architecture AES multi-blocs

I. Description de notre modèle d'architecture

Selon les conclusions du chapitre précédant, le meilleur rapport coût / sécurité est atteint pour un AES conçu avec un module de génération d'aléa basé sur un réseau de Benes (avec optimisation en surface).

Dans ce chapitre, nous présentons notre dernière contribution dans laquelle l'aléa est étendu sur une architecture multi-blocs. Afin de générer les permutations, nous allons donc de nouveau utiliser un réseau de Benes. Nous nous laissons toutefois la possibilité de tester d'autres architectures à l'avenir, comme le réseau Omega.

Dans le mode standard de l'algorithme *AES-Electronic CodeBook (ECB)*, chaque bloc est chiffré de la même manière. On obtient alors systématiquement le même texte chiffré pour un texte clair et une clé donnée. En considérant une image, dans laquelle chaque pixel serait chiffré avec la même approche, les contours de l'image chiffrée resteraient distinguables étant donné que les pixels d'une même couleur en *clair* seraient affectés une couleur certes différente du *clair*, mais toujours identique. Les autres modes de chiffrement quant à eux permettent de produire des chiffrés uniques même avec un même *clair* de départ.

Bien que le mode de chiffrement ECB ne soit pas utilisé, les mêmes contre-mesures peuvent s'appliquer pour tous les modes de chiffrement. Parmi ces autres modes de chiffrement, le mode compteur est l'un des plus connus, car il combine plusieurs avantages :

- Il permet un pré-calcul du chiffrement car celui-ci, dans le mode compteur, ne dépend pas que du texte clair.
- Il n'y a pas de dépendance entre les blocs ce qui permet de paralléliser certaines opération, et de réaliser un traitement avec accès aléatoire pour un chiffrement plus rapide. Toutefois, pour un message clair donné, il n'est pas possible de chiffrer un bloc C_i sans avoir chiffré les $i-1$ blocs précédents de ce bloc.
- Enfin, ce mode permet une parallélisation des calculs des blocs.

Toutefois, il faut garder en tête que l'objectif de nos travaux est de maximiser la robustesse des architectures contre les attaques par observation, par l'ajout d'aléa dans le traitement des blocs. Notre approche permet ainsi d'obtenir une probabilité de $\frac{1}{16B}$ (avec B le nombre de blocs) qu'un point de fuite d'un octet donné se retrouve dans au même point de la trace

observée, en considérant une permutation uniforme $16B!$. Cependant, ceci nécessite un réseau de Benes $2^{N+4} \times 2^{N+4}$ avec $N = \lg(B)$. Nous avons donc étudié deux approches :

- *Génération complète des permutations* : cette solution nécessite un réseau de permutation $2^{N+4} \times 2^{N+4}$. Un réseau de cette taille implique un surcoût non négligeable en surface, comme nous le verrons par la suite.
- *Génération limitée des permutations* : utilisation d'un réseau 16×16 afin de permuter l'ordre de traitement des blocs et l'ordre de traitement des octets de chaque bloc en $2 \times B$ cycles.

Pour nos deux architectures multi-blocs, l'interface comporte un PRNG (Pseudo-Random Number Generator) et un compteur de *32* bits. Le vecteur d'initialisation est défini selon les normes spécifiées dans [96]. L'algorithme AES chiffre des vecteurs de 128 bits composés d'un vecteur d'initialisation de 96 bits, concaténé avec le compteur de 32 bits. Le texte chiffré est le résultat d'une opération *XOR* entre le vecteur chiffré et le texte clair.

Dans ce chapitre, nous présentons une implémentation 8 bits d'un algorithme AES générique en mode compteur, avec un nombre de bloc variable. Cette architecture est conçue de façon générique avec aléa sur l'ordre de traitement des octets dans les blocs. La *Figure 85* rappelle le schéma de principe de l'architecture.

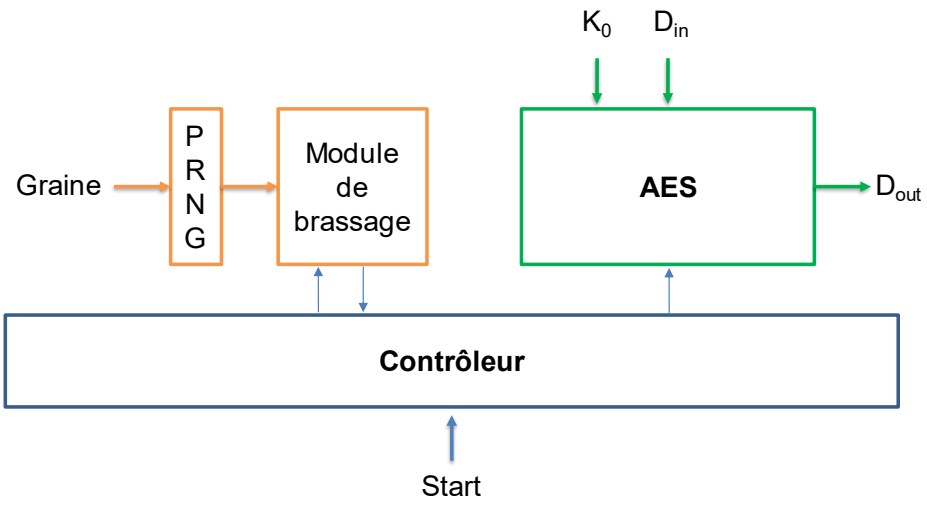


Figure 85: Schéma de principe (rappel)

Les octets des différents blocs sont calculés dans un ordre aléatoire, lui-même généré par notre *Module de brassage*. Ce dernier est alimenté par un PRNG et permet de générer l'ordre de traitement des octets et des blocs. Le contrôleur permet de générer l'itération courante du

compteur de bits au module de brassage et reçoit en retour l'indice de l'octet et du bloc à traiter. Il permet aussi de générer les adresses aux mémoires dans le chemin de données. Ce dernier reçoit en entrée le texte clair et la clé de chiffrement, et produit le texte chiffré à la fin des traitements.

Dans la section suivante, nous détaillons l'implémentation des deux alternatives possibles pour le module de brassage (limité et complet).

II. Génération des permutations par le module de brassage

1. Génération limitée des permutations

L'architecture est illustrée dans la *Figure 86*, un seul réseau de Benes 16x16 est utilisé pour générer les permutations des blocs et des 16 octets de la matrice d'état. Il est piloté par un mot de commande cmd de 56 bits générés par le PRNG. Un compteur Cpt, généré par le contrôleur, permet de sélectionner le bloc et l'octet à traiter.

Dans un premier temps, B permutations de 16 "indices" sont définies en *B* cycles d'horloge $Perm(\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\})$. Le résultat des N bits de poids faible de la permutation $Pb[N-1:0]$ est stocké dans la SRL 2 :

$$N = 1, Pb[N-1:0] = Perm(\{0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1\})$$

$$N = 2, Pb[N-1:0] = Perm(\{0,1,2,3,0,1,2,3,0,1,2,3,0,1,2,3\})$$

$$N = 3, Pb[N-1:0] = Perm(\{0,1,2,3,4,5,6,7,0,1,2,3,4,5,6,7\})$$

$$N = 4, Pb[N-1:0] = Perm(\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\})$$

Chaque permutation Pb permet de définir l'ordre de traitement des blocs durant 16 cycles d'horloge.

Par la suite, B permutations des 16 octets $Pn = Perm(\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\})$ de chaque bloc B_n sont stockées dans la SRL 1 de 4 bits en B cycles.

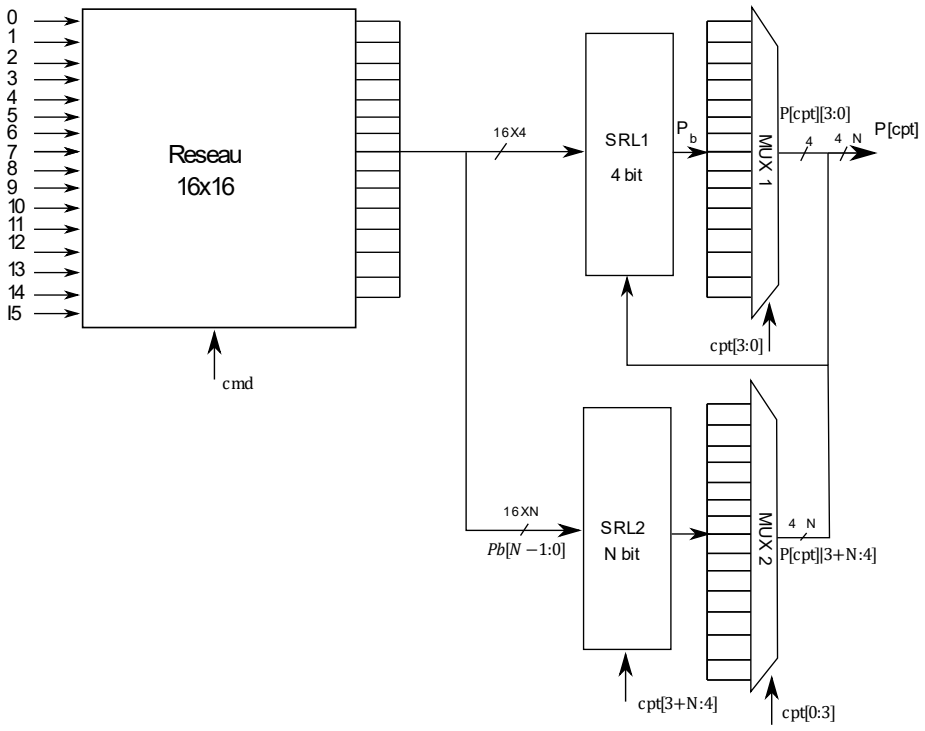


Figure 86 : Générateur de permutation

La $SRL\ 2$ est pilotée par N bits de l'itération du compteur $cpt[N+3:4]$. Elle délivre en sortie la permutation Pb , i.e. l'ordre de traitement des blocs durant les 16 cycles courants. Le $MUX\ 2$ est piloté par les 4 bits du compteur $cpt[\ 3:0]$ et permet de parcourir les permutations. Celle-ci, Pb, est sélectionnée parmi les 16 cycles courants.

La sortie de ce multiplexeur est l'indice du bloc $P[cpt][3+N:4]$. Cette valeur est utilisée comme adresse dans la $SRL\ 1$ afin de sélectionner la permutation courante P_b dans laquelle l'ordre de traitement des 16 octets de la matrice d'état est définis.

La sortie de la $SRL1$ est parcourue par $MUX\ 1$. La sortie de ce multiplexeur correspond à l'indice de l'octet à traiter $P[cpt]$. Cette disposition permet d'obtenir $2 \leq B \leq 16$.

Exemple de permutation :

On a $B = 2, N = 1$

- **Génération des permutations des blocs :**

  $Cmd1 = 6898e0f3$

  $Cmd2 = B4e715$

  $Pb[0] = \{0,0,1,1,0,1,1,1,0,0,0,1,1,0,0,1\}$

  $Cmd1 = DA738A$

$Cmd2 = 344C7079$

$Pb[1] = \{0,1,0,1,1,1,0,0,1,1,0,1,0,0,0,1\}$

- **Génération des permutations des octets :**

$Cmd1 = 69CE2B$

$Cmd2 = D131C1E7$

$P[1] = \{D, c, e, 0, b, 9,4, a, f, 7,3,2,5,1,6,8\}$

$Cmd1 = D39c56$

$Cmd2 = A26383ce$

$P[2] = \{9,4, c, a, d, b, 5,2,0,1,8,3, e, f, 7,6\}$

Le résultat de la première permutation générée dans l'intervalle du compteur $cpt = [0,15]$ est illustrée dans le

Tableau *19*. A chaque itération cpt, un octet parmi les 16 de la matrice d'état d'un des deux bloc 0 et 1 est sélectionné via $P[cpt]$. L'ordre de traitement des blocs durant les 16 premiers cycles est défini par la première permutation $Pb[0]$ générée par les mots de commande $Cmd1 = 6898e0f3$ et $Cmd2 = B4e715$ à l'entrée du réseau de Benes.

L'ordre de traitement des octets du premier bloc est défini par la permutation $P[0]$. Cette permutation est utilisée à chaque fois que le bloc 0 est sélectionné. L'octet à calculer est alors défini en sortie du multiplexeur MUX2 par les 3 bits de poids faible de l'itération courante cpt.

Tableau 19: Résultat des permutations générée entre l'itération 0 et 15 du compteur cpt

cpt	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$P[cpt][N+3:4]$	0	0	1	1	0	1	1	1	0	0	0	1	1	0	0	1
$P[cpt][3:0]$	**9**	**4**	**D**	**C**	**C**	**E**	**0**	**B**	**A**	**D**	**B**	**9**	**4**	**5**	**2**	**A**
$P[cpt]$	9	4	1D	1C	0C	1E	10	1B	0A	0D	0B	19	14	5	2	1A

L'ordre de traitement des octets du second bloc est défini par la permutation $P[1]$ générée par les mots de commande $Cmd1 = D39c56$ et $Cmd2 = A26383ce$. La permutation courante $P[cpt]$ correspond alors à la concaténation du bloc courant $Pb[0][cpt]$ et de l'octet à calculer $P[0][cpt[3:0]] : P[cpt] = Pb[0][cpt] \parallel P[0][cpt[3:0]]$.

Le résultat de la permutation générée dans l'intervalle $cpt = [16,31]$ est illustré dans le Tableau 20

Tableau 19. Cette fois l'ordre de traitement des blocs est défini par la permutation $P[1]$ générée par les mots de commande $Cmd1 = DA738A$ et $Cmd = 344C7079$ du Benes. L'ordre de traitement des octets restant du bloc 0 et 1 sont alors défini par les permutations $P[0]$ et $P[1]$.

Tableau 20 : Résultat des permutations générée entre l'itération 16 et 31 du compteur cpt

cpt	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**1A**	**1B**	**1C**	**1D**	**1E**	**1F**
$P[cpt][N+3:4]$	0	1	0	1	1	1	0	0	1	1	0	1	0	0	0	1
$P[cpt][3:0]$	**0**	**F**	**1**	**7**	**3**	**2**	**8**	**3**	**5**	**1**	**E**	**6**	**F**	**7**	**6**	**8**
$P[cpt]$	0	1F	1	17	13	12	8	3	15	11	0E	16	0F	7	6	18

2. Génération complète des permutations

Dans cette approche, l'aléa est maximal. Pour ce faire nous utilisons un réseau de Benes $2^{N+4} \times 2^{N+4}$, dont le code est écrit de façon générique en fonction du nombre de blocs qui seront traiter, impactant donc la taille du réseau de Benes à utiliser.

Dans cette architecture, la valeur minimale de N est 1, ce qui correspond à deux blocs dont les permutations sont générées par un réseau de Benes 32x32. Si $N = 0$, alors un réseau de Benes 16x16 serait alors généré.

Le réseau est composé de $2 \times (N+4) - 1$ étages, chacun comportant de $\frac{2^{N+4}}{2}$ commutateurs. Ces derniers se composent de deux multiplexeurs *2 vers 1* de $N+4$ bits.

Une matrice *S* de connexion de $N+4$ bits de dimension $(2 \times (N+3)) + 1, 2^{N+4})$, permets de lier les commutateurs entre chaque étage (par ex. pour un réseau de Benes 16×16 -avec $N = 0$- S est de dimension (7,16)).

Le réseau de Benes est construit de façon symétrique (cf. *Figure 87*). Il est constitué de motifs (blocs en vert pointillé) qui se dédoublent et dans lesquels les liaisons de deux groupes de commutateurs se croisent (groupes rouges, groupes bleus).

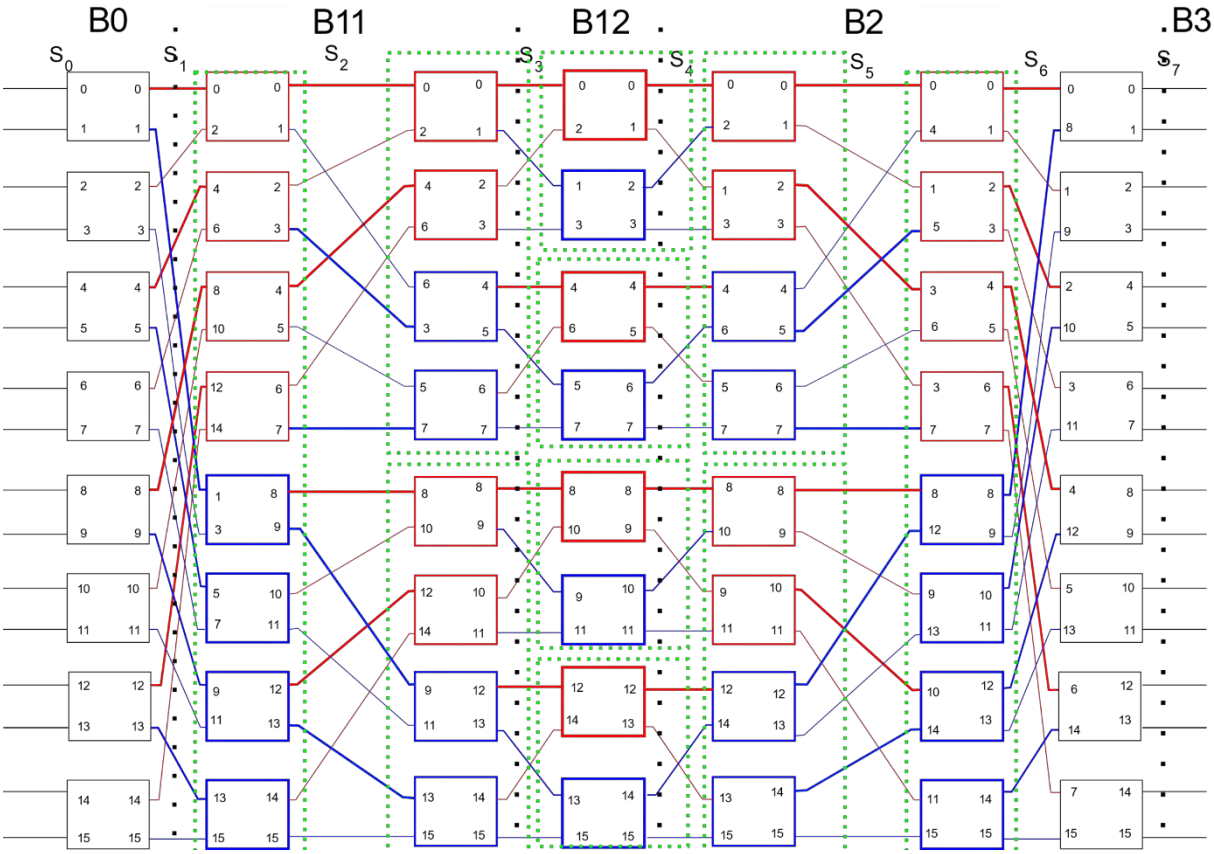


Figure 87: Composition d'un Benes 16x16

Le réseau est constitué de $2 \times (N + 4 - 1) + 1$ étages, eux-mêmes constitués de $\frac{2^{N+4}}{2}$ commutateurs. L'architecture du module de brassage est illustrée dans la *Figure 88*. Le résultat de la permutation P à la sortie du réseau est alors stocké dans 2^{N+4} registres de $N + 4$ bits, puis un multiplexeur 2^{N+4} vers 1 piloté par le compteur cpt, permet de parcourir les 2^{N+4} permutations et délivre l'octet $P[cpt][3:0]$ du bloc courant à calculer $P[cpt][N + 4:4]$.

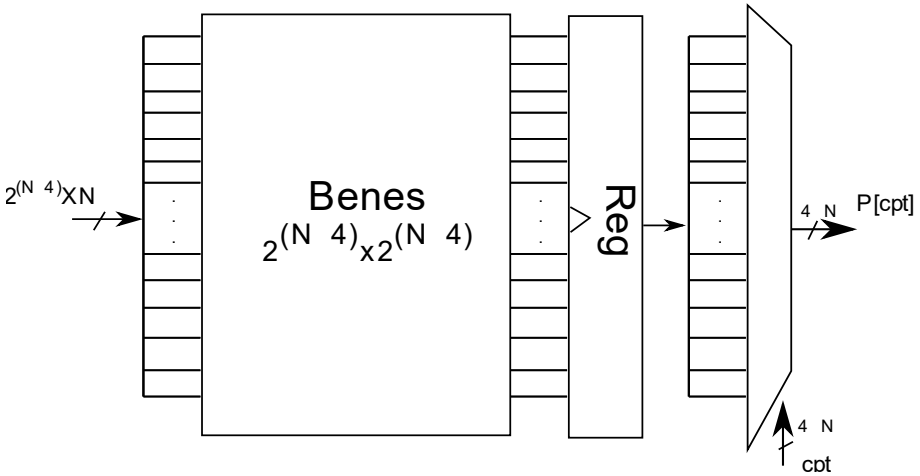


Figure 88: Chemin de données du module de brassage dans l'architecture avec ajout d'aléa complet

3. Architecture Multi-blocs dynamique

Le nombre de blocs B est défini dans la bibliothèque par une constante N avec $B = 2^N$.

Le chemin de données est le même que celui de l'architecture mono bloc avec ajout d'aléa complet (cf. *Figure 89*) mais sans aléa sur le stockage, à l'exception des mémoires dont la taille est dans ce cas de $16 \times B$ octets.

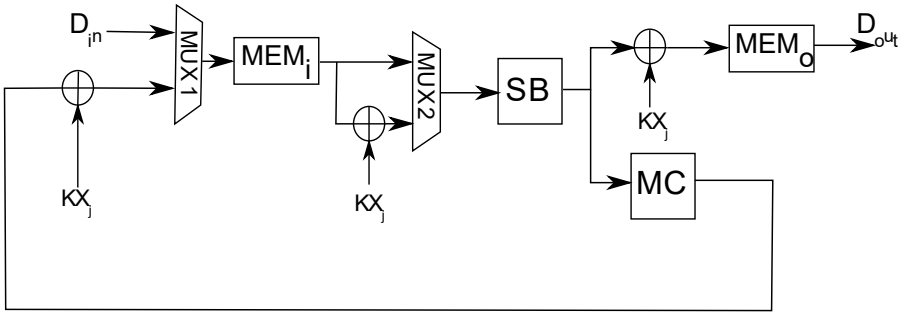


Figure 89 : Chemin de données de l'architecture avec ajout d'aléa complet

Nous avons utilisé des mémoires qui permettent d'obtenir un grand espace d'adressage sans surcoût majeur sur un FPGA. En effet, il est possible de stocker les 16 octets des quatre blocs à l'aide de 8 LUTs (64 adresses de 8 bits). Les adresses des mémoires de $N + 4$ bits sont délivrées par le contrôleur.

Durant la phase d'initialisation, le texte clair découpé en B blocs est stocké dans MEM_i, pendant les $16 \times B$ premiers cycles. L'opération du ShiftRows est assurée par un adressage adéquat de la mémoire MEM_i. Chaque clé de ronde wKX est calculée et stockée dans les premiers 16 cycles de la ronde courante X.

Le calcul de chaque ronde est effectué en deux phases de $16 \times B$ cycles : dans la première phase les opérations indépendantes ShiftRows (SR), SubBytes (SB) et la première partie du MixColumns ($\mathrm{MC}\,[1/2]$) sont effectuées sur les 16 octets des B blocs. L'opération Key-Whitening $\mathrm{ADK0}$ est effectuée en parallèle de ces opérations durant la première ronde. Le pilotage du $MUX\ 2$ permet de réaliser l'opération du Key-Whitening lors de la première ronde uniquement ; dans la seconde phase, le calcul de la ronde est complété par la seconde partie du MixColumns ($\mathrm{MC}\,[2/2]$) et l'opération d'AddRoundKey.

Les adresses des mémoires dans le module du MixColumns $\mathrm{MEM}_0,\ MEM_1, MEM_2, MEM_3$ sont délivrées par le contrôleur tel que :

$$Ad_i = P\big[cpt[N+3:2]\big] \parallel Ai\left[P\big[cpt[1:0]\big]\right]$$

- Ad_i : L'adresse de chaque mémoire $\mathrm{MEM_i}$,
- $P[cpt[N+3{:}2]]$: L'indice de du bloc $cpt[N+3:4]$ et de la colonne $cpt[3:2]$
- Ai : La table utilisé dans mémoire MEM_i pour stocker les dépendances entre les octets à calculer $B_{c,j}$ et les quatre octets $b_{c,j}$ de la colonne courante c_i sont stockés.

Lors de la dernière ronde, le résultat de l'opération d'AddRoundKey $ADKX$ qui suit celle du SubBytes, est stocké dans MEM_0. Dans la phase de génération de la sortie, le résultat est stocké dans la mémoire et écrit sur la sortie D_{OUT}. La latence globale incluant l'initialisation, le chiffrement et l'écriture de la sortie est de $336 \times B$ cycles d'horloge.

III. Résultat d'implémentation des architectures AES-CTR

Dans cette section, nous présentons nos deux architectures multi-bocs, dont la différence repose dans la façon de générer les permutations dans le module de brassage. Dans la première version (permutations limitées), on utilise donc un unique réseau de Benes 16x16 avec le chiffrement de deux blocs AES en parallèle (*2B_Ben-16_Mem*), alors que la seconde repose sur un réseau de Benes avec une taille dynamique utilisé pour chiffrer nos deux blocs, soit un réseau de Benes 32x32 (*2B_Ben-32_Mem*).

Nota bene :

Afin d'évaluer le surcoût en surface et les gains en performances de nos architectures, le module de chiffrement complet (avec module shuffling) est synthétisé.

Toutefois, dans le cas des architectures multi-blocs, le nombre d'entrées/sorties est supérieurs au nombre maximal d'*IOB (I/O Blocks)* géré par la cible xc6slx9 (contrainte technique liée à la plateforme de tests). En effet, l'outil transforme toutes les entrées sorties d'un module, en IOB durant la synthèse.

Afin de passer l'étape de routage, la génération d'IOB est désactivée en décochant la case « *trim unconnected signals* » dans les propriétés *MAP* et la case « *Add I/O Buffers* » dans Xilinx specific Options.

Par conséquence, les résultats en performance (fréquence maximal) ne peuvent pas être obtenus. Les résultats en surface sont détaillés dans le Tableau 21.

Tableau 21: Résultats en surface des architectures multi-blocs implémentées sur la cible xc6slx9

Architectures	[26]* xc6slx9	Ben-16_Mem	2B_Ben-16_Mem	2B_Ben-32_Mem
Fréquence (Mhz)	114,09	104.97	-	-
Latence	160	336	672	672
Débit (Mbit/s)	91,272	39,9	-	-
Slice	81	109	321	765
Slice LUT	218	272	486	957
Slice Register	97	49	82	234
BRAM	0	0	0	0
DSP	0	0	0	0
*Ces résultats sont ceux de notre architecture				

La version limitée a été implémentée sur 321 slices avec un surcoût de 296% en terme de surface par rapport à la version non protégée de référence [CB12], et un surcoût de 194% par rapport à la version Ben-16_Mem mono-bloc.

Afin d'obtenir les résultats en performances, les architectures étaient implémentées sur la cible *xc6slx16*. Celle-ci est similaire à la cible *xc6slx9*, mais a un nombre d'IOB plus important. Le Tableau 22 illustre les résultats synthèse de l'architecture mono bloc Ben-16_Mem et compare les résultats des deux cibles *xc6slx9* et *xc6slx16*.

On remarque qu'en termes de slices, les résultats sont similaires dans l'architecture mono bloc sur les deux cibles (5 slices de différences). La fréquence quant à elle est supérieure dans la cible *xc6sl16*. On constate également une perte de fréquence de 4% dans la version limitée et de 10% dans la version avec aléa complet, par rapport à la version mono bloc. On peut en déduire que la fréquence et d'autant plus réduit dans la cible *xc6slx9*. La chute de débit dans la version avec aléa limitée par un facteur de ***x1.04*** et par un facteur de ***x1,11*** dans la version avec aléa complet par rapport à la version mono bloc implémentée sur la même cible.

Tableau 22: Résultats en surface des architectures multi-blocs implémenté sur la cible xc6slx16

Architectures	Ben-16_Mem		*2*B_Ben-16_Mem	*2*B_Ben-32_Mem
	xc6slx9	*xc6slx16*	*xc6slx16*	
Fréquence (Mhz)	104,37	113.63	108,63	101,6
Latence	336	336	672	672
Débit (Mbit/s)	39,76	43,29	41,38	38,73
Slice	143	148	305	787
Slice LUT	344	344	491	960
Slice Register	49	49	82	234
BRAM	0	0	0	0
DSP	0	0	0	0
Débit/Surface	***0.27***	***0.29***	***0,135***	***0.049***

IV. Evaluation de sécurité

Dans nos architectures, un vecteur d'initialisation est fourni par le PRNG et stocké durant le premier chiffrement, avant d'être utilisé dans la suite des opérations. Pour cette évaluation de sécurité, nous évaluons l'aléa sur 2 blocs. La latence de chiffrement est donc doublée, de même que le nombre de points d'échantillonnage (3000 points au lieu de 1500).

La *Figure 90* illustre les traces de consommation d'une architecture Ben-16 avec des mémoires distribuées et sans aléa sur les adresses (c.à.d. Ben-16_Mem du chap. 4). Cette architecture est limitée dans les permutations (2B_Ben-16_Mem) et la *Figure 91* illustre les traces de consommation d'une architecture Ben-16 avec un aléa complet (2B_Ben-32_Mem).

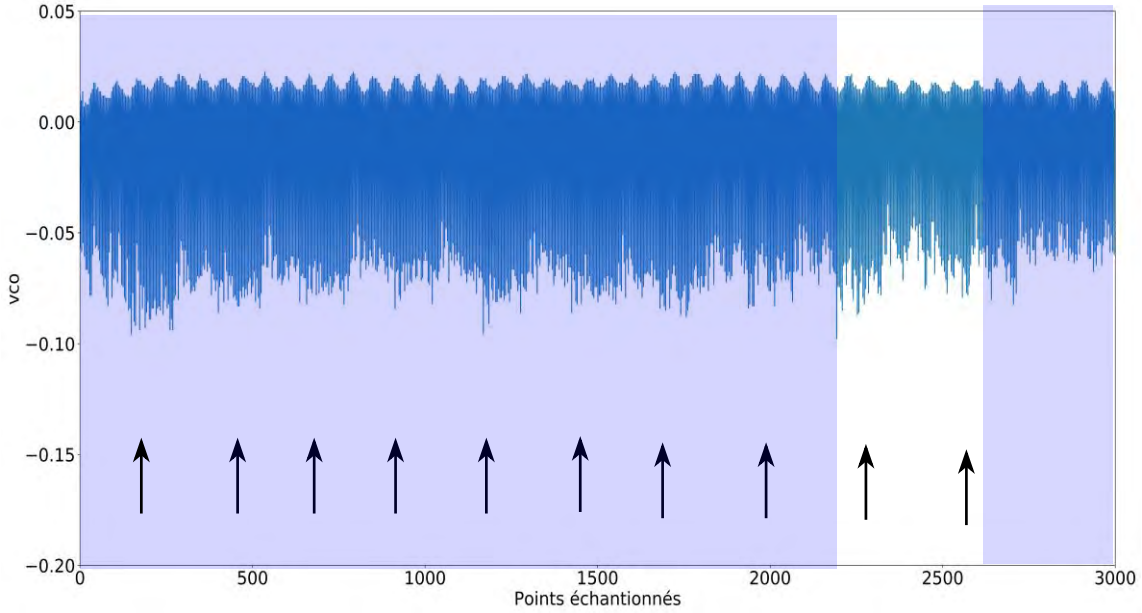


Figure 90 : Traces de consommation de l'architecture 2B_Ben-16_Mem

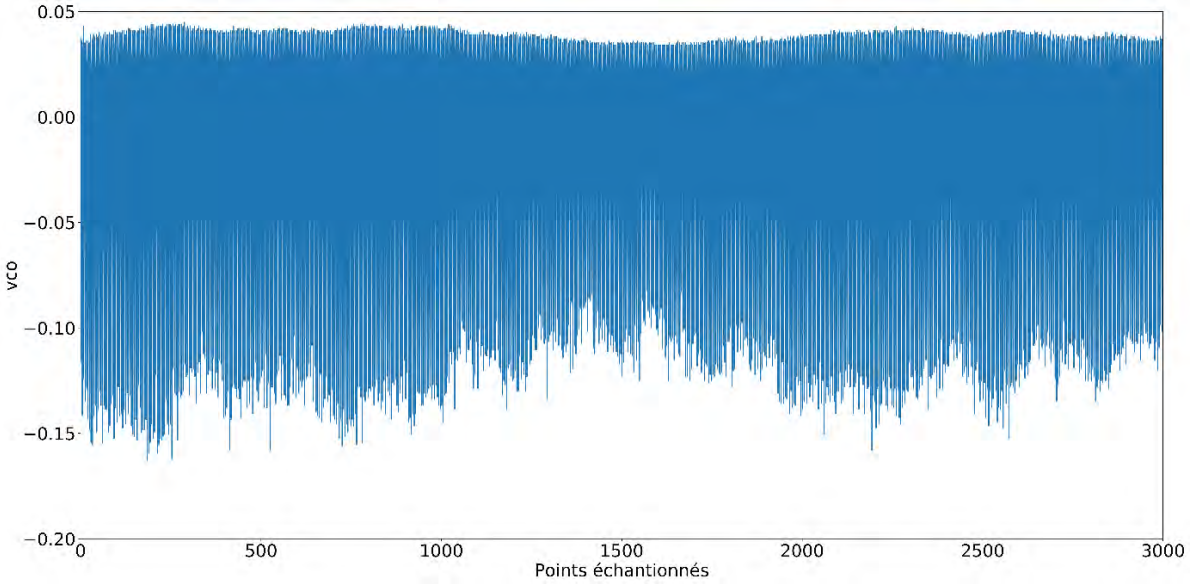


Figure 91 : Traces de consommation de l'architecture 2B_Ben-32_Mem

On remarque que l'architecture limitée en permutations consomme deux fois moins que celle avec un aléa complet. De plus, les rondes sont beaucoup moins facilement distinguables dans la trace de consommation, dans le cas d'un chiffrement avec un aléa complet.

Le résultat de l'attaque CPA (sur un million de traces) avec aléa limité est illustré dans la *Figure 92*. On remarque des pics avec des coefficients de corrélation compris entre 0.005 et 0.01 (entre 2400 et 2600 traces). En revanche, on constate à nouveau que dans l'architecture

Ben-16_Mem avec aléa complet illustrée dans la *Figure 93* aucun pic de corrélation ne ressort.

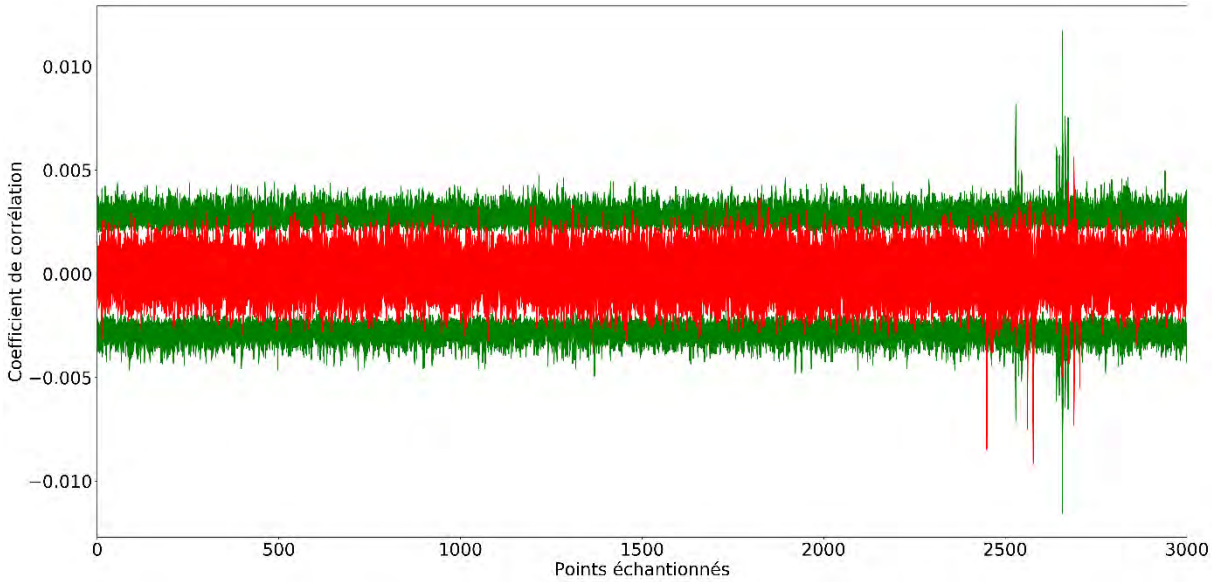


Figure 92 : Courbe CPA pour l'architecture 2B_Ben-16_Mem

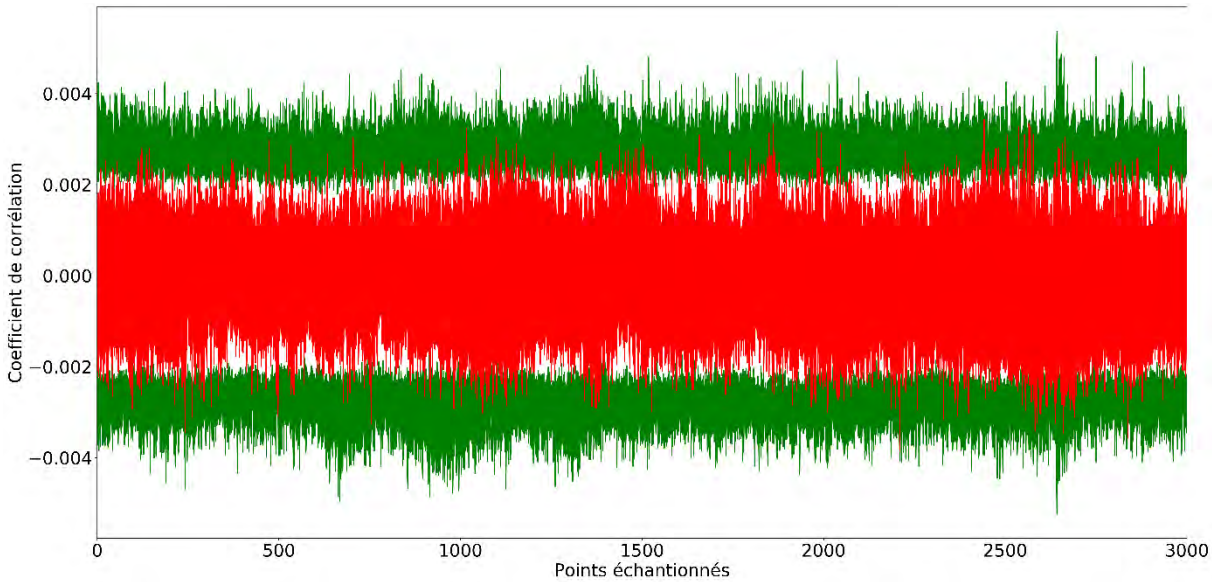


Figure 93 : Courbe CPA pour l'architecture 2B_Ben-32_Mem

La *Figure 94* illustre le résultat des attaques CPA en fonction du nombre d'octets révélés par rapport au nombre de traces sur les trois versions des architectures Ben-16_Mem : mono-bloc en mode ECB et multi-blocs (2 blocs) avec aléa limité et aléa complet.

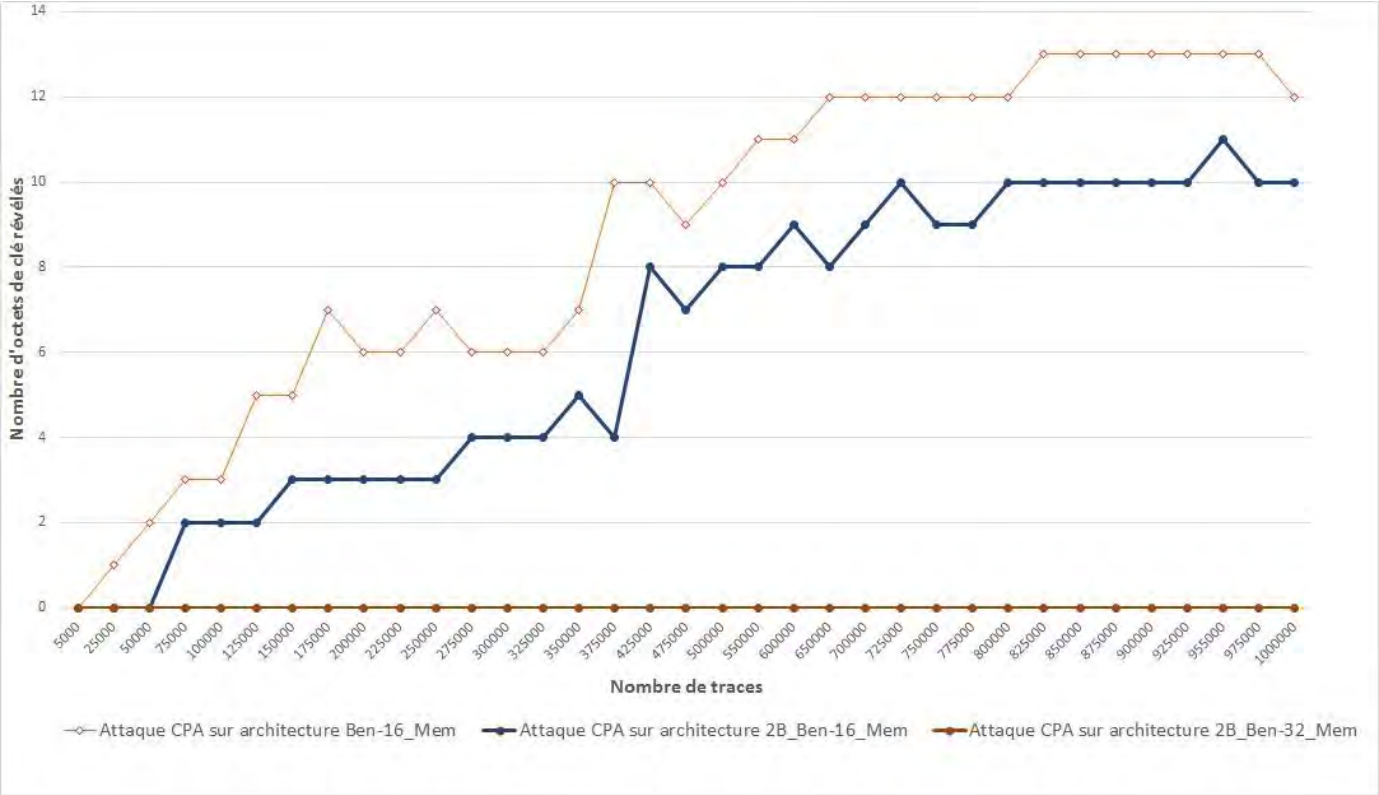


Figure 94 : Résultat des attaques CPA et CPA intégrée sur les différentes architectures

En utilisant un million de traces, 12 octets de la clé sont révélés dans l'architecture mono-bloc (et 50% de la clé avec 365k traces). Dans l'architecture multi-blocs avec aléa limité, il faut un million de traces pour révéler 10 octets (et 50% de la clé est révélée avec 485k traces). Pour l'architecture avec aléa complet, aucun octet n'est révélé.

Une attaque CPA intégrée a été réalisée sur les deux architectures multi-blocs avec une fenêtre de 200 points dans l'intervalle [2400 :2600], mais de nouveau aucun octet n'a été révélé. En effet, l'augmentation de la fenêtre engendre mécaniquement l'augmentation de l'accumulation du bruit. L'architecture multi-blocs avec aléa complet se révèle, selon nos expérimentations, la plus robuste de nos travaux. Il en va de même s'il l'on cherche à réduire la taille de la fenêtre.

V. Conclusion

Dans ce chapitre, nous avons détaillé les deux architectures de notre dernière contribution, dans lequel l'aléa est ajouté dans l'ordre de traitement des blocs, ainsi que dans l'ordre de traitement des 16 octets de la matrice, pour chacun des blocs dans une architecture AES-CTR. Malgré un surcoût non négligeable en surface, une amélioration importante en termes de sécurité a été obtenue.

On peut conclure de ces expériences que la réduction de l'aléa dégrade fortement la sécurité, alors que le gain en surface est lui limité. L'architecture avec aléa complet a quant à elle, montrée une très forte résistance face aux attaques CPA et semble *immunisée* contre l'attaque CPA intégrée. Cette architecture est ainsi la plus robuste de nos travaux.

La section suivante présente une conclusion globale, ainsi que les perspectives de travaux à venir.

Chapitre 6 Conclusion et perspectives

Dans ce manuscrit, nous avons détaillé les différentes architectures proposées avec ajout d'aléa dans l'ordre des calculs ainsi que dans les adresses de stockage des données. Ces modèles d'architecture présentent différents niveaux de compromis *surface vs performances vs sécurité*. Dans une première approche, l'ajout d'aléa est limité afin d'optimiser les performances de calcul, et réduire le surcoût en latence. Dans ce cas, l'aléa est uniquement ajouté dans l'ordre de traitement des colonnes de la matrice d'état de l'AES, ainsi que dans l'ordre de traitement des octets de ces colonnes. Dans une deuxième approche, nus étendons l'ajout d'aléa à l'adressage et la mémorisation des données. Concernant la gestion matérielle de cet aléa, trois alternatives ont été étudiées. La première est basée sur l'utilisation de SRLs (des composant internes aux FPGA utilisés) pour le stockage. La seconde utilise des mémoires distribuées (mais sans aléa pour la gestion des adresses). La troisième utilise des mémoires distribuées avec aléa pour les adresses.

Pour chacune de ces approches, deux versions ont été conçues. La première utilise un réseau de permutation Benes 16X16 capable de générer 16 ! permutations différentes en un cycle d'horloge, et la seconde utilise un réseau de permutation Omega, capable de générer 2^{32} permutations en un cycle d'horloge. Le surcoût en terme de surface et de performances de ces implémentations a été évalué et mis en perspective avec le design de référence non protégé que nous avons reproduit selon l'architecture AES 8 bits décrite dans [26]. Le surcoût nos approches est en moyenne, et selon la configuration choisie, entre 32 % et 88 % en termes de surface, ce qui correspond à notre contrainte initiale. En terme de performances les résultats ont montré une perte de débit pouvant aller jusqu'à un facteur de 2,44.

Contrairement aux précédents travaux existant de l'état de l'art, ces architectures ont montré une résistance contre les attaques CPA et CPA intégrées, lorsque l'aléa est ajouté à la fois sur l'ordre des calculs de l'AES et sur la gestion et la mémorisation des données. Il faut également signaler que les résultats ont montré un impact non négligeable des options de synthèse sur les résultats. En effet, le choix de ces derniers impacte fortement la consommation de puissance de l'architecture résultante, et cela se reflète donc sur la puissance du signal utile pour l'attaquant (le signal comportant les informations de fuites). Plus cette puissance consommée est réduite, plus le rapport signal sur bruit est réduit. Une autre alternative consiste à étendre l'aléa dans le temps pouvant rendre l'intégration totalement inefficace présentée dans le chapitre 5 et représente donc notre contribution finale

dans cette thèse. Cela consiste à ajouter l'aléa non seulement sur le traitement des 16 octets de la matrice d'état, mais aussi sur de multiples blocs de chiffrement.

Une architecture d'un algorithme AES en mode CTR, avec ajout d'aléa sur des blocs multiples, a également été conçue. Par ailleurs, le code VHDL de cette architecture est dynamique, de sorte que la taille des mémoires ainsi que le contrôleur soient adéquates au nombre de blocs sélectionnés. Or, l'augmentation d'aléa implique un réseau de permutation couteux pour un ajout d'aléa maximal. Deux versions ont alors été conçues :

- La première est limitée en permutation, où seul un réseau de Benes 16x16 est utilisé pour générer l'aléa de l'ordre de traitement des blocs, ainsi que l'ordre de traitement des octets de chaque bloc.
- La seconde utilise un réseau de Benes supportant toutes les permutations possibles.

Les résultats en synthèse ont montré un surcoût important en surface supérieur au double. La version avec aléa complet a cependant montré une forte résistance aux attaques par observation. En effet, l'architecture multi bloc avec aléa complet est l'architecture la plus robuste parmi les architectures présentées.

L'architecture ayant le meilleur compromis sécurité, surface et performance est donc l'architecture mono-bloc utilisant un réseau de Benes 16x16 avec dans l'ordre des calculs ainsi que dans les adresses de stockage des données. En effet cette dernière est robuste contre les attaques CPA et CPA intégrées avec in surcout de 88% en termes de slices et une perte de débit 2.44.

Durant notre compagne d'évaluation, nous avons utilisé des traces de consommation brute, sans prétraitement au préalable, une évaluation plus poussée pourraient être menée par un filtrage numérique en utilisant le filtre de kalman [88], un filtre à réponse impulsionnelle infini, utilisé généralement dans le domaine d'électronique. Ce filtre est capable d'estimer l'état courant en ayant connaissance de l'état précédent et les mesures actuelles. En premier temps le l'état courant est prédit en se basant sur l'état à l'instant précédant. En second temps l'état prédit est corrigé par l'observation de l'instant courant.

De plus, des attaques plus puissantes pourraient être menées pour compléter l'évaluation de la contremesure d'ajout d'aléa en matérielle comme les attaques *template* et l'apprentissage profond. Ces deux méthodes sont comparées dans le travail proposé dans Zotkin et al [89] sur l'algorithme AES protégée avec des contremesures connu (désynchronisation, ajout

d’aléa et masquage). Dans ce type d’attaque est constitué en deux phases. La première phase est une phase d’apprentissage et la seconde est une phase de test. Ce type d’attaque est très puissant, mais difficile à monter dans la pratique.

Aussi, il serait intéressant de mener des études supplémentaires afin d’identifier et de caractériser les sources de fuites induites par les options de synthèse. Pour cela, il est possible de s'appuyer sur des outils statistiques comme par exemple la méthode Vector Leakage Assessment (TVLA) (qui s'appuie sur le "test T de Welch") ou des analyses basées sur l'information mutuelle (Mutual Information Analysis - MIA) en complément des attaques CPAs utilisées dans ces travaux de thèses.

Enfin, une évaluation poussée pourrait être effectuée en combinant l’ajout d’aléa avec le masquage dans une implémentation matérielle. Le masquage pourrait être utilisé aussi pour protéger l’opération d’expansion de la clé, qui ne peut être protégée par l’ajout d’aléa.

Références bibliographiques

[1] M. Hung, «Leading the IoT,» 2017. [En ligne]. Available: https://www.gartner.com/imagesrv/books/iot/iotEbook_digital.pdf.

[2] F. Dahlqvist, M. Patel, A. Rajko et J. Shulman, «Growing opportunities in the Internet of Things,» 2019. [En ligne]. Available: https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things#.

[3] D. Bonderud, «Leaked Mirai Malware Boosts IoT Insecurity Threat Level,» 2016. [En ligne]. Available: https://securityintelligence.com/news/leaked-mirai-malware-boosts-iot-insecurity-threat-level/.

[4] C. Adam, «Mixed-Signal Hardware Security Thwarts Powerful Electromagnetic Attacks,» 2020. [En ligne]. Available: https://www.purdue.edu/newsroom/releases/2020/Q1/mixed-signal-hardware-security-thwarts-powerful-electromagnetic-attacks.html.

[5] D. GENKIN, L. PACHMANOV, I. PIPMAN et T. ERAN, «ECDH key-extraction via low-bandwidth electromagnetic attacks on PCs,» *Cryptographers' Track at the RSA Conference,* pp. 219-235, 2016.

[6] D. GENKIN, L. PACHMANOV, I. PIPMAN et T. ERAN, «Stealing keys from PCs using a radio: Cheap electromagnetic attacks on windowed exponentiation,» *International workshop on cryptographic hardware and embedded systems,* pp. 207-228, 2015.

[7] R. SCHUSTER, V. SHMATIKOV et E. TROMER, «Beauty and the burst: Remote identification of encrypted video streams,» *26th {USENIX} Security Symposium ({USENIX} Security 17),* pp. 1357-1374, 2017.

[8] A. KERCKHOFFS, « La cryptographic militaire.,» *Journal des sciences militaires,* pp. 5-38, 1883.

[9] E. BARKER, W. BARKER, W. BURR, T. Polk et S. Miles, «Recommendation for key management: Part 1,» *General. National Institute of Standards and Technology, Technology Administration.,* 2006.

[10] D. E. Standard, «Data encryption standard,» *Federal Information Processing Standards Publication,* p. 112, 1999.

[11] N. F. Pub, «Specification for the advanced encryption standard (aes),» 2001.

[12] C. E. SHANNON, « Communication theory of secrecy systems.,» *The Bell system technical journal,* pp. vol. 28, no 4, p. 656-715., 1949.

[13] L. HATHAWAY, «National policy on the use of the advanced encryption standard (AES) to protect national security systems and national security information.,» *National Security Agency,* 2003.

[14] L. BOSSUET, «Approche didactique pour l'enseignement de l'attaque DPA ciblant l'algorithme de chiffrement AES,» *J3eA,* vol. 12, p. 4, 2012.

[15] «SECURE IC REVERSE ENGINEERING & DATA EXTRACTION,» [En ligne]. Available: https://www.texplained.com/.

[16] J. KELSEY, B. SCHNEIER, D. WANGER et C. HALL, «Side channel cryptanalysis of product ciphers,» *European Symposium on Research in Computer Security,* pp. 97-110, 1998.

[17] P. C. KOCHER, «Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems,» *Annual International Cryptology Conference,* pp. 104-113, 1996, August.

[18] P. KOCHER, J. JAFFE et B. JUN, «Differential power analysis,» *Annual international cryptology conference,* pp. 388-397, 1999.

[19] S. MANGARD, E. OSWALD et T. POPP, Power analysis attacks: Revealing the secrets of smart cards., vol. 31, Springer Science & Business Medi, 2008.

[20] A. MORADI et T. SCHNEIDER, «Improved side-channel analysis attacks on Xilinx bitstream encryption of 5, 6, and 7 series,» *International Workshop on Constructive Side-Channel Analysis and Secure Design,* pp. 71-87, 2016.

[21] A. VASSELLE et A. WURCKER, «Optimizations of Side-Channel Attack on AES MixColumns Using Chosen Input,» *IACR Cryptol. ePrint Arch,* p. 343, 2019.

[22] A. HODJAT et I. VERBAUWHEDE, «A 21.54 Gbits/s fully pipelined AES processor on FPGA,» *2th Annual IEEE Symposium on Field-Programmable Custom Computing Machines.,* pp. 308-309, 2004.

[23] P. CHODOWIEC et K. GAJ, «Very compact FPGA implementation of the AES algorithm.,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 319-333, 2003.

[24] P. HAMALAINEN, T. ALHO, M. HANNIKAINEN et T. D. HÄMÄLÄINEN, «Design and implementation of low-area and low-power AES encryption hardware core,» *9th EUROMICRO conference on digital system design (DSD'06) ,* pp. 577-583, 2006.

[25] A. SATOH, S. MORIOKA, K. TAKANO et S. MUNETOH, «A compact Rijndael hardware architecture with S-box optimization,» *International Conference on the Theory and Application of Cryptology and Information Security,* pp. 239-254, 2001.

[26] J. CHU et M. BENAISSA, «Low area memory-free FPGA implementation of the AES algorithm,» *22nd International Conference on Field Programmable Logic and Applications (FPL),* pp. 623-626, 2012.

[27] P. SASDRICH et T. GÜNEYSU, «A grain in the silicon: SCA-protected AES in less than 30 slices.,» *2016 IEEE 27th International Conference on Application-specific Systems,* pp. 25-32, 2016.

[28] S. N. DHANUSKODI et D. HOLCOMB, «An improved clocking methodology for energy efficient low area aes architectures using register renaming,» *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED),* pp. 1-6, 2017.

[29] L. GOUBIN, «A sound method for switching between boolean and arithmetic masking,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 3-15, 2001.

[30] M.-L. AKKAR et C. GIRAUD, «An implementation of DES and AES, secure against some attacks,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 309-318, 2001.

[31] T. S. MESSERGES, «Using second-order power analysis to attack DPA resistant software,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 238-251, 2000.

[32] A. SHAMIR, «How to share a secret,» *Communications of the ACM,* vol. 22, n° %111, pp. 612-613, 1979.

[33] M. RIVAIN et E. PROUFF, «Provably secure higher-order masking of AES,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 413-427, 2010.

[34] S. NIKOVA, C. RECHBERGER et V. RIJMEN, «Threshold implementations against side-channel attacks and glitches,» *International conference on information and communications security,* pp. 529-545, 2006.

[35] S. CHARI, C. S. JUTLA, J. R. RAO et P. ROHATGI, «Towards sound approaches to counteract power-analysis attacks,» *Annual International Cryptology Conference,* pp. 398-412, 1999.

[36] F. REGAZZONI, Y. WANG et F.-X. STANDAERT, «FPGA implementations of the AES masked against power analysis attacks,» *Proceedings of COSADE,* vol. 2011, pp. 56-66, 2011.

[37] K. TIRI et I. VERBAUWHEDE, «A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation,» *Proceedings Design, Automation and Test in Europe Conference and Exhibition,* pp. 246-251, 2004.

[38] A. SHAMIR, «Protecting smart cards from passive power analysis with detached power supplies,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 71-77, 2000.

[39] D. MESQUITA, J.-D. TECHER, L. TORRES, G. SASSATELLI, G. CAMBON, M. ROBERT et F. MORAES, «Current mask generation: a transistor level security against DPA attacks,» *In 2005 18th Symposium on Integrated Circuits and Systems Design,* pp. 115-120, 2005.

[40] S. GUILLEY, L. SAUVAGE, J.-L. DANGER et P. HOOGCORST, «Area optimization of cryptographic co-processors implemented in dual-rail with precharge positive logic,» *International Conference on Field Programmable Logic and Applications,* pp. 161-166, 2008.

[41] M. NASSAR, S. BHASIN, J.-L. DANGER, G. DUC et S. GUILLEY, BCDL: a high speed balanced DPL for FPGA with global precharge and no early evaluation, IEEE, Éd., 2010, pp. 849-854.

[42] K. A. M. TIRI et I. VERBAUWHEDE, «A dynamic and differential CMOS logic with signal independent power consumption to withstand differential power analysis on smart cards.,» *Proceedings of the 28th European solid-state circuits conference,* pp. 403-406, 2002.

[43] L. SAUVAGE, GUILLEY, D. J.-L. Sylvain, Yves, Mathieu et M. NASSER, «SAUVAGE, Laurent, GUILLEY, Sylvain, DANGER, Jean-Luc, et al. Successful attack on an FPGA-based WDDL DES cryptoprocessor without place and route constraints,» *Design, Automation & Test in Europe Conference & Exhibition,* pp. 640-645, 2009.

[44] J. A. AMBROSE, A. IGNJATOVIC et S. PARAMESWARAN, «CoRaS: A multiprocessor key corruption and random round swapping for power analysis side channel attacks: A DES case study,» *IEEE International Symposium on Circuits and Systems (ISCAS),* pp. 253-256, 2012.

[45] J. A. AMBROSE, S. PARAMESWARAN et A. IGNJATOVIC, «MUTE-AES: A multiprocessor architecture to prevent power analysis based side channel attack of the

AES algorithm,» *IEEE/ACM International Conference on Computer-Aided Design,* pp. 678-684, 2008.

[46] J. IRWIN, D. PAGE et N. P. SMART, «Instruction stream mutation for non-deterministic processors,» *Proceedings IEEE International Conference on Application-Specific Systems, Architectures, and Processors,* pp. 286-295, 2002.

[47] G. AGOSTA, A. BARENGHI et G. PELOSI, «A code morphing methodology to automate power analysis countermeasures.,» *Proceedings of the 49th Annual Design Automation Conference,* pp. 77-82, 2012.

[48] J. A. AMBROSE, R. G. RAGEL et S. PARAMESWARAN, «RIJID: random code injection to mask power analysis based side channel attacks,» *Proceedings of the 44th annual Design Automation Conference,* pp. 489-492, 2007.

[49] A. G. BAYRAK, N. VELICKOVIC, P. IENNE et w. BURLESON, «An architecture-independent instruction shuffler to protect against side-channel attacks,» *ACM Transactions on Architecture and Code Optimization (TACO),* vol. 8, n° %14, pp. 1-19, 2012.

[50] D. MAY, H. L. MULLER et N. P. SMART, «Non-deterministic processors,» *Australasian Conference on Information Security and Privacy,* pp. 115-129, 2001.

[51] P. GRABHER, J. GROßSCHÄDL et D. PAGE, «Non-deterministic processors: FPGA-based analysis of area, performance and security,» *Proceedings of the 4th Workshop on Embedded Systems Security,* pp. 1-10, 2009.

[52] A. WAKSMAN, «Corrigendum:``A Permutation Network",» *Journal of the ACM (JACM),* vol. 15, n° %12, p. 340, 1968.

[53] T. KATASHITA, Y. HORI, H. SAKANE et A. SATOH, «Side-channel attack standard evaluation board SASEBO-W for smartcard testing,» *Power,* vol. 3, n° %12012, p. 400, 2012.

[54] D. COUROUSSÉ, T. BARRY, B. ROBISSON, P. JAILLON, O. POTIN et J.-L. LANET, «Runtime code polymorphism as a protection against side channel attacks,» *IFIP*

International Conference on Information Security Theory and Practice, pp. 136-152, 2016.

[55] N. VEYRAT-CHARVILLON, M. MEDWED, S. KERCKHOF et F.-X. STANDAERT, «Shuffling against side-channel attacks: A comprehensive study with cautionary note,» *International Conference on the Theory and Application of Cryptology and Information Security,* pp. 740-757, 2012.

[56] B. POETTERING, «AVRAES: The AES block cipher on AVR controllers,» [En ligne]. Available: http://point-at-infinity.org/avraes/.

[57] S. CHARI, J. R. RAO et P. ROHATGI, «Template attacks,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 13-28, 2002.

[58] T. PLOS, M. HUTTER et M. FELDHOFER, «On comparing side-channel preprocessing techniques for attacking RFID devices,» *International Workshop on Information Security Applications,* pp. 163-177, 2009.

[59] C. H. GEBOTYS, S. HO et C. C. TIU, «EM analysis of rijndael and ECC on a wireless java-based PDA,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 250-264, 2005.

[60] D. AGRAWAL, B. ARCHAMBEAULT, J. R. RAO et P. ROHATGI, «The EM side—channel,» *International workshop on cryptographic hardware and embedded systems,* pp. 29-45, 2002.

[61] S. CRANE, A. HOMESCU, S. BRUNTHALER, P. LARSEN et F. MICHAEL, « Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity,» *NDSS,* pp. 8-11, 2015.

[62] E. TROMER, D. A. OSVIK et A. SHAMIR, «Efficient cache attacks on AES, and countermeasures,» *Journal of Cryptology,* vol. 23, n° %11, pp. 37-71, 2010.

[63] D. MAY, H. L. MULLER et N. P. SMART, «Random register renaming to foil DPA,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 28-38, 2001.

[64] T. GÜNEYSU et A. MORADI, «Generic side-channel countermeasures for reconfigurable devices,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 33-48, 2011.

[65] H. GENG, J. L. J. WU, M. C. CHOI et Y. SHI, «Utilizing random noise in cryptography: where is the Tofu?,» *IEEE/ACM International Conference on Computer-Aided Design (ICCAD),* pp. 163-167, 2012.

[66] D. DAS, S. MAITY, S. B. NASIR, S. GHOSH, RAYCHOWDHURY et S. SEN, «High efficiency power side-channel attack immunity using noise injection in attenuated signature domain,» *IEEE International Symposium on Hardware Oriented Security and Trust (HOST),* pp. 62-67, 2017.

[67] S. JERÁBEK, J. SCHMIDT, M. NOVOTNÝ et V. MIŠKOVSKÝ, «Dummy rounds as a DPA countermeasure in hardware,» *21st Euromicro Conference on Digital System Design (DSD),* pp. 523-528, 2018.

[68] A. BOGDANOV, L. R. KNUDSEN, G. LEANDER, C. Paar, A. POSCHMANN et M. ROBSHAW, «PRESENT: An ultra-lightweight block cipher,» *International workshop on cryptographic hardware and embedded systems,* pp. 450-466, 2007.

[69] J. COOPER, E. DEMULDER, G. GOODWILL, J. Jaffe, G. Kenworthy, T. Kouzminov, A. Leiserson, M. Marson, P. Rohatgi et S. Saab, «Test vector leakage assessment (TVLA) methodology in practice,» *International Cryptographic Module Conference,* vol. 20, 2013.

[70] P. MOUCHA, S. JEŘÁBEK et M. NOVOTNÝ, «Novel Dummy Rounds Schemes as a DPA Countermeasure in PRESENT Cipher,» *2020 23rd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS),* pp. 1-4, 2020.

[71] N. MENTENS, B. GIERLICHS et I. VERBAUWHEDE, «Power and fault analysis resistance in hardware through dynamic reconfiguration,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 346-362, 2008.

[72] P. SASDRICH, A. MORADI et O. T. MISCHKE, «Achieving side-channel protection with dynamic logic reconfiguration on modern FPGAs,» *IEEE International Symposium on Hardware Oriented Security and Trust (HOST),* pp. 130-136, 2015.

[73] M. SOCHA et N. MENTENS, «August. Dynamic Logic Reconfiguration Based Side-Channel Protection of AES and Serpent,» *Euromicro Conference on Digital System Design (DSD),* pp. 277-282, 2019.

[74] K. PATRANABIS, D. MUKHOPADHYAY et S. GHOSH, «Shuffling across rounds: A lightweight strategy to counter side-channel attacks.,» *IEEE 34th International Conference on Computer Design (ICCD),* pp. 440-443, 2016.

[75] S. N. DHANUSKODI et D. HOLCOMB, «Enabling Microarchitectural Randomization in Serialized AES Implementations to Mitigate Side Channel Susceptibility,» *In 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI),* pp. 314-319, 2019.

[76] S. N. DHANUSKODI, S. ALLEN et D. HOLCOMB, «Efficient Register Renaming Architectures for 8-bit AES Datapath at 0.55 pJ/bit in 16-nm FinFET,» *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 28, n° %18, pp. 1807-1820, 2020.

[77] I. LEVI, D. BELLIZIA et F. TANDAERT, «Beyond algorithmic noise or how to shuffle parallel implementations?,» *International Journal of Circuit Theory and Applications,* vol. 48, n° %15, pp. 674-695, 2020.

[78] «Spartan-6 FPGA Configurable Logic Block https,» 2010. [En ligne]. Available: https://www.xilinx.com/support/documentation/user_guides/ug384.pdf.

[79] C. DE CANNIERE et B. PRENEEL, «Trivium specifications,» *eSTREAM, ECRYPT Stream Cipher Project,* p. 2005.

[80] L. R. GOKE et G. LIPOVSKI, «Banyan networks for partitioning multiprocessor systems,» *Proceedings of the 1st annual symposium on Computer architecture,* pp. 21-28, 1973.

[81] D. H. LAWRIE, «Access and alignment of data in an array processor,» *IEEE Transactions on Computers,* vol. 100, n° %112, pp. 1145-1155, 1975.

[82] J. H. PATEL, «Performance of processor-memory interconnections for multiprocessors,» *IEEE Transactions on Computers,* n° %110, pp. 771-780, 1981.

[83] P. MC III, «The indirect binary n-cube microprocessor array.,» *IEEE Transactions on Computers,* n° %15, pp. 458-473, 1977.

[84] C. CLOS, «A study of non-blocking switching networks,» *System Technical Journal,* vol. 32, n° %12, pp. 406-424, 1953.

[85] V. E. BENEŠ, «Optimal rearrangeable multistage connecting networks,» *Bell system technical journal,* vol. 43, n° %14, pp. 1641-1656, 1964.

[86] K. GAJ, «GMU Source Code,» 2016. [En ligne]. Available: https://cryptography.gmu.edu/athena/index.php?id=CAESAR_source_codes.

[87] C. CLAVIER, J.-S. CORON et N. DABBOUS, «Differential power analysis in the presence of hardware countermeasures,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 252-263, 2000.

[88]

[89] J. DI-BATTISTA, J.-C. COURREGE, B. ROUZEYRE, T. Lionel et P. PERDU, «When failure analysis meets side-channel attacks,» *In International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 188-202, 2010, August.

[90] L. G. S. D. J.-L. M. SAUVAGE et M. NASSER, «Successful Attack on an FPGA-based Automatically Placed and Routed WDDL+ Crypto Processo,» 2008.

[91] B. Metodiev, «Online Channels Account for Record 28 Percent of Global Smartphone Sales in 2020,» 2020. [En ligne]. Available: https://news.strategyanalytics.com/press-releases/press-release-details/2020/Strategy-Analytics-Online-Channels-Account-for-Record-28-Percent-of-Global-Smartphone-Sales-in-2020/default.aspx.

[92] C. E. SHANNON, «Communication theory of secrecy systems,» *The Bell system technical journal,* vol. 28, n° %14, pp. 656-715, 1949.

[93] J. BORGHOFF, A. G. T. CANTEAUT, E. B. KAVUM, M. KNEZEVIC, L. R. KNUDSEN, G. LEANDER et V. NIKOV, «PRINCE–aa low-latency block cipher for pervasive

computing applications,» *In International conference on the theory and application of cryptology and information security,* pp. 208-225, 2012.

[94] B. ROBISSON et P. MANET, «Differential behavioral analysis,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 413-426, 2007.

[95] M. RIVAIN, E. PROUFF et J. DOGET, «Higher-order masking and shuffling for software implementations of block ciphers,» *International Workshop on Cryptographic Hardware and Embedded Systems,* pp. 171-188, 2001.

[96] R. HOUSLEY, «Using advanced encryption standard (aes) counter mode with ipsec encapsulating security payload (esp),» *RFC 3686,* 2004.

[97] G. HARCHA, V. LAPÔTRE, C. CHAVET et P. COUSSY, «Toward Secured IoT Devices: A Shuffled 8-Bit AES Hardware Implementation,» *2020 IEEE International Symposium on Circuits and Systems (ISCAS),* pp. 1-4, 2020.