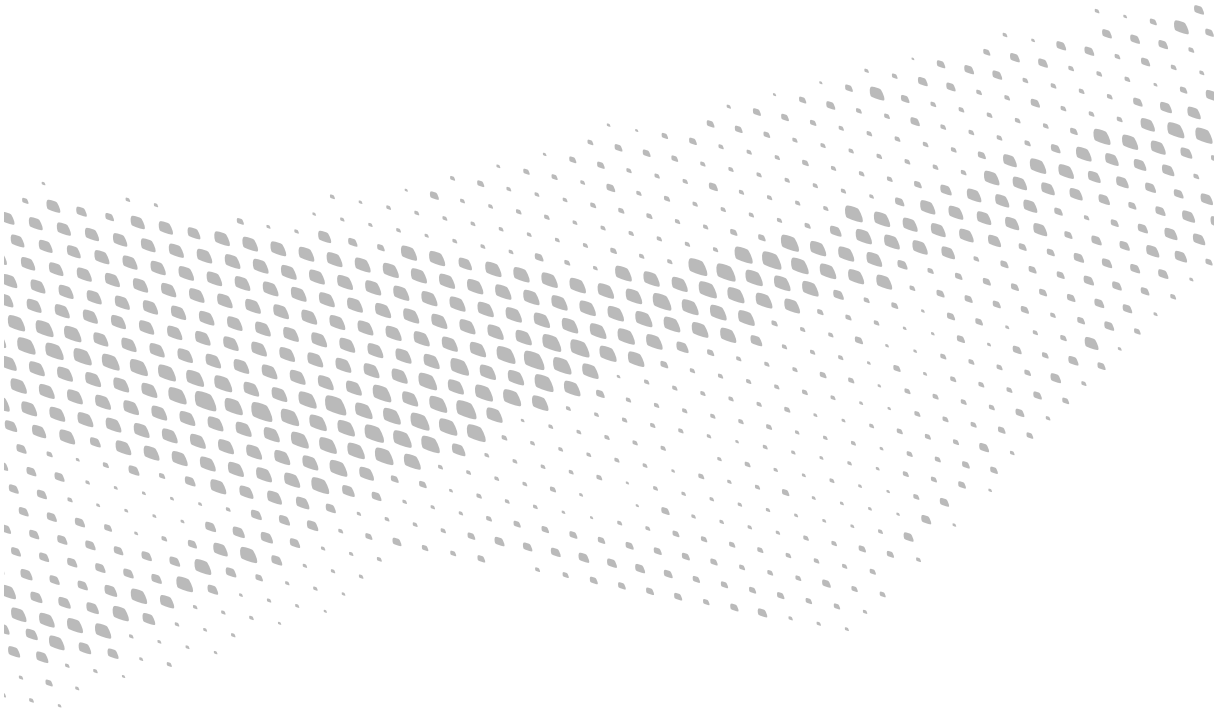


THE CASCADE : résultats finaux d'implémentation

Éditeur	Francq Julien
Reference	Livrable projet ANR THE CASCADE
Date	15/01/2018

Historique

Version	Date	Auteur(s)	Modifications
0.0	01/09/2017	JFR	Transmission du template
0.1	01/10/2017	BDR	Parties incluses
0.2	17/10/2017	JFR	Revue
1.0	18/10/2017	JFR	Parties incluses
1.1	10/11/2017	JFR	Insertion des acronymes, du logo groupé des partenaires et relecture globale
1.2	21/11/2017	JFR	Revue de la fin du livrable (à partir du chapitre 8)
1.3	24/11/2017	JFR	Envoi d'une première version draft pour revue
2.1	06/12/2017	JFR	Prise en compte des remarques des partenaires, et renvoi de la nouvelle version pour revue
2.2	18/12/2017	JFR	Prise en compte des dernières remarques suite au dernier meeting projet
3.1	15/01/2018	JFR	Livraison de la version finale à l'ANR

Glossaire (dans l'ordre alphabétique)

ASIC (*Application-Specific Integrated Circuit*) : circuit intégré propre à une application
AES (*Advanced Encryption Standard*) : standard de chiffrement avancé
CFB (*Cipher Feedback Mode*) : mode de chiffrement à rétroaction
CMOS (*Complementary Metal Oxyde Silicium*) : logique complémentaire pour semiconducteurs métal-oxyde
CPA (*Correlation-Power Analysis*) : analyse par corrélation de courant électrique
CRT (*Chinese Remainder Theorem*) : théorème des restes chinois
CTR (*Counter-Mode*) : mode compteur
DES (*Data-Encryption Standard*) : standard de chiffrement de données
DPA (*Differential Power Analysis*) : analyse différentielle de la consommation électrique
EEPROM (*Electrically-Erasable Programmable Read-Only Memory*) : mémoire morte effaçable électriquement et programmable
FPGA (*Field Programmable Gate Array*) : réseau de portes programmables
GE (*Gate Equivalent*) : porte équivalente
(G)LUT (*(Global) Look-Up Table*) : table (globale) de correspondance
HD (*Hamming Distance*) : distance de Hamming
HOMLUT (*Higher-Order Masking of Look-Up Tables*) : masquage grand ordre de tables de correspondance
HW (*Hamming Weight*) : poids de Hamming
ICS (*Industrial Control Systems*) : systèmes de contrôle industriel
IV (*Initialization Vector*) : vecteur d'initialisation
LFSR (*Linear Feedback Shift-Register*) : registre à décalage à rétroaction linéaire
LPV (*Linear Parameter-Varying*) : linéaire à paramètres variants
MAC (*Media Access Control*) : couche physique
NLFSR (*Non-Linear Feedback Shift-Register*) : registre à décalage à rétroaction non-linéaire
RAM (*Random Access Memory*) : mémoire vive
RC4 (*Rivest Cipher 4*) : chiffrement de Ronald Rivest numéro 4
RSA (*Rivest Shamir Adleman*) : primitive de chiffrement asymétrique
SCA (*Side-Channel Attack*) : attaque par canaux auxiliaires
SCADA (*Supervisory Control and Data Acquisition*) : système de contrôle de supervision et d'acquisition de données
SISO (*Single Input Single Output*) : une seule entrée-une seule sortie
SPA (*Simple-Power Analysis*) : analyse simple de la consommation électrique
SPN (*Substitution-Permutation Network*) : réseau de substitution permutation
SRAM (*Static Random Access Memory*) : mémoire-vive statique
SSC (*Self-Synchronizing Cipher*) : chiffreur auto-synchronisant
SSSC (*Self-Synchronizing Stream Cipher*) : chiffreur par flot auto-synchronisant
TERO (*Transition Effect Ring Oscillator*) : oscillateur en anneaux à régime oscillant transitoire
TI (*Threshold Implementation*) : schéma de masquage basé sur un partage de secrets
UV (*Ultraviolet*) : ultraviolet
VHDL (*Very high speed integrated circuits HDL*) : langage de description matérielle de circuits intégrés à très hautes vitesses

Table des matières

Table des figures

1 Introduction générale

Le projet THE CASCADE a conduit, suite à une étude approfondie et collaborative, à des spécifications d'un algorithme auto-synchronisant innovant. Tout au long de ce processus de spécification, des discussions entre partenaires ont permis d'aboutir à des résultats d'implémentation optimaux et adaptés aux plate-formes de prototypage du projet.

Ces plate-formes ont été utilisées afin de démontrer les fonctionnalités de l'algorithme finalement choisi. La pertinence de ces démonstrations ont été validées lors des différents meetings de consortium.

Les implémentations d'algorithmes cryptographiques peuvent être le sujet d'attaques physiques sur composant afin de récupérer la clé secrète embarquée dans ce dernier. Ces attaques peuvent être de deux ordres : passives ou actives. Pour ce qui concerne les attaques passives, celles dites par "canaux auxiliaires" (*Side-Channel Attacks*, SCA) consistent à utiliser une ou plusieurs caractéristiques du circuit monitoré (sa consommation électrique, son rayonnement électromagnétique, son temps de calcul, *etc.*) pour récupérer la clé cryptographique secrète. C'est un monitoring passif ici : le circuit n'est pas perturbé par l'attaquant au contraire des attaques dites "actives". Dans ce dernier type d'attaques, l'attaquant agit directement sur le circuit via différentes méthodes possibles (par un effet photoélectrique injecté par un laser, par une émission électromagnétique, *etc.*) afin de sortir le composant de ses conditions de fonctionnement nominales, et pouvoir ainsi injecter des fautes pendant un calcul cryptographique. Ces fautes, si elles sont correctement injectées, peuvent amener de l'information sur la clé secrète utilisée dans les textes chiffrés fautés récupérés par l'attaquant.

Dans THE CASCADE, nous avons étudié plusieurs aspects des attaques passives et actives.
Pour ce qui concerne les attaques passives, nous avons d'une part travaillé sur une nouvelle approche d'attaque par analyse de corrélation (*Correlation Power Analysis*, CPA) n'utilisant pas le coefficient de Pearson comme indiqué dans l'article séminal de Brier *et al.*, mais une approche spectrale instanciant une analyse de Fourier de nature plus algébrique. Ce résultat a une portée qui dépasse le simple cadre des algorithmes auto-synchronisants, et touche l'ensemble des algorithmes cryptographiques symétriques. De plus, nous avons évalué le coût d'implémentation des contre-mesures aux attaques par canaux auxiliaires. Ceci est très utile afin de pouvoir jauger les ressources nécessaires à une implémentation susceptible d'être industrialisée et de passer des certifications de sécurité.
Pour ce qui concerne les attaques actives, nous avons dans un premier temps étudié les conditions nécessaires pour pouvoir réaliser une attaque en fautes sur un exemple jouet de notre algorithme final (de plus petite dimension). Cette pré-étude nous a par la suite permis dans une démarche constructive d'implémenter une version finale d'algorithme intrinsèquement plus difficile à attaquer par des fautes.

Le plan de ce livrable est le suivant. Dans la section 2, des rappels théoriques sur les chiffrements auto-synchronisants seront donnés afin de mieux comprendre les autres parties. Dans la section 3, les spécifications finales de notre algorithme seront fournies, avec des justifications sur sa conception. Dans la section 4, nous illustrerons nos travaux par le descriptif de nos démonstrations. Dans la section

5, nous décrirons les implémentations matérielles réalisées ainsi que les résultats d'implémentation associés. Dans la section 6, notre "CPA spectrale" sera détaillée. Dans la section 7, nous décrirons nos implémentations matérielles protégées contre les SCAs. Dans la section 8, nous donnerons les résultats de notre étude de la sécurité de notre algorithme vis-à-vis des attaques par injection de fautes sur une version réduite de notre algorithme et sur sa version finale. La section 9 conclura ce livrable en fournissant notamment un résumé des travaux réalisés et des pistes de recherche futures.

2 Théorie pour la construction de chiffreurs par flot auto-synchronisants

2.1 Introduction du chapitre

Cette partie aborde les notions et concepts de la théorie du contrôle qui sont utiles pour la construction des chiffreurs par flot auto-synchronisants (*Self-Synchronizing Stream Ciphers*, SSSCs). Il illustre l'intérêt potentiel des systèmes dynamiques linéaires à paramètres variants (*Linear Parameter-Varying*, LPV) plats pour répondre à des problématiques en cryptographie. En effet, ces systèmes LPV sont des systèmes linéaires dont la représentation d'état dépend d'un vecteur de paramètres qui varient dans le temps. Ils sont utilisés depuis bien des années en théorie du contrôle [1, 2, 3, 4, 5, 6] et dans le domaine de l'observation ou du filtrage [7, 8, 9, 10]. Les modèles LPV peuvent être par exemple instanciés pour construire des contrôleurs programmables utilisés par exemple dans le contrôle de l'espace aérien [11] ou de véhicules [12]. En outre, ces modèles LPV peuvent représenter des systèmes non linéaires sous certaines conditions bien définies. Ceci a fait l'objet d'études dans [13] ou dans [14].

2.2 Systèmes LPV

Un système linéaire à paramètres variants à une seule entrée et une seule sortie (*Single Input Single Output*, SISO) en anglais noté Σ_ρ et défini sur un corps fini $\mathbb{F}$ est décrit par la représentation d'espace d'état suivant :

$$\Sigma_\rho : \begin{cases} x_{k+1} = A_{\rho(k)} x_k + B_{\rho(k)} u_k \\ y_k = C_{\rho(k)} x_k + D_{\rho(k)} u_k \end{cases} \tag{2.1}$$

où $k \in \mathbb{N}$ correspond au temps discret, $x_k \in \mathbb{F}^n$ correspond au vecteur d'état, $u_k \in \mathbb{F}$ correspond à l'entrée, $y_k \in \mathbb{F}$ correspond à la sortie. Les matrices $A \in \mathbb{F}, B \in \mathbb{F}^{n\times 1}$, $C \in \mathbb{F}^{1\times n}$ et $D \in \mathbb{F}^{1\times 1}$ correspondent respectivement à la matrice dynamique, la matrice d'entrée, la matrice de sortie et la matrice de transfert direct. Comme on peut donc le voir dans l'Equation (2.1) caractéristique d'un système LPV, le vecteur d'état dépend linéairement de l'entrée et de la sortie du système. Cependant, les matrices A, B, C et D dépendent d'un vecteur de paramètres $\rho(k) = \left[\rho^1(k), \rho^2(k), ..., \rho^{L_\rho}(k) \right] \in \mathbb{F}^{L_\rho}$ qui confèrent une dynamique non linéaire au système. Ici L_ρ désigne le nombre total de coefficients (éventuellement variables) non nuls des matrices. En effet, les paramètres variants $\rho^i(k)$ sont définis comme des fonctions non linéaires φ_i en la sortie y_k : $\rho^i(k) = \varphi_i(y_k, y_{k-1}, \cdots)$.

Une fois la définition des systèmes dynamiques LPV établie, on va établir ensuite une correspondance avec les SSSCs à travers la notion de platitude que l'on définit ci-après. On définit la séquence $\{\rho(k), \rho(k+1), \ldots\}$ de paramètres à temps variant, avec k un entier positif, et on appelle cette séquence une "réalisation". La définition d'un système LPV est donc donnée d'un point de vue générique c'est-à-dire pour presque toute réalisation.

2.2.1 Platitude

Définition 2.2.1. *On dit que le système* (2.1) *est génériquement plat, si pour presque toute réalisation ρ, il existe une variable y_k, appelée sortie plate, telle qu'on puisse exprimer toute variable du système comme fonction des itérés antérieurs et ultérieurs de cette sortie plate. En d'autres termes, il existe deux fonctions $\mathcal{F}_\rho$ et $\mathcal{G}_\rho$ paramétrisées par ρ telles que :*

$$\begin{cases} x_k &= \mathcal{F}_\rho\Big(y_{k+k_\mathcal{F}}, \ldots, y_{k+k'_\mathcal{F}}\Big) \\ u_k &= \mathcal{G}_\rho\Big(y_{k+k_\mathcal{G}}, \ldots, y_{k+k'_\mathcal{G}}\Big) \end{cases} \tag{2.2}$$

avec $k_\mathcal{F}$, $k'_\mathcal{F}$, $k_\mathcal{G}$ et $k'_\mathcal{G}$ des valeurs entières de $\mathbb{Z}$.

La platitude est cependant une notion importante qui admet divers et variés domaines d'application comme la planification de trajectoires [15, 16], le contrôle prédictif ou encore la gestion de contraintes [17, 18, 19]. Dans cette thèse, on va s'intéresser, au-delà d'une simple vérification des sorties plates d'un système LPV, à la construction des systèmes LPV qui admettent des sorties plates. Ceci revient donc d'après la définition 2.2.1, à construire des systèmes LPV dont les variables s'expriment comme fonction de la sortie.

2.2.2 Caractérisation de la platitude

Il existe plusieurs approches qui permettent de caractériser les sorties plates d'un système LPV. On peut citer :

l'approche directe : elle consiste à exprimer par itérations successives les variables du système comme fonction de la sortie. Il en résulte cependant que cette approche est onéreuse même pour des systèmes de petite dimension et devient presque impossible à adopter pour des systèmes de grande dimension.

l'approche basée sur le système inverse : il s'agit d'une approche qui utilise, lorsqu'il existe, le *système inverse à gauche* du système LPV (2.1). Ce système inverse admet dans ce cas une matrice dynamique notée P et qui s'exprime en fonction des matrices A, B, C et D du système LPV.

Une caractérisation de la sortie plate basée sur le système inverse a été proposée pour les systèmes linéaires switchés dans [20]. L'extension de cette caractérisation aux systèmes LPV est immédiate, en considérant la fonction de commutation comme une fonction à valeurs dans

un ensemble indéterminé au lieu d'un ensemble fini. Cette caractérisation est basée sur la notion de degré relatif d'un système dynamique [21] que l'on rappelle ci-après :

Définition 2.2.2. *Le degré relatif d'un système dynamique est le nombre minimal r d'itérations à partir duquel la sortie y_{k+r} à l'instant $k+r$ est influencée par l'entrée u_k.*

Dans le cas d'un système LPV (2.1), on montre que si le système admet un degré relatif r, alors, pour toute réalisation ρ, la sortie s'exprime par :

$$y_{k+r} = C_{\rho(k+r)} \prod_{l=k+r-1}^{k} A_{\rho(l)} x_k + \mathcal{T}^{r,0}_{\rho(k)} u_k \tag{2.3}$$

où $\mathcal{T}^{r,0}_{\rho(k)}$ est une quantité définie suivant les valeurs de r par :

1. si $r = 0$: $\mathcal{T}^{0,0}_{\rho(k)} \neq 0$ pour tout entier relatif k
2. sinon si $r < \infty$, alors r est le plus petit entier s tel que :

$$\begin{array}{l} \mathcal{T}^{i,j}_{\rho(k)} = 0 \quad \text{pour } i = 0, \ldots, s-1 \text{ et } j = 0, \ldots, \ i, \\ \mathcal{T}^{s,0}_{\rho(k)} \neq 0 \end{array} \tag{2.4}$$

pour tout entier relatif $k \geq 0$ et pour toute réalisation ρ.

On a donc le théorème suivant qui permet de caractériser la sortie plate d'un système LPV (2.1).

Théorème 2.2.1 ([22])**.** *Si le système LPV (2.1) admet un degré relatif r, alors y_k est une sortie plate si et seulement si il existe un entier positif K tel que pour tout entier relatif $k \geq 0$ et pour toute séquence $\{\rho(k), \ldots, \rho(k+K-1+r)\} \in \Theta^{r+K}$, la condition suivante est vérifiée :*

$$P_{\rho(k+K-1:k+K-1+r)} P_{\rho(k+K-2:k+K-2+r)} \cdots P_{\rho(k:k+r)} = 0 \tag{2.5}$$

avec

$$P_{\rho(k:k+r)} = A_{\rho(k)} - B_{\rho(k)} (\mathcal{T}^{r,0}_{\rho(k)})^{-1} C_{\rho(k+r)} \prod_{l=k+r-1}^{k} A_{\rho(l)} \tag{2.6}$$

La preuve de ce théorème est constructive et permet en outre de montrer que l'état interne x_k ainsi que l'entrée u_k du système LPV (2.1) s'expriment en fonction des itérés de la sortie y_k par :

$$x_k = \sum_{i=0}^{K-1} \left[\prod_{j=1}^{i} P_{\rho(k-j:k-j+r)} \right] (\mathcal{T}^{r,0}_{\rho(k-1-i)})^{-1} B_{\rho(k-1-i)} y_{k-1-i+r} \tag{2.7}$$

et

$$\begin{array}{rcl} u_k & = & (\mathcal{T}^{r,0}_{\rho(k)})^{-1} y_{k+r} - (\mathcal{T}^{r,0}_{\rho(k)})^{-1} C_{\rho(k+r)} \prod_{l=k+r-1}^{k} A_{\rho(l)} \\ & & \cdot \sum_{i=0}^{K-1} \left[\prod_{j=0}^{i-1} P_{\rho(k-j-1:k-j-1+r)} \right] (\mathcal{T}^{r,0}_{\rho(k-1-i)})^{-1} B_{\rho(k-1-i)} y_{k-1-i+r} \end{array} \tag{2.8}$$

L'intérêt du Théorème 2.2.1 est qu'il offre une expression explicite des fonctions $\mathcal{F}_\rho$ et $\mathcal{G}_\rho$ lorsque y_k est une sortie plate. Cependant, vérifier que y_k est une sortie plate exigerait de vérifier le produit (2.5) pour un nombre infini de réalisations ρ ; ce qui ne peut être réalisé dans la pratique. Pour cela, il sera proposé une approche basée sur les graphes orientés que l'on décrira un peu plus bas.

Problème de la mortalité : il est à noter que l'égalité de l'Equation (2.5) n'est pas nécessairement vérifiée pour tout produit de matrice $P_{\rho(k:k+r)}$ mais seulement pour un nombre de séquences admissibles de telles matrices. Cette égalité est connue de façon plus générale sous le nom de *problème de mortalité* qui est un problème indécidable ([23, 24, 25]). Il a cependant été proposé dans [21] une approche basée sur les semi-groupes nilpotents de matrices vérifiant l'égalité (2.5).

Définition 2.2.3 (Semi-groupe nilpotent)**.** *Un semi-groupe $\mathcal{S}$ est un ensemble avec une loi multiplicative associative interne. On dit qu'un semi-groupe $\mathcal{S}$ qui admet un élément absorbant 0 est nilpotent s'il existe un entier $t \in \mathbb{N}^*$ tel que tout produit de t éléments appartenant à $\mathcal{S}$ est nul. Le plus petit entier t est appelé dans ce cas la classe de nilpotence de $\mathcal{S}$.*
Cette notion de semi-groupe nilpotent est intrinsèquement liée à la notion de *triangularisation simultanée*. En effet, le Théorème de Levitzky [26] dit qu'un ensemble de matrices forme un semi-groupe nilpotent si et seulement si cet ensemble est simultanément triangularisable, autrement dit si et seulement si cet ensemble admet une base commune de triangularisation.
On a donc le théorème suivant qui caractérise une sortie plate d'un système LPV.

Théorème 2.2.2 ([21])**.** *Si la matrice $P_{\rho(k:k+r)}$ est triangularisable indépendemment de ρ, alors y_k est une sortie plate du système* (2.1).
Cette dernière approche offre une complexité polynomiale, lorsque l'ensemble de matrices est donné, pour vérifier si une sortie du système est plate ou non. L'inconvénient est qu'elle n'est pas appropriée pour répondre à notre problématique à savoir la construction de matrices qui doivent vérifier le Théorème 2.2.2.

Pour toutes les raisons énumérées ci-dessus, il sera proposé une approche complémentaire basée sur les graphes orientés qui fournit des conditions nécessaires et suffisantes afin de caractériser les sorties plates d'un système LPV et plus encore qui fournit un moyen de construction systématique de systèmes LPV qui admettent une sortie plate.

2.2.3 Approche basée sur les graphes orientés

Cette approche considère le système linéaire structuré associé au système LPV (2.1) et qui est décrit par le système d'équations :

$$\Sigma_\Lambda : x_{k+1} = I_A x_k + I_B u_k \tag{2.9}$$

Ce système structuré exprime une forme simplifiée du système LPV. Ceci permet de traduire les variables de ce dernier en termes de sommets d'un graphe orienté noté par $\mathcal{G}(\Sigma_\Lambda)$, les arcs de ce graphe décrivant la dynamique du système LPV. Ainsi, aux matrices $A_{\rho(k)}$ et $B_{\rho(k)}$ du système LPV (2.1), on associe les matrices I_A et I_B du système structuré (2.9) dont les coefficients sont des $'0'$ et des $'1'$; les variables non nulles des matrices du système LPV étant remplacées par des coefficients $'1'$ dans les matrices du système structuré.

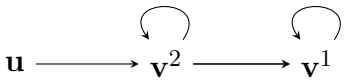


FIGURE 2.1 – Graphe orienté du système linéaire structuré associé au système LPV (2.10).

Si on considère l'exemple suivant de matrices d'un système LPV

$$A_{\rho(k)} = \begin{pmatrix} 1 & \rho^1(k) \\ 0 & \rho^2(k) \end{pmatrix}, B_{\rho(k)} = \begin{pmatrix} 0 \\ \rho^3(k) \end{pmatrix} C_{\rho(k)} = \begin{pmatrix} 1 & 0 \end{pmatrix} D_{\rho(k)} = 0. \tag{2.10}$$

alors les matrices du système structuré sont données par

$$I_A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, I_B = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Le graphe orienté associé au système est donné par la Figure 2.1.

On note par $\mathbf{v^F}$ et $\mathbf{u}$ les sommets du graphe orienté $\mathcal{G}(\Sigma_\Lambda)$ qui sont associés respectivement à la sortie (supposée plate) et l'entrée du système LPV (2.1) ; et par $\mathbf{X}$ l'ensemble des sommets de $\mathcal{G}(\Sigma_\Lambda)$ associés aux composantes de l'état interne x du système LPV. À partir de ce graphe orienté associé au système LPV (2.1), on établit trois conditions nécessaires et suffisantes données par le théorème suivant :

Théorème 2.2.3 ([27]). *Considérons le système linéaire structuré Σ_Λ décrit par (2.9). La sortie y^F associée aux sommets $\mathbf{v^F} \in \mathbf{X} \cup \{\mathbf{u}\}$ est génériquement une sortie plate si et seulement si les trois conditions suivantes sont vérifiées dans le graphe $\mathcal{G}(\Sigma_\Lambda)$ associé :*
C0. *$\mathbf{v^F}$ est un successeur de $\mathbf{u}$ autrement dit, il existe un chemin dans le graphe $\mathcal{G}(\Sigma_\Lambda)$ qui relie l'entrée à la sortie ;*
C1. *tous les chemins simples[1] $\{\mathbf{u}\}$-$\{\mathbf{v^F}\}$ reliant les deux sommets, associés à l'entrée et à la sortie, sont de même longueur $\ell(\mathbf{u}, \mathbf{v^F})$;*
C2. *tous les cycles du graphe couvrent au moins un élément de l'ensemble des sommets essentiels[2] $V_{ess}(\{\mathbf{u}\}, \{\mathbf{v^F}\})$.*

Ainsi l'approche graphe-orienté, à travers ces trois conditions, permet de caractériser de façon plus aisée avec une complexité polynomiale les sorties plates du système LPV (2.1). Et en se basant sur ces conditions, on peut aisément construire des schémas ou structures de graphes qui donnent des systèmes LPV plats, ce que ne permettait pas l'approche algébrique (Théorème 2.2.1). De plus l'approche graphe-orienté fournit explicitement les valeurs du degré relatif (Définition 2.2.2) et de l'entier K du Théorème 2.2.1. Ces valeurs sont données par les deux propositions suivantes :

Proposition 2.2.1 ([22]). *Le degré relatif r du système (2.1) est égal à $\ell(\mathbf{u}, \mathbf{v^F})$ dans le graphe orienté $\mathcal{G}(\Sigma_\Lambda)$.*

Proposition 2.2.2 ([22]). *Si y_k^F est une sortie plate, alors l'entier K du Théorème 2.2.1 est génériquement égal à la longueur maximale de tous les chemins simples $\{\mathbf{u}\}$-$\mathbf{X}$ du graphe orienté $\mathcal{G}(\Sigma_\Lambda)$.*

1. Un chemin simple est un chemin qui passe une et une seule fois par les sommets qu'il couvre.
2. Un sommet essentiel de l'entrée et de la sortie est un chemin qui appartient à l'intersection de tous les chemins reliant l'entrée à la sortie.

En combinant donc les deux approches algébrique et graphe-orienté, on obtient une caractérisation complète des sorties plates du système LPV (2.1) avec des expressions explicites des variables x_k et u_k comme données par les Equations (2.7) et (2.8).

2.3 Structures des chiffreurs autosynchronisants

Nous sommes donc en mesure, grâce aux idées qui ont été développées jusqu'ici, d'établir le lien entre les SSSCs et les systèmes dynamiques LPV plats tout en proposant une construction systématique des SSSCs.

Rappelons avant tout que les SSSCs sont utilisés en cryptographie symétrique comme chiffreurs par flot. De tels chiffreurs sont utilisés dans la transmission de données de façon sécurisée à un débit très élevé et via des appareils électroniques qui sont limités en ressources matérielles. On distingue deux classes principales de chiffreurs par flot :

les chiffreurs dit synchrones (*Synchronous Stream Ciphers*, SSC)

et les SSSCs qui sont donc l'objet d'étude de THE CASCADE.

L'avantage majeur de ces derniers par rapport aux premiers est qu'ils ne nécessitent aucun protocole supplémentaire de resynchronisation entre les entités qui échangent des données sécurisées. L'utilisation des SSSCs est donc d'un intérêt crucial pour la mise en place de communications sécurisées de groupe et permet par exemple à toute entité autorisée de s'insérer dans une communication en cours sans qu'elle n'ait besoin d'initialiser son dispositif de communication avec les autres membres du groupe.

Les SSSCs sont décrits d'une façon générale à travers une forme dite canonique illustrée par la Figure 2.2.

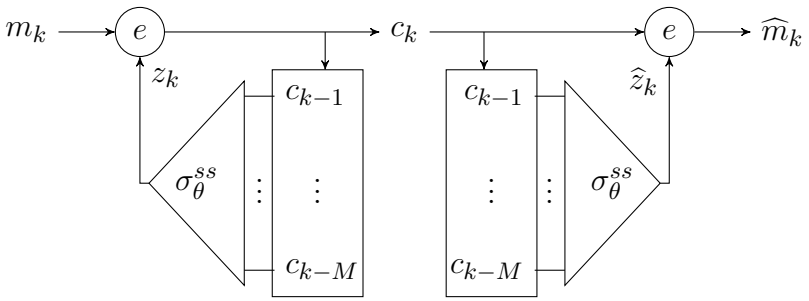


FIGURE 2.2 – Forme canonique d'un SSSC pour $l = 1$.

Les équations de chiffrement et de déchiffrement sont données par les Equations (2.11) et (2.12) suivantes :

$$\begin{cases} z_k = & \sigma_\theta^{ss}(c_{k-l-M+1}, \ldots, c_{k-l}) \\ c_k = & e(z_k, m_k) \end{cases} \tag{2.11}$$

$$\begin{cases} \widehat{z}_k = & \sigma_\theta^{ss}(c_{k-l-M+1}, \ldots, c_{k-l}) \\ \widehat{m}_k = & e(z_k, c_k) \end{cases} \tag{2.12}$$

où θ désigne un paramètre secret [3] partagé par l'émetteur et le récepteur des messages chiffrés échangés, σ_θ^{ss} est appelée *fonction de filtrage* et génère la suite chiffrante notée z_k côté chiffreur et $\widehat{z}_k$ côté déchiffreur. Ces suites chiffrantes dépendent des M précédents chiffrés c_k. On voit donc qu'il suffit au déchiffreur de recevoir les M symboles chiffrés précédant c_k et transmis sans erreur par le chiffreur pour pouvoir déchiffrer correctement c_k.

2.3.1 SSSC et automate à mémoire finie

Les études sur les SSSCs ont été initiées au début des années 90 par les travaux de Maurer [28]. Dans la forme canonique des SSSCs, toute la complexité du schéma réside dans la fonction de filtrage σ_θ^{ss} ce qui n'est pas évident à concevoir. Pour remédier à cela, Maurer a proposé une construction de SSSCs basée sur des *automates à mémoire finie* (illustrée d'une façon généralisée par la Figure 2.3), qui de façon récursive produira la même séquence de suite chiffrante que la forme canonique.

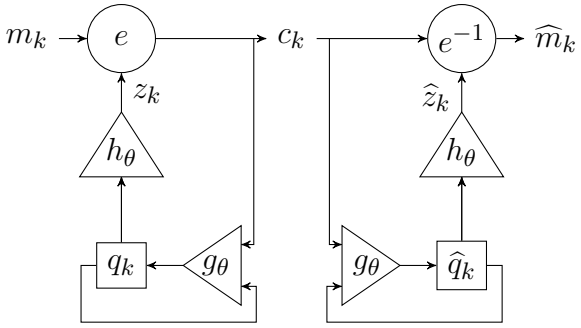


FIGURE 2.3 – Architecture d'un SSSC basée sur un automate à mémoire finie.

L'automate à mémoire finie est caractérisé par le système d'équations suivant :

$$\begin{cases} q_{k+1} = g_\theta(q_k, c_{k+b}) \\ z_{k+b} = h_\theta(q_k) \end{cases} \tag{2.13}$$

où g_θ correspond à la fonction de transition d'état de l'automate, q_k désigne l'état interne et z_k la suite chiffrante générée par la fonction de filtrage h_θ.

Suite aux travaux de Maurer, quelques constructions ont été proposées [29, 30, 31, 32] mais ont été toutes cassées par des travaux de cryptanalyse dans [33, 34, 35, 36, 37]. Il y a cependant un point commun à toutes ces constructions, c'est que la fonction de transition g_θ dans (2.13) est une fonction dite *triangulaire* ou "*T-function*" [38] en anglais.

2.3.2 Nouvelle construction

Un des objectifs du projet est donc de proposer une classe plus générale de fonctions de transition et qui ne sont pas nécessairement triangulaires. Les résultats sur la platitude des systèmes LPV,

3. Autrement dit, en cryptographie, la clé symétrique (secrète).

notamment à travers les Equations (2.7) et (2.8) nous permettent d'ores et déjà d'établir le lien entre les SSSCs et les systèmes LPV plats. On montre qu'on peut utiliser le système LPV direct (2.1) comme chiffreur d'un SSSC et le système inverse du système LPV comme déchiffreur. Ceci est traduit par la proposition suivante :

Proposition 2.3.1 ([39]). *Si le système LPV* (2.1) *admet un degré relatif r et est plat, alors l'automate à mémoire finie donné par les systèmes d'équations suivants définit un SSSC :*

$$\begin{cases} q_{k+1} &= P_{\rho(k:k+r)} q_k + B_{\rho(k)} (\mathcal{T}^{r,0}_{\rho(k)})^{-1} y_{k+r} \\ z_{k+r} &= C_{\rho(k+r)} \prod_{l=k+r-1}^{k} A_{\rho(l)} q_k \end{cases} \qquad \begin{cases} \widehat{q}_{k+1} &= P_{\rho(k:k+r)} \widehat{q}_k + B_{\rho(k)} (\mathcal{T}^{r,0}_{\rho(k)})^{-1} y_{k+r} \\ \widehat{z}_{k+r} &= C_{\rho(k+r)} \prod_{l=k+r-1}^{k} A_{\rho(l)} \widehat{q}_k \end{cases}$$

En conclusion, pour construire notre automate à mémoire finie qui peut être utilisé comme un SSSC, on génère un graphe orienté à partir des Conditions **C0**–**C2** du Théorème 2.2.1. Ensuite, on extrait de ce graphe un système LPV plat et son système inverse. Ces systèmes correpondent à l'automate donné par la Proposition 2.3.1.

2.4 Conclusion du chapitre

Ce chapitre a permis de décrire la théorie nécessaire à la construction générique de chiffreurs par flot auto-synchronisants. Le lien entre théorie du contrôle et cryptographie est maintenant opéré, et conduira au chapitre suivant à détailler une instanciation d'un schéma LPV-SSSC sécurisé d'un point de vue cryptographique.

3 Détails sur les spécifications d'un schéma LPV SSSC

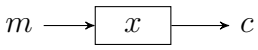
Notations

$\mathbb{F}$	corps fini GF(16)
m	message à chiffrer
n	dimension du système LPV
r	degré relatif du système LPV, correspond au retard de la sortie du système sur l'entrée
ρ	paramètre à temps variant (caractérise les fonctions non-linéaires) du système LPV
$\mathbf{\Phi}_i$	fonctions non-linéaires du système LPV
s	nombre de symboles chiffrés utilisés comme entrées des fonctions non linéaires
m_t	symbole clair (entrée du système) qui est chiffré à l'instant t
$m_t[i]$	i^{eme} bit du symbole m_t
x_{t+1}	état interne du système dynamique (vecteur de dimension n) à l'instant t, côté chiffreur
$x_t[i]$	i^{eme} composante du vecteur x_t
c_{t+1}	cryptogramme (sortie du système) obtenu à l'instant t
$c_t[i]$	i^{eme} bit du symbole c_t
Rc	registre de taille s utilisé pour stocker les cryptogrammes côté chiffreur
Rx	registre de taille n utilisé pour stocker l'état interne du système LPV côté chiffreur
$\widehat{x}_{t+1}$	état interne du système dynamique (vecteur de dimension n) à l'instant t, côté déchiffreur
$\widehat{m}_{t+1}$	symbole clair calculé par le déchiffeur à l'instant t
$\widehat{\text{Rc}}$	registre de taille $\text{s} + r$ utilisé pour stocker les cryptogrammes côté déchiffreur
$\widehat{\text{Rx}}$	registre de taille n utilisé pour stocker l'état interne du système LPV côté déchiffreur

3.1 Introduction du chapitre

Nous pouvons à présent construire un SSSC à partir d'un LPV. Ce chapitre va expliquer comment les paramètres du SSSC sont choisis afin que ce dernier soit sécurisé d'un point de vue cryptographique. L'expert en cryptographie pourra retrouver dans ce chapitre un champ lexical qui lui est familier : chiffreur, déchiffreur, boîte-S (table de substitution ou "*SBox*"), état interne, délai de diffusion, etc. Nous verrons dans ce chapitre que l'essentiel du travail consiste à spécifier le contenu des matrices de chiffrement et de déchiffrement.

3.2 Système dynamique



On construit le schéma SSSC à partir d'un système dynamique LPV (système à paramètres variants) caractérisé par une entrée m, un état interne x et une sortie c. Les équations qui caractérisent le schéma SSSC ainsi obtenu sont données par l'équation générale (3.1) (où r est une valeur qui caractérise le retard de la sortie par rapport à l'entrée du système).

$$\begin{array}{rl} \text{chiffreur :} & \left\{ \begin{array}{rcll} x_{t+1} & = & A_{\rho(t)} \cdot x_t + B \cdot m_t & \\ c_{t+1} & = & C \cdot x_{t+1} = x_{t+1}[r] & \end{array} \right. \quad t \geq 0 \\ \text{déchiffreur :} & \left\{ \begin{array}{rcll} \widehat{m}_t & = & C \cdot A^{\rho(t-1)}_{\rho(t-r)} \cdot \widehat{x}_t + c_t & \\ \widehat{x}_{t+1} & = & P_{\rho(t-r:t)} \cdot \widehat{x}_t + B \cdot \ c_t & \end{array} \right. \quad t \geq 1 \end{array} \tag{3.1}$$

Les additions ainsi que les multiplications de ce système sont définies sur le corps fini $\mathbb{F} = \mathrm{GF}(16)$. Les paramètres du système sont définis sur $\mathbb{F}$ par :

$$\begin{array}{rcl} A_\rho \in \mathbb{F}^{\mathrm{n}\times\mathrm{n}} & \longrightarrow & \text{matrice n} \times \text{n sur } \mathbb{F} \\ P_\rho \in \mathbb{F}^{\mathrm{n}\times\mathrm{n}} & \longrightarrow & \text{matrice n} \times \text{n sur } \mathbb{F} \\ B \in F^{\mathrm{n}\times 1} & \longrightarrow & \text{matrice n} \times 1 \text{ sur } \mathbb{F} \\ C \in \mathbb{F}^{1\times\mathrm{n}} & \longrightarrow & \text{matrice } 1 \times \text{n sur } \mathbb{F} \\ x_t \in \mathbb{F}^{\mathrm{n}} & \longrightarrow & \text{vecteur de dimension n sur } \mathbb{F} \\ m_t, c_t \in \mathbb{F} & & \end{array}$$

où les matrices B et C sont définies par

$$B = {}^t(1, 0, \ldots, 0)\,^1 \qquad C = (0, \ldots, 0, \underbrace{1}_{r}, 0, 0)$$

On notera par $x[i], i = 1, \ldots, \mathrm{n}$ les composantes du vecteur x. Ainsi on aura $x_t = x_t[1], \ldots, x_t[\mathrm{n}]$.

1. le produit $B \cdot m_t$ correspond donc au produit du vecteur B par le scalaire m_t

La notation en indice $\rho(t)$ des matrices $A_{\rho(t)}$ et $P_{\rho(t)}$ indiquent que celles-ci sont paramétrées par un ensemble de fonctions non linéaires (voir (3.7)). Les entrées de ces fonctions sont un nombre s de cryptogrammes précédents.

Les structures des matrices A_ρ et P_ρ (par rapport aux fonctions non linéaires) sont données dans la sous-section 3.2.1. Par définition

$$P_{\rho(t-r:t)} = A_{\rho(t-r)} - B \cdot C \cdot A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}.$$

Autrement dit (compte tenu de l'expression des matrices B et C) :

$$\begin{array}{rcl} P_{\rho(t-r:t)}[0] & = & A_{\rho(t-r)}[0] - \left(A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}\right)[r-1] \\ P_{\rho(t-r:t)}[i] & = & A_{\rho(t-r)}[i], \qquad 1 \leq i \leq n-1 \end{array} \tag{3.2}$$

Par définition, la valeur $A^{\rho(t-1)}_{\rho(t-r)}$ correspond au produit matriciel $\left(A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}\right)$. Ainsi, la matrice $C \cdot A^{\rho(t-1)}_{\rho(t-r)}$ est une matrice ligne qu'on notera par $A^{(r)}_{\rho(t-r)} = \left(A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}\right)[r]$ et qui n'est rien d'autre que la ligne r du produit $A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}$. Elle se déduit des premières lignes de la matrice $P_{\rho(t-r:t)}$ et $A_{\rho(t-r)}$ d'après (3.2).

Pour le chiffrement, pour calculer x_{t+1} on utilise s cryptogrammes $c_t, \ldots, c_{t-(\mathrm{s}-1)}$ comme entrée à la matrice $A_{\rho(t)}$.

Pour le déchiffrement, pour calculer $\widehat{m}_t$ et $\widehat{x}_{t+1}$, on utilise c_t et la séquence $c_{t-i}, \ldots, c_{t-i-(s-1)}$ comme vecteur d'entrée des matrices $A_{\rho(t-i)}$, $1 \leq i \leq r$.

3.2.1 Structures des matrices A_ρ et P_ρ

Pour l'implémentation à réaliser, on considère un système de dimension n $= 40$, $r = 3$ et s $= 1$. Le choix de ces paramètres sera justifié dans ce qui suit.

La structure de la matrice A_ρ est illustrée par un graphe orienté dont les sommets correspondent aux composantes de l'état interne x du système (3.1). A partir de ce graphe, on obtient une matrice (booléenne) notée A qui définit les influences entre les variables x_i.

La matrice A_ρ est obtenue à partir de la matrice A en remplaçant les coefficients 1 de A par des fonctions non-linéaires qui sont notées $\Phi_i, i = 1, \ldots, \mathrm{L}_\rho$, où L_ρ est le nombre de coefficients non nuls de la matrice A [2]. La fonction Φ_i est définie par :

$$\begin{array}{rccc} \Phi_i : & \mathbb{F}^s & \longrightarrow & \mathbb{F} \\ & (c_{t-1}, \ldots, c_{t-s}) & \mapsto & (y_1, y_2, y_3, y_4) \qquad y_i \in \mathbb{F}_2 \end{array}$$

2. Certains coefficients non-nuls ne sont pas toujours remplacés par des boîtes-S : il n'y aura donc au plus que L_ρ fonctions non-linéaires.

Pour réaliser les fonctions Φ_i, on utilise des boîtes-S pour produire des sorties de 4 bits (voir (3.7)). Ainsi à chaque fonction Φ_i qui correspond à un coefficient non nul dans la matrice A_ρ, on associe une boîte-S qui sera notée S_i. La figure 3.1 montre un exemple pour un système de dimension $\mathrm{n} = 7$ où $\mathbf{v}^i$ est le sommet correspondant à la composante $x[i]$ de x.

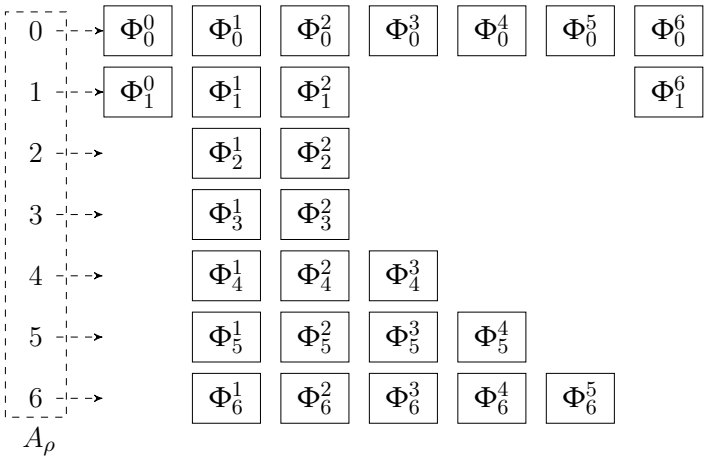


FIGURE 3.1 – Exemple pour un système de dimension $\mathrm{n} = 7$, de degré relatif $\mathrm{r} = 3$ et de nombre de coefficients non nuls $\mathrm{L}_\rho = 27$ dans la matrice A_ρ. La figure à droite décrit les dispositifs $\Phi_i, (i = 1, \ldots, \mathrm{L}_\rho)$ de la matrice A_ρ.

Le tableau de la figure 3.1 donne la disposition des boîtes-S dans A_ρ.

Remarque : La figure 3.1 donne une structure de la matrice A_ρ pour un nombre de nœuds (nombre de coefficients non nuls) $\mathrm{L}_\rho = 27$. Les dispositifs Φ_i représentés ici sont ceux qui correspondent aux coefficients non nuls.

Notre spécification décrit un système de dimension $\mathrm{n} = 40$ défini par l'équation (3.1), avec 119 coefficients non nuls dont 80 boîtes-S paramétrées par des sous clés de 4 bits et 39 coefficients constants qui valent 1.

Système inverse : Tout comme le système de chiffrement qui est caractérisé par la matrice d'état A_ρ, le système inverse qui permet le déchiffrement est lui caractérisé par la matrice P_ρ (3.2). Pour déchiffrer le cryptogramme c_t, on calcule $P_{\rho(t-3:t)}$

$$\begin{array}{rcll} P_{\rho(t-3:t)}[0] & = & A_{\rho(t-3)}[1] - (A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdot A_{\rho(t-3)})[2] & \\ P_{\rho(t-3:t)}[i] & = & A_{\rho(t-3)}[i], & 1 \leq i \leq n-1 \end{array} \tag{3.3}$$

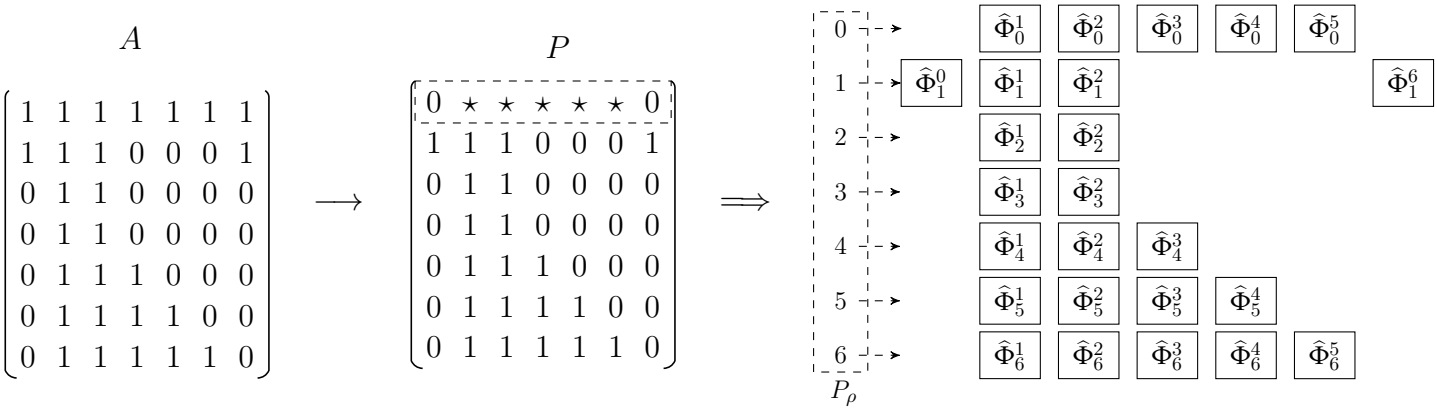


FIGURE 3.2 – Déduction de la matrice P_ρ. La matrice P s'obtient à partir de la matrice A. La première ligne de P_ρ dépend de la matrice ligne $A_\rho^{(r)}$ et de A_ρ. Les autres lignes dépendent uniquement de A_ρ avec un retard sur les cryptogrammes.

Les coefficients de la matrice P_ρ sont les mêmes que ceux de la matrice A_ρ, sauf ceux de la première ligne (cf. (3.4)). Par contre les entrées (séquence de cryptogrammes) de P_ρ ne sont pas les mêmes que les entrées de A_ρ. Ainsi pour déchiffrer c_t, P_ρ prendra en entrée la séquence $c = c_{t-1}, c_{t-2}, c_{t-3}$. Plus précisément on a :

$$\begin{array}{lll} \text{ligne 0} & \left|\begin{array}{rcll} \widehat{\Phi}_0^1(c) & = & \Phi_{39}^1(c_{(t-3)}) + \Phi_1^2(c_{(t-2)}) + 1 & \\ \widehat{\Phi}_0^2(c) & = & \Phi_{39}^2(c_{(t-3)}) + \Phi_1^2(c_{(t-2)}) + 1 & \\ \widehat{\Phi}_0^j(c) & = & \Phi_{39}^j(c_{(t-3)}) & j = 3, 4, \ldots, 38. \\ \widehat{\Phi}_0^0(c) & = & \widehat{\Phi}_0^{39}(c) = 0 & \end{array}\right. \\ \text{lignes 1 à 39} & \left|\widehat{\Phi}_i^j(c) = \Phi_i^j(c_{(t-3)})\right. & \end{array} \tag{3.4}$$

A partir de l'équation (3.4), on déduit l'expression de $A_\rho^{(3)}$ qui prend également en entrée la séquence $c = c_{t-1}, c_{t-2}, c_{t-3}$. Notons par $\tilde{\Phi}_1^j$ les coefficients de $A_\rho^{(3)}$; on a :

$$\begin{array}{rcll} \tilde{\Phi}_0^0(c) & = & \Phi_0^0(c_{(t-3)}) & \\ \tilde{\Phi}_0^j(c) & = & \Phi_0^j(c_{(t-3)}) + \widehat{\Phi}_0^j(c), & j = 1, \ldots, 38 \\ \tilde{\Phi}_0^6(c) & = & \Phi_0^{39}(c_{(t-3)}) & \end{array} \tag{3.5}$$

3.2.2 Paramètre du schéma SSSC et mise à jour de l'état interne

On considère les valeurs suivantes pour les paramètres du schéma :

SYS_DIM $= \mathrm{n} = 40$	$\longrightarrow$	dimension du système (taille de l'état interne)
BUFFER_SIZE $= \mathrm{s} = 1$	$\longrightarrow$	nombre de symboles chiffrés utilisés en entrée des fonctions Φ_i^j
SB_NBR $= 119$	$\longrightarrow$	nombre de noeuds de la matrice d'état A_ρ qui correspond au nombre de coefficients non nuls.
$\mathrm{L}_\rho = 80$	$\longrightarrow$	nombre de sous clés de la matrice d'état A_ρ

Chaque symbole du schéma est constitué de 4 bits. Les opérations d'addition et de multiplication se font dans le corps GF(16). Les paramètres du schéma sont définis comme suit :

- **état interne** : l'état interne du schéma SSSC est constitué à chaque instant t d'un registre Rc de cryptogrammes et d'un registre Rx de n variables $x = (x[0], \ldots, x[\mathrm{n}-1])$ (état interne du système dynamique)
- **entrée** : chaque entrée m_t est un symbole de 4 bits du texte clair à chiffrer
- **sortie** : chaque sortie c_t est un symbole de 4 bits.

Equation pour x **:** $x_{t+1}[0] = \sum_{j=0}^{n-1} \Phi_1^j(c_{t-1}) \cdot x_t[j] + m_t$ et pour $i = 1, \ldots, \mathrm{n}-1$, $x_{t+1}[i] = \sum_{j=0}^{n-1} \Phi_i^j(c_{t-1}) \cdot x_t[j]$. Cette dernière égalité donne l'expression de chaque composante $x[i]$ de l'état interne x à l'instant t.

La mise à jour du vecteur d'état x, côté chiffreur est donc donnée par (cf. 3.1) :

$$\begin{aligned} x_{t+1}[0] &= \textstyle\sum_{j=0}^{\mathrm{n}-1} \Phi_0^j(c(t)) \cdot x_t[j] + m_t \\ x_{t+1}[1] &= \Phi_1^0(c(t)) \cdot x_t[0] + \Phi_1^1(c(t)) \cdot x_t[1] + \Phi_1^2(c(t)) \cdot x_t[2] + \Phi_1^6(c(t)) \cdot x_t[6] \\ x_{t+1}[2] &= \Phi_2^1(c(t)) \cdot x_t[1] + \Phi_2^2(c(t)) \cdot x_t[2] \\ x_{t+1}[i] &= \textstyle\sum_{j=1}^{i-1} \Phi_i^j(c(t)) \cdot x_t[j] \; i = 3, \ldots, \mathrm{n}-1 \end{aligned} \tag{3.6}$$

avec $c(t) = c_t, \ldots, c_{t-(\mathrm{s}-1)}$. On notera simplement par la suite cette séquence par c (cf. sous-section 3.2.4.1 pour l'équation détaillée du chiffrement).

On calcule ensuite le cryptogramme par $c_{t+1} = x_{t+1}[2]$.
Il y a donc un retard de la sortie (cryptogramme) par rapport à l'entrée (le clair). Le cryptogramme correspondant au symbole clair m_t n'est pas c_{t+1} mais plutôt c_{t+3} qui est obtenu après un délai de 3 itérations.

Le registre Rc du chiffreur est constitué de s symboles utilisés comme entrée des fonctions Φ_i^j.

Le déchiffrement est réalisé à partir d'une équation qui est la même que (3.6) sauf pour la première composante $x[1]$ de l'état interne et avec un retard sur le cryptogramme (entrée des fonctions Φ_i^j) [3]. En effet, le registre $\widehat{\mathrm{Rc}}$ du déchiffreur est quant à lui constitué de s symboles auxquels il faut ajouter

3. L'équation détaillée du déchiffrement est donnée en sous-section 3.2.4.2.

r symboles (compte tenu du retard). La taille de $\widehat{\text{Rc}}$ est donc de 4 symboles [4] (resp. 5 symboles) pour $\text{s} = 1$ (resp. $\text{s} = 2$) et est constitué de $c_t, c_{t-1}, \ldots, c_{t-3}$ (resp. $c_t, c_{t-1}, \ldots, c_{t-4}$). La mise à jour de Rc et de $\widehat{\text{Rc}}$ se fait par simple décalage en éliminant le plus ancien symbole. Dans le registre $\widehat{\text{Rc}}$ de taille $\text{s} + r$ cases contenant les cryptogrammes, le cryptogramme le plus récent c_t est stocké à la position 0 tandis que le plus ancien $c_{t-(\text{s}+r-1)}$ est stocké à la position $\text{s} + r - 1$ (cf. Figure 3.3).

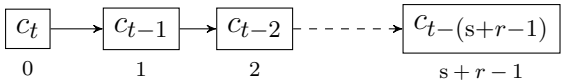


FIGURE 3.3 – Équation pour $c : i = 1, \ldots, \text{s} + r - 1,\ c[i]_{(t+1)} = c[i]_{(t)}$. Le symbole $c[0]$ est remplacé par le dernier symbole de cryptogramme calculé.

Chaque fonction Φ_i^j est définie à partir d'une boîte-S et prend en entrée la séquence de cryptogrammes donnée par $c = c_{t-1}, c_{t-2}, \ldots, c_{t-\text{s}}$. Ainsi $c = c_{t-1}$ (resp. $c = c_{t-1}, c_{t-2}$) pour $\text{s} = 1$ (resp. $\text{s} = 2$). La sortie de Φ_i^j est une séquence de 4 bits. Plus précisément, la fonction Φ_i^j est donnée par :

$$\Phi_i^j(c) = \begin{cases} 0 & \text{si } a_{ij} = 0 \\ \text{S}_i^j(c + \text{SK}_{ij}) & \text{si } a_{ij} \neq 0 \end{cases} \tag{3.7}$$

En réindexant les coefficients de la matrice A_ρ et en considérant uniquement les coefficients non nuls, notés $a_1, \ldots, a_{\text{L}_\rho}$, l'équation (3.7) s'écrit pour $i = 1, \ldots, \text{L}_\rho$:

$$\Phi_i(c) = \text{S}_i(c + \text{SK}_i) \tag{3.8}$$

Chaque S_i est : une boîte-S 4×4 donc une table de 16 éléments qui produit une sortie de 4 bits pour une entrée de 4 bits. Pour ce cas, on choisit la même boîte-S 4×4, qui correspond à la fonction $x \longmapsto \frac{1}{x} + \alpha^2$ sur GF(16) où α est un générateur de GF(16).

On fixe les boîtes-S associées à $\Phi_1^0, \Phi_1^1, \Phi_1^6, \Phi_2^1, \Phi_2^2$ toutes égales à la fonction constante 1 [5].

La structure de Φ_i est donnée par la figure 3.4.

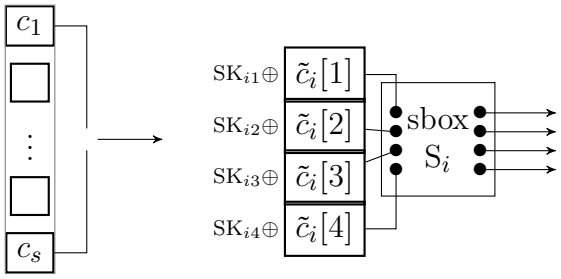


FIGURE 3.4 – Description du dispositif $\Phi_i,\ i = 1, \ldots, \text{L}_\rho$. La sous clé $\text{SK}_i = (\text{SK}_{i1}, \ldots, \text{SK}_{i4})$ associée à Φ_i est constituée de 4 bits avec $\text{s} = 1$.

4. Le nombre de symboles est donné par $(\text{s} + r)$
5. Ce qui explique la différence entre le nombre de coefficients non nuls et le nombre de sous clés.

La boîte-S est donnée par la table 3.1

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
4	5	D	A	9	F	3	2	B	6	8	1	E	0	7	C

TABLE 3.1 – Valeurs de la boîte-S

Remarque 3.2.1. *La taille de la sous clé SK_i dépend uniquement de la taille de l'entrée de la boîte-S S_i et non de la valeur de s. Ainsi pour une boîte-S 4×4 (resp. 8×4), on a des sous clés de 4 bits (resp. 8 bits).*
La figure 3.4 illustre le calcul de la sortie de Φ_i pour une valeur quelconque de s : dans ce cas on sélectionne le nombre de bits nécessaires par rapport à l'entrée de la boîte-S associée à Φ_i.

Par exemple :
- *si* $s = 2 \longrightarrow Rc = c = (c_1, c_2)$
  - *pour des boîtes-S* $4 \times 4 \rightarrow \tilde{c} = (\tilde{c}[1], \tilde{c}[2], \tilde{c}[3], \tilde{c}[4])$
  - *pour des boîtes-S* $8 \times 4 \rightarrow \tilde{c} = (\tilde{c}[1], \ldots, \tilde{c}[8])$
- *si* $s = 3 \longrightarrow Rc = c = (c_1, c_2, c_3)$
  - *pour des boîtes-S* $4 \times 4 \rightarrow \tilde{c} = (\tilde{c}[1], \tilde{c}[2], \tilde{c}[3], \tilde{c}[4])$
  - *pour des boîtes-S* $8 \times 4 \rightarrow \tilde{c} = (\tilde{c}[1], \ldots, \tilde{c}[8])$

3.2.3 Instanciation des paramètres du schéma

Dérivation des sous clés : La taille de l'espace des clés est de l'ordre de la racine carrée du nombre d'états internes possibles. Le nombre d'états internes est défini à partir du déchiffreur soit au total $2^{4(\mathrm{n+s+r})} = 2^{4\times 44}$. La clé maîtresse notée K est donc, dans notre cas, de taille 88 bits. Elle permet de dériver L_ρ sous clés SK_i chacune de taille 4 bits pour des boîtes-S 4×4 (ou 8 bits pour des boîtes-S 8×4) et qui sont associées aux dispositifs Φ_i. Chacune de ces sous-clés SK_i est additionnée bit à bit aux entrées des boîtes-S comme décrit dans la figure 3.4.

La taille des clés ainsi que les *taps* des polynômes de rétroaction du registre à décalage à rétroaction linéaire (*Linear-Feedback Shift Register*, LFSR) sont définis par la table 3.2, et le LFSR résultat est représenté en Figure 3.5.

s	Longueur de la clé (bits)	*Taps*
1	88	77; 79; 80; 88
2	90	85; 87; 88; 90

TABLE 3.2 – Table des valeurs des *taps* des polynômes de rétroaction suivant la taille des clés.

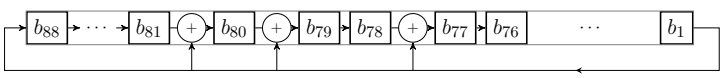


FIGURE 3.5 – LFSR de 88 bits en mode Galois. Le polynôme de rétroaction est défini par $p(x) = x^{88} + x^{80} + x^{79} + x^{77} + 1$.

On réalise le registre à décalage à rétroaction non-linéaire (*Non-Linear Feedback Shift Register*, NLFSR) en rajoutant une boîte-S au LFSR. Les bits d'entrée du LFSR sont définis par les derniers bits (4 bits ou 8 bits) de sortie du LFSR.

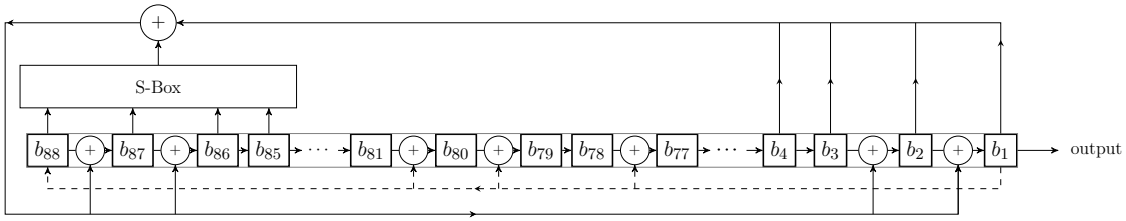


FIGURE 3.6 – NLFSR avec une boîte-S 4×4.

Example d'initialisation du LFSR : Soit une clé de 88 bits, Clé = `ac 2f bf 08 03 9e ed d8 5f 6e fc` (du MSB au LSB). On initialise le registre du LFSR octet par octet. Ainsi on aura, en remarquant que ac = 10101100 (octet de poids fort), 2f = 00101111, ..., et fc = 11111100 (octet de poids faible) :

b_{88}	b_{87}	b_{86}	b_{85}	b_{84}	b_{83}	b_{82}	b_{81}	b_{80}	b_{79}	b_{78}	b_{77}	b_{76}	b_{75}	b_{74}	b_{73}	$\cdots$	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1
1	0	1	0	1	1	0	0	0	0	1	0	1	1	1	1	$\cdots$	1	1	1	1	1	1	0	0

Les 4 bits de sortie de la boîte-S sont mélangés (XOR après un décalage) aux bits du LFSR comme défini par le tableau suivant :

s	Longueur de la clé (bits) \ Boîte-S	4×4
1	88	$1; 2; 86; 87$

TABLE 3.3 – Table des bits d'entrée du LFSR obtenus par addition (XOR) des bits de sortie de la boîte-S. Les bits de sortie de la boîte-S sont additionnés aux positions $1; 2; 86; 87$ du LFSR : le bit de poids fort à la position 87 et le bit de poids faible à la position 1.

3.2.4 Chiffrement et déchiffrement

On décrit les mécanismes de chiffrement et de déchiffrement pour s = 1.

Le déchiffrement est réalisé avec un retard de $r = 3$ itérations sur le clair m_t. La taille du registre Rc (contenant les cryptogrammes côté chiffreur) est s = 1. Celle du registre $\widehat{\text{Rc}}$ (contenant les cryptogrammes côté déchiffreur) est $\text{s} + r = 4$.

Le symbole clair $\widehat{m}_t$ obtenu après déchiffrement de c_t correspond au clair m_{t-3}.
Le déchiffrement de c_t nécessite $c_{t-1}, c_{t-2}, c_{t-3}$ pour calculer $\widehat{m}_t$ et $\widehat{x}_{t+1}$ (cf. Figure 3.2.4).

La section A.1 de l'annexe donne les valeurs des paramètres, des entrées, états internes et sorties.

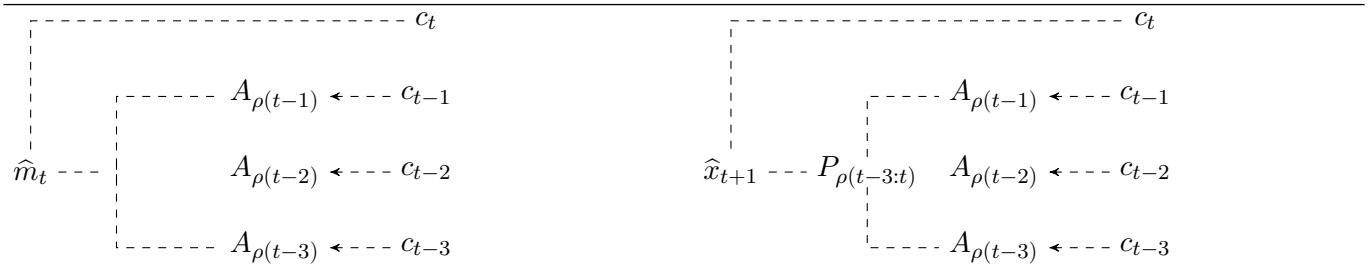


FIGURE 3.7 – Symboles chiffrés nécessaires pour déchiffrer c_t et pour mettre à jour l'état interne $\widehat{x}_{t+1}$.

Les mécanismes de chiffrement et de déchiffrement sont définis ci-après.

3.2.4.1 Méchanisme de chiffrement

Pour chiffrer un message $m = m_0, \ldots, m_{\ell-1}$ de ℓ symboles dans $\mathbb{F}$, on procède comme suit (cf. Figure 3.8) :

1. On choisit aléatoirement un vecteur x_0 de dimension n comme séquence d'initialisation de l'état interne x du système LPV. Cette séquence n'est pas transmise au déchiffreur.
2. On choisit aléatoirement $\mathrm{s} + \mathrm{n} - (r-1)$ symboles pour la *séquence de synchronisation* que l'on concatène au début du message m à chiffrer. Ce qui permet d'obtenir une séquence $m_0, \ldots, m_{\mathrm{s}+n-r+\ell}$.
   On choisit ensuite $r-1$ symboles aléatoires que l'on ajoute à la fin du message. Ces symboles sont nécessaires uniquement pour traiter les $r-1$ derniers symboles clairs du message à chiffrer compte tenu du retard au niveau du chiffrement.
3. On choisit un vecteur d'initialisation (*Initialization Vector*, IV) du schéma SSSC de s symboles $(c_1^\dagger, \ldots, c_s^\dagger)$. Cette séquence est utilisée pour le calcul des s premiers symboles clairs aléatoires.
   Les $\mathrm{n} + \mathrm{s}$ premiers symboles chiffrés proviennent de la séquence de synchronisation et du retard. Ils sont déchiffrés au cours du délai de synchronisation (qui est $\mathrm{n} + \mathrm{s}$). Les symboles chiffrés qui suivent correspondent quant à eux aux chiffrés du message m.
4. Pour chaque symbole clair m_t à l'instant $t \geq 0$,
   (a) on calcule l'état interne suivant $x_{t+1}[i] = \sum_{j=1}^{n_i} \Phi_j(t) \cdot x_t[j]$ en utilisant (3.6)
   (b) on ajoute le symbole clair : $x_{t+1}[1] = x_{t+1}[1] + m_t$
   (c) puis on calcule le symbole chiffré $c_{t+1} = x_{t+1}[3]$.

Pour les schémas retenus, le lecteur intéressé pourra consulter les équations en annexe.

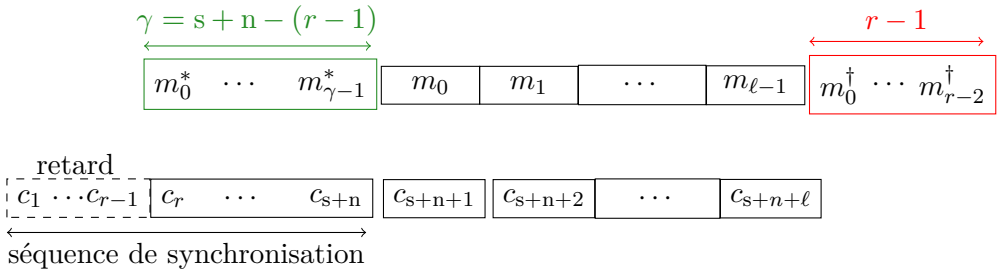


FIGURE 3.8 – Chiffrement d'un message m de longueur ℓ. On rembourre le début de m par $\gamma = \mathrm{s}+\mathrm{n}-(r-1) = n-1$ symboles aléatoires $\left(m_0^*, \ldots, m_{\gamma-1}^*\right)$. Le registre qui permet de stocker les cryptogrammes est initialisé par un IV $\left(c_1^\dagger, \ldots, c_{\mathrm{s}}^\dagger\right)$

avec

$$\begin{aligned} \mathrm{n} &= 40 \\ \mathrm{r} &= 3 \\ \mathrm{s} &= 1 \\ \gamma &= 39 \end{aligned}$$

Pour un message $m = m_0, \ldots, m_{\ell-1}$ de longueur ℓ, le chiffré qui correspond à m_i est $c_{\mathrm{s+n}+i+1},\ i = 0, \ldots, \ell-1$. La taille du message chiffré (qui correspond au message m complété) est $\mathrm{s}+\mathrm{n}+\ell$ où les $\mathrm{s}+\mathrm{n}$ premiers symboles seront utilisés par le déchiffreur pour synchroniser avec le chiffreur.
Le délai de synchronisation est $\mathrm{s}+\mathrm{n}$: dans le cas d'une perte de synchronisation provoquée par une mauvaise transmission ou une altération d'un cryptogramme, il y a resynchronisation au bout de $\mathrm{s}+\mathrm{n}$ itérations.

3.2.4.2 Déchiffrement

Pour déchiffrer le message chiffré de $\mathrm{s}+\mathrm{n}+\ell$ symboles, on procède comme suit (cf. Figure 3.9) :

1. On initialise le registre de cryptogrammes $\widehat{\mathrm{R}}\mathrm{c}$ à $(0, \ldots, 0)$ (on rappelle que la taille de $\widehat{\mathrm{R}}\mathrm{c}$ est de $\mathrm{s}+r = 4$)
2. On initialise le registre de l'état interne $\widehat{x}$ avec n'importe quelle séquence de symboles arbitraires, par exemple un vecteur de dimension n nul : $\widehat{\mathrm{R}}\mathrm{x} = \widehat{x}_1 = (0, \ldots, 0)$.
3. Pour chaque symbole à l'instant $t \geq 1$,
   (a) On calcule le texte clair par : $\widehat{m}_t = \left(A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdot A_{\rho(t-3)}\right)[2] \cdot \widehat{x}_t + c_t$
   (cf. (3.5) pour le calcul de $\left(A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdot A_{\rho(t-3)}\right)[2] = A_\rho^{(3)}$)
   (b) On calcule l'état interne suivant par : $\widehat{x}_{t+1} = P_{\rho(t-3:t)} \cdot \widehat{x}_t + B.c_t$
   (cf (3.3) et (3.4) pour le calcul de $P_{\rho(t-3:t)}$). On a donc :

$$\begin{aligned} \widehat{x}_{t+1}[0] &= \left(P_{\rho(t-3:t)} \cdot \widehat{x}_t\right)[0] + c_t \\ \widehat{x}_{t+1}[i] &= \left(P_{\rho(t-3:t)} \cdot \widehat{x}_t\right)[i] \qquad i = 1, \ldots, 39 \end{aligned}$$

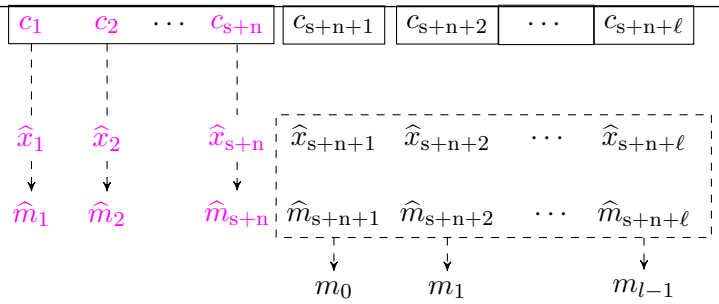


FIGURE 3.9 – Déchiffrement. Les s + n premiers symboles du cryptogramme sont déchiffrés pendant le délai de synchronisation. Le déchiffrement est proprement réalisé à partir de $\widehat{m}_{s+n+1}$. Chaque symbole $\widehat{m}_{s+n+i+1}$ doit correspondre à m_i, $i = 0, \ldots, \ell - 1$.

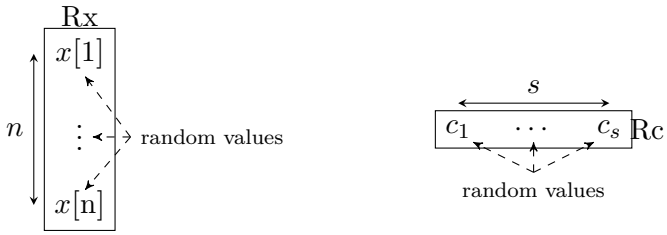


FIGURE 3.10 – Chiffrement : Initialisation du registre Rx de l'état interne x et du registre Rc contenant les cryptogrammes. Toutes les valeurs aléatoires sont des symboles de 4 bits.

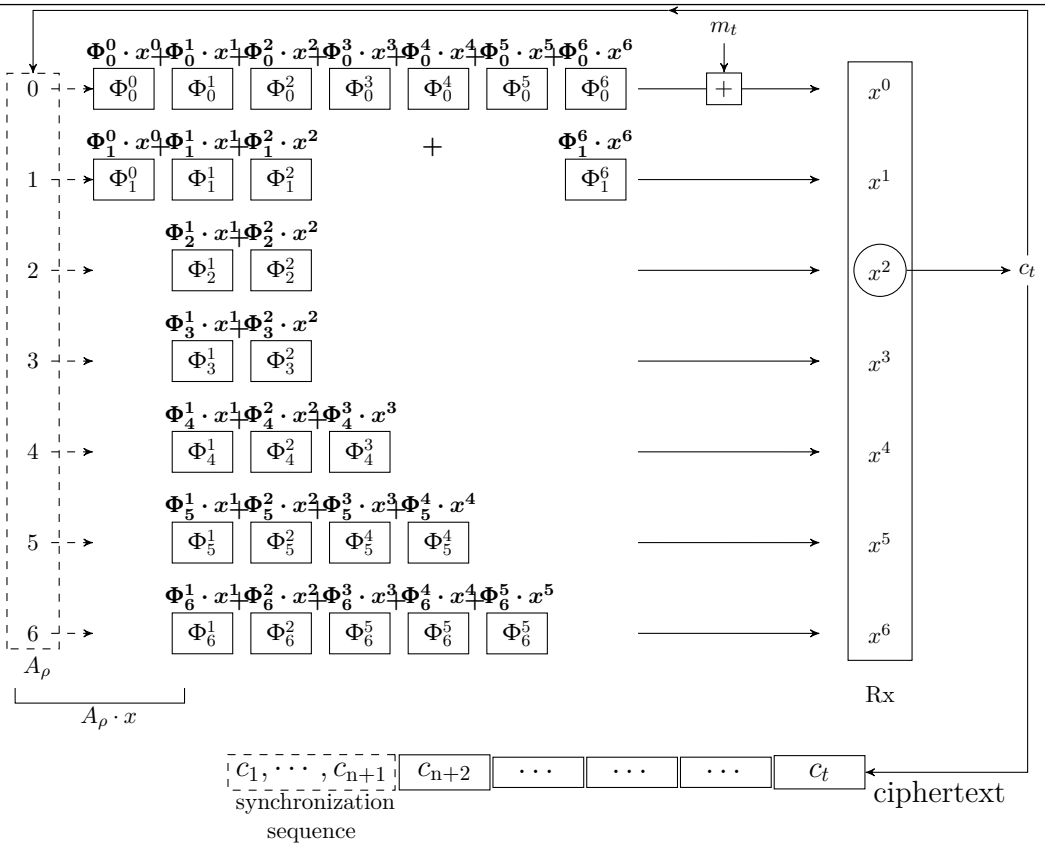


FIGURE 3.11 – Schéma du chiffrement pour $\mathrm{n} = 7, \mathrm{s} = 1, r = 3$. On calcule c_{t+1} à partir de m_t et c_t. Chaque fonction Φ_i^j a pour entrée c_t.

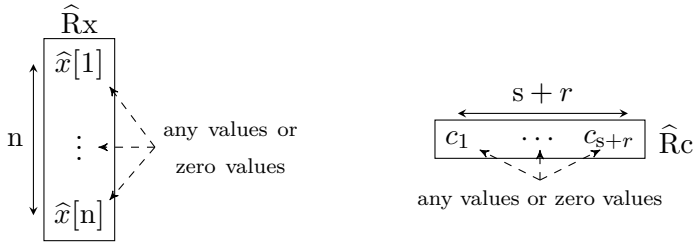


FIGURE 3.12 – Déchiffrement : initialisation du registre $\widehat{\mathrm{Rx}}$ de l'état interne $\widehat{x}$ et du registre $\widehat{\mathrm{Rc}}$ contenant les cryptogrammes. Toutes les valeurs aléatoires sont des symboles de 4 bits.

On procède au déchiffrement en initialisant le registre $\widehat{\mathrm{Rc}}$ et celui de l'état interne $\widehat{\mathrm{Rx}}$ par une séquence de symboles quelconques ou nuls (cf. Figure 3.12).

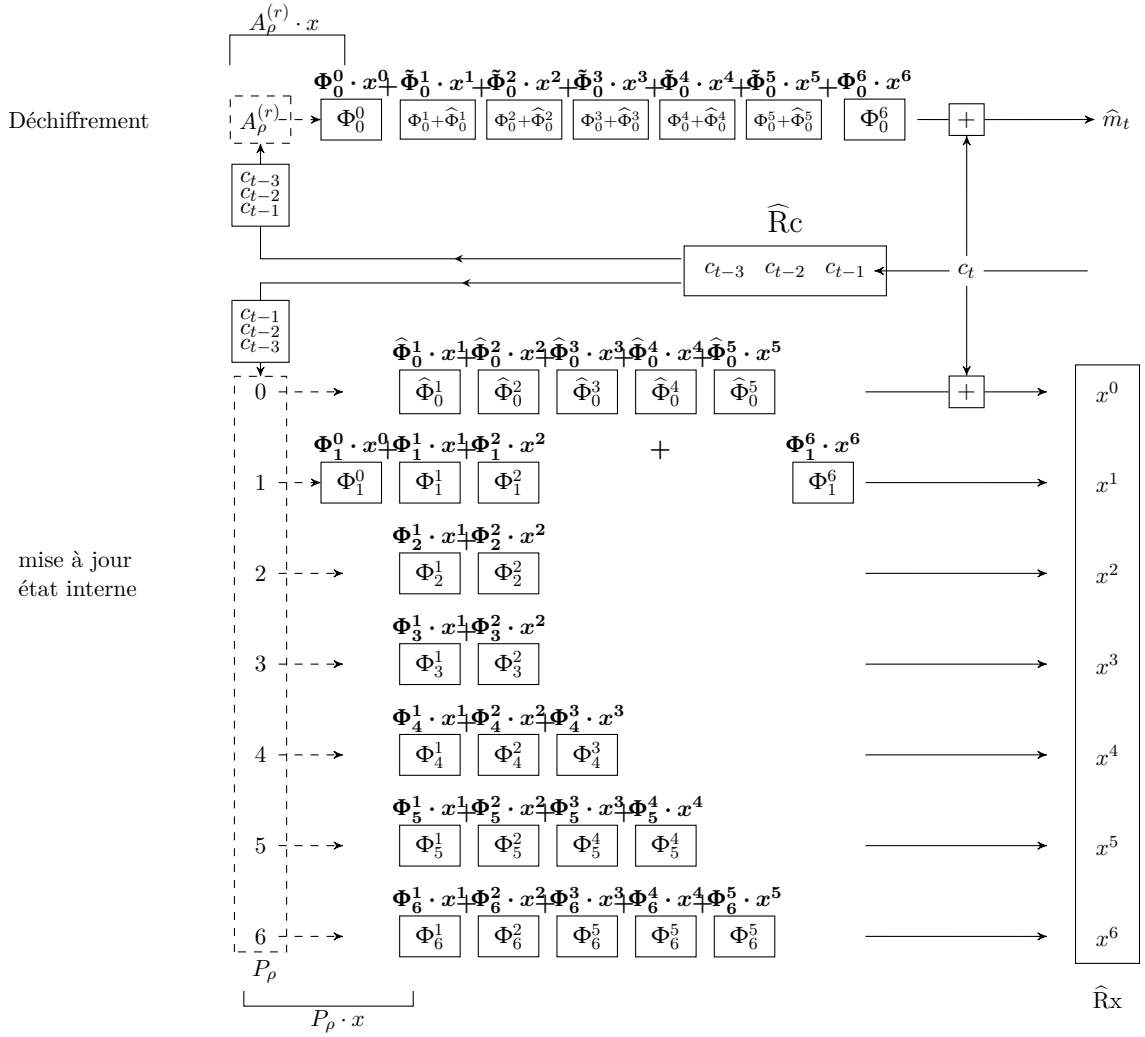


FIGURE 3.13 – Schéma du déchiffrement pour $\mathrm{n} = 7, \mathrm{s} = 1, r = 3$. Le registre $\widehat{\mathrm{Rc}}$ est initialisé à $(0,0,0,0)$. Pour déchiffrer le symbole c_t, on utilise en plus 3 symboles $c_{t-1}, c_{t-2}, c_{t-3}$. On utilise le symbole c_{t-i} comme entrée de la matrice $A_{\rho(t-i)},\ i = 1,2,3$ pour calculer l'état interne $\widehat{x}$ puis les s symboles $c_t, \ldots, c_{t-\mathrm{s}+1}$ comme entrée du vecteur $A_\rho^{(r)}$ pour calculer le clair $\widehat{m}_t$.

3.3 Études sur les matrices

On décrit dans cette section les études et critères de sélection des matrices du système LPV à partir duquel les algorithmes auto-synchronisants ont été implémentés. On rappelle les équations de chiffrement et de déchiffrement (3.1) :

$$\text{Chiffrement :} \quad \begin{array}{rcl} x_{t+1} & = & A_{\rho(t)} \cdot x_t + B \cdot m_t \\ c_{t+1} & = & C \cdot x_{t+1} = x_{t+1}[r] \end{array} \quad t \geq 0$$

$$\text{Déchiffrement :} \quad \left\{ \begin{array}{rcl} \widehat{m}_t & = & C \cdot A^{\rho(t-1)}_{\rho(t-r)} \cdot \widehat{x}_t + c_t \\ \widehat{x}_{t+1} & = & P_{\rho(t-r:t)} \cdot \widehat{x}_t + B \cdot \ c_t \end{array} \right. \quad t \geq 1$$

L'objet de cette partie est de justifier le choix des matrices $A_{\rho(t)}$ et $P_{\rho(t)}$ qui caractérisent respectivement la transition d'état du chiffreur et du déchiffreur.

La notation $A^{\rho(t-1)}_{\rho(t-r)}$ correspond au produit $A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)}$.

Rappelons que cette architecture est obtenue à partir de la structure de graphe orienté décrite dans la Section 2. Aussi la matrice $A_{\rho(t)}$ est obtenue à partir de la matrice d'incidence I_A (voir Equation (2.9)) obtenue à partir du graphe orienté [6]. L'expression de la matrice $P_{\rho(t:t+r)}$ est donnée par :

$$\begin{array}{rcll} P_{\rho(t-r:t)}[0] & = & A_{\rho(t-r)}[0] - (A_{\rho(t-1)} \cdot A_{\rho(t-2)} \cdots A_{\rho(t-r)})[r-1] & \\ P_{\rho(t-r:t)}[i] & = & A_{\rho(t-r)}[i], & 1 \leq i \leq n-1 \end{array} \tag{3.9}$$

L'étude des critères de sélection de ces matrices est basée sur les travaux de Berger *et al.* [40]. On remplace ici des entrées non nulles de la matrice d'incidence I_A par une fonction non linéaire que l'on note par S. Et on note tout simplement la matrice ainsi obtenue par A.

$$\underset{I_A}{\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}} \rightarrow \underset{A}{\begin{pmatrix} 0 & 0 & S & 0 & S & 0 & S \\ 1 & 0 & S & S & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & S & 1 & 0 & 0 & 0 \\ 0 & S & 0 & 0 & 1 & 0 & 0 \\ 0 & S & 0 & 0 & 0 & 1 & 0 \end{pmatrix}}$$

De même on définira par P la matrice associée à $P_{\rho(t:t+r)}$ obtenue à partir de la matrice A en appliquant l'équation (3.9).

On considère ensuite les produits des matrices A et P et on recherche le plus petit entier k tel que A^k ou P^k ne contienne plus de coefficient non nul. Ce plus petit entier k s'appelle *délai de diffusion* : cette notion a été introduite dans [41], puis développée dans [42] et [43]. Une autre mesure qui est également utilisée dans les critères de sélection de ces matrices est la notion de *profondeur* qui a été introduite dans [40] et qui par

6. I_A est en fait la transposée de la matrice d'adjacence. Une matrice d'incidence contient également des coefficients égaux à -1 dans le cas d'un graphe orienté.

définition est le plus petit entier k tel que tous les coefficients des matrices A^k et P^k soient des polynômes de degré supérieur ou égal à 1.

Ainsi les critères de sélection des matrices portent sur un choix optimal du délai de diffusion et de la profondeur à la fois pour les matrices A et P. Il faut noter que cette étape est une première étape qui permet d'exhiber des matrices potentielles $A_{\rho(t)}$ et $P_{\rho(t:t+r)}$. En effet il est très difficile d'appliquer directement les critères de sélection aux matrices $A_{\rho(t)}$ et $P_{\rho(t:t+r)}$, ceci à cause du temps exorbitant nécessaire pour lancer un grand nombre de simulations.

3.3.1 Sélection des matrices

Dans la suite, on considère un système LPV plat de dimension $n = 40$. Le choix d'une telle dimension se justifie par le fait que le chiffreur sous-jacent reste résistant contre des attaques par compromis temps mémoire [44] avec un niveau de sécurité de 80 bits.

En effet, les variables du système LPV sont des quartets, c'est-à-dire des symboles de 4 bits. L'automate associé à ce système LPV qui permet de construire un SSSC admet donc un état interne de 160 bits.

On a donc réalisé des simulations afin d'obtenir les matrices souhaitées. La procédure de simulation consiste à générer de façon aléatoire sous certaines conditions un graphe orienté qui fournit un système dynamique LPV (2.1) plat, puis ensuite d'étudier leurs propriétés par rapport aux critères mentionnés ci dessus.

La Figure 3.14 illustre pour un nombre na donné d'arcs du graphe orienté, c'est-à-dire pour un nombre donné de coefficients non nuls de la matrice P, le pourcentage de matrices P^{40} qui admettent des coefficients nuls.

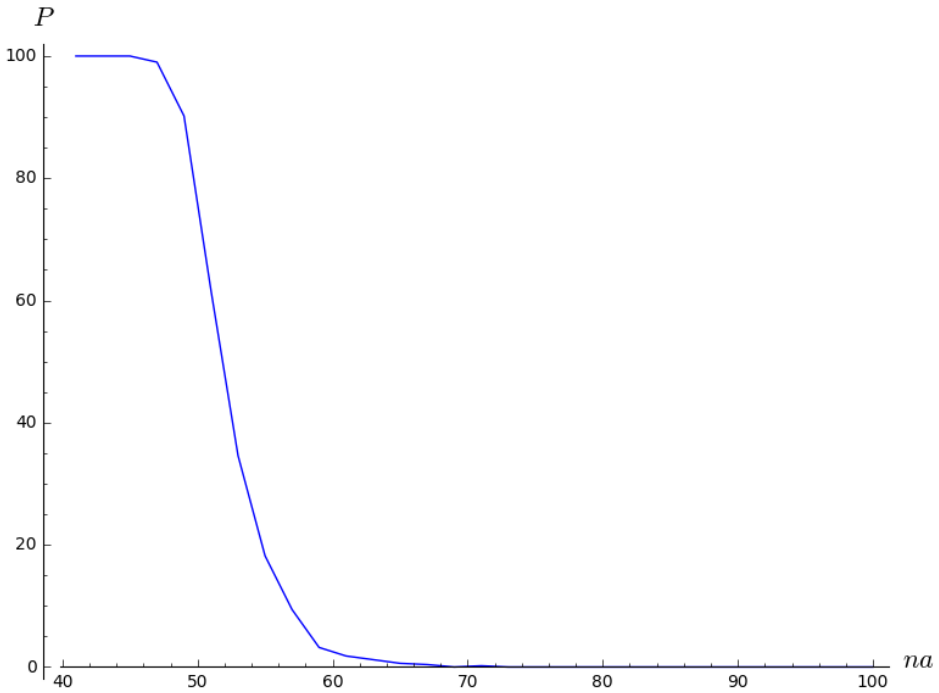


FIGURE 3.14 – Pourcentage P de matrices P^{40} contenant des valeurs nulles dans un ensemble donné de matrices P à au plus na coefficients non nuls.

La Figure 3.15 illustre le même pourcentage à la fois pour la matrice A et P.

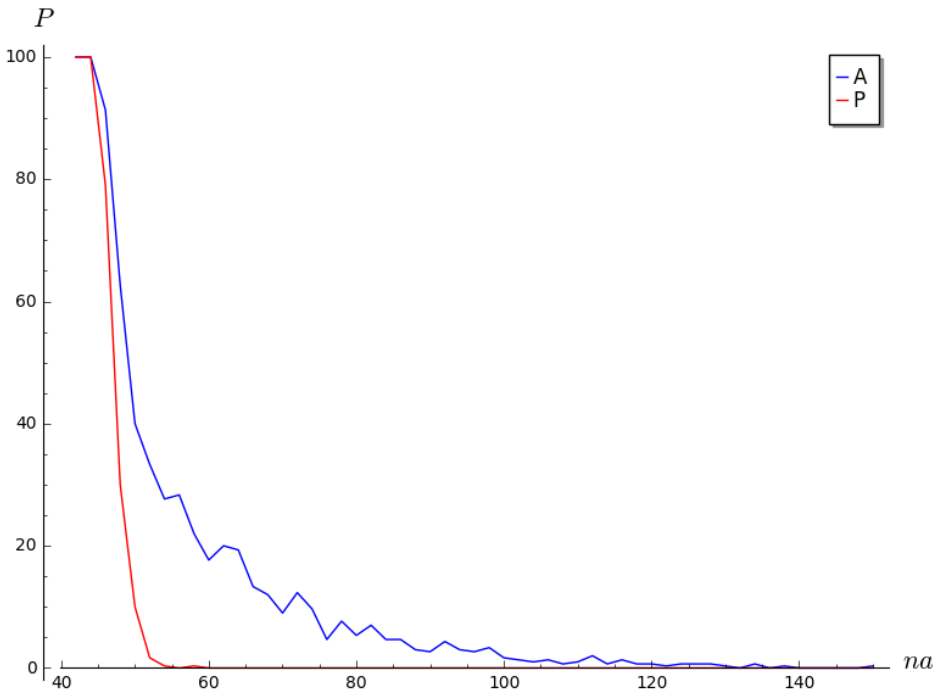


FIGURE 3.15 – Pourcentage P de matrices A^{40} et P^{40} contenant des valeurs nulles dans un ensemble donné de matrices P à au plus na coefficients non nuls.

Les Tables 3.4 et 3.5 correspondent respectivement à la matrice retenue et au délai de diffusion des matrices A et P (valant tous les 7).

3.4 Conclusion du chapitre

Dans ce chapitre, il a été décrit la justification des spécifications de notre algorithme SSSC innovant. Pour ne pas surcharger ce chapitre, les spécifications finales de notre algorithme SSSC sont décrites en annexe.

Une fois l'algorithme décrit, il peut être testé sur notre plate-forme de démonstration afin de pouvoir en constater les avantages opérationnels : c'est tout l'objet du prochain chapitre.

Il faut cependant noter que l'interaction entre les partenaires n'a pas été partitionnée et linéaire pendant le déroulement du projet : Airbus Cybersecurity a défini dès le début du projet les contraintes de sa plate-forme de démonstration, et plusieurs itérations et implémentations successives de l'algorithme ont été nécessaires afin d'arriver à la version finale de l'algorithme décrite dans ce livrable.

[0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	0	0	SB]
[1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	SB	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	SB	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	SB	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	SB	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	SB	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	SB	0	0	0	0	0	0	0	0	0	0	0	SB	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	SB	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0	0	0]
[0	0	SB	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	1	SB	0	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	SB	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0	0]
[0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0	0	0]
[0	0	0	0	0	0	0	SB	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	SB	SB	0	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0	0]
[0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	SB	0	0	0	0]
[0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	1	SB	0	0	0]
[0	SB	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	0	1	SB	0	0]
[0	0	0	0	0	SB	0	SB	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	SB	0	0	0	SB	0	0	1	SB	0]

TABLE 3.4 – Matrice A.

```
A initial set of degrees: [-1, 0, 1]
A^2 set of degrees: [-1, 0, 1, 2]
A^3 set of degrees: [-1, 0, 1, 2, 3]
A^4 set of degrees: [-1, 0, 1, 2, 3, 4]
A^5 set of degrees: [-1, 0, 1, 2, 3, 4, 5]
A^6 set of degrees: [-1, 1, 2, 3, 4, 5, 6]
A^7 set of degrees: [2, 3, 4, 5, 6, 7]
A^8 set of degrees: [4, 5, 6, 7, 8]
A^9 set of degrees: [5, 6, 7, 8, 9]
A^10 set of degrees: [7, 8, 9, 10]
A^11 set of degrees: [9, 10, 11]
A^12 set of degrees: [10, 11, 12]
A^13 set of degrees: [11, 12, 13]
A^14 set of degrees: [12, 13, 14]
A^15 set of degrees: [13, 14, 15]
A^16 set of degrees: [14, 15, 16]
A^17 set of degrees: [15, 16, 17]
A^18 set of degrees: [16, 17, 18]
A^19 set of degrees: [17, 18, 19]
A^20 set of degrees: [18, 19, 20]
A^21 set of degrees: [19, 20, 21]
A^22 set of degrees: [20, 21, 22]
A^23 set of degrees: [21, 22, 23]
A^24 set of degrees: [22, 23, 24]
A^25 set of degrees: [23, 24, 25]
A^26 set of degrees: [24, 25, 26]
A^27 set of degrees: [25, 26, 27]
A^28 set of degrees: [26, 27, 28]
A^29 set of degrees: [27, 28, 29]
A^30 set of degrees: [28, 29, 30]
A^31 set of degrees: [29, 30, 31]
A^32 set of degrees: [30, 31, 32]
A^33 set of degrees: [31, 32, 33]
A^34 set of degrees: [32, 33, 34]
A^35 set of degrees: [33, 34, 35]
A^36 set of degrees: [34, 35, 36]
A^37 set of degrees: [35, 36, 37]
A^38 set of degrees: [36, 37, 38]
A^39 set of degrees: [37, 38, 39]
A^40 set of degrees: [38, 39, 40]
```

```
P initial set of degrees: [-1, 0, 1, 2, 3]
P^2 set of degrees: [-1, 0, 1, 2, 3, 4]
P^3 set of degrees: [-1, 0, 1, 2, 3, 4, 5]
P^4 set of degrees: [-1, 0, 1, 2, 3, 4, 5, 6]
P^5 set of degrees: [-1, 0, 1, 2, 3, 4, 5, 6, 7]
P^6 set of degrees: [-1, 1, 2, 3, 4, 5, 6, 7, 8]
P^7 set of degrees: [2, 3, 4, 5, 6, 7, 8, 9]
P^8 set of degrees: [4, 5, 6, 7, 8, 9, 10]
P^9 set of degrees: [6, 7, 8, 9, 10, 11]
P^10 set of degrees: [8, 9, 10, 11, 12]
P^11 set of degrees: [9, 10, 11, 12, 13]
P^12 set of degrees: [10, 11, 12, 13, 14]
P^13 set of degrees: [11, 12, 13, 14, 15]
P^14 set of degrees: [12, 13, 14, 15, 16]
P^15 set of degrees: [13, 14, 15, 16, 17]
P^16 set of degrees: [14, 15, 16, 17, 18]
P^17 set of degrees: [15, 16, 17, 18, 19]
P^18 set of degrees: [16, 17, 18, 19, 20]
P^19 set of degrees: [17, 18, 19, 20, 21]
P^20 set of degrees: [18, 19, 20, 21, 22]
P^21 set of degrees: [19, 20, 21, 22, 23]
P^22 set of degrees: [20, 21, 22, 23, 24]
P^23 set of degrees: [21, 22, 23, 24, 25]
P^24 set of degrees: [22, 23, 24, 25, 26]
P^25 set of degrees: [23, 24, 25, 26, 27]
P^26 set of degrees: [24, 25, 26, 27, 28]
P^27 set of degrees: [25, 26, 27, 28, 29]
P^28 set of degrees: [26, 27, 28, 29, 30]
P^29 set of degrees: [27, 28, 29, 30, 31]
P^30 set of degrees: [28, 29, 30, 31, 32]
P^31 set of degrees: [29, 30, 31, 32, 33]
P^32 set of degrees: [30, 31, 32, 33, 34]
P^33 set of degrees: [31, 32, 33, 34, 35]
P^34 set of degrees: [32, 33, 34, 35, 36]
P^35 set of degrees: [33, 34, 35, 36, 37]
P^36 set of degrees: [34, 35, 36, 37, 38]
P^37 set of degrees: [35, 36, 37, 38, 39]
P^38 set of degrees: [36, 37, 38, 39, 40]
P^39 set of degrees: [37, 38, 39, 40, 41]
P^40 set of degrees: [38, 39, 40, 41, 42]
```

TABLE 3.5 – Délai de diffusion et profondeur.

4 Descriptif de nos démonstrations

4.1 Introduction du chapitre

Dans le chapitre précédent, il a été justifié d'un point de vue sécuritaire les spécifications d'un nouvel algorithme de chiffrement auto-synchronisant. Ce chapitre décrit les démonstrations permettant d'illustrer les propriétés intéressantes de cet algorithme lorsque ce dernier est utilisé pour sécuriser les communications d'un réseau de capteurs sans-fil mises en jeu dans un système de contrôle industriel.

Ce chapitre décrira donc dans l'ordre plus précisément le cas d'utilisation, la plate-forme de démonstration et les démonstrations elles-mêmes.

4.2 Cas d'utilisation

Le cas d'utilisation des résultats de THE CASCADE s'inscrit dans la protection des systèmes de contrôle industriels (*Industrial Control Systems*, ICS), et des systèmes de contrôle de supervision et d'acquisition de données (*Supervisory Control and Data Acquisition*, SCADA), axe majeur de développement des activités d'Airbus Cybersecurity. En particulier, nous nous concentrerons sur la sécurité des communications d'un réseau de capteurs sans-fil se transmettant des informations de contrôle. Beaucoup d'ICS-SCADA déjà installés ou en cours d'installation sont ainsi potentiellement concernés : centrales nucléaires, réseau de distribution d'eau, pipelines pétrolifères, etc. Ces informations critiques (niveau d'un liquide, présence ou absence de tel élément chimique, etc.) se doivent d'être protégés notamment en confidentialité pour éviter que d'éventuels attaquants puissent par la suite injecter des paquets réseaux compromettants. De plus, la criticité de ces communications étant élevée, la capacité des algorithmes auto-synchronisants à pouvoir se resynchroniser en cas de perte de paquets réseaux est très utile pour assurer une fiabilité des communications entre capteurs, fiabilité qui peut être mise à mal lorsque les capteurs sont placés dans des environnements hostiles (ex. : capteurs pétrolifères opérant pendant une tempête de sable).

Dans notre nouvelle démonstration, nous proposons de montrer l'intérêt du développement des algorithmes de THE CASCADE via le chiffrement d'une communication sans fil entre deux modules.

4.3 Plate-forme de démonstration

Tout au long du projet, des démonstrations ont été présentées devant les partenaires de THE CASCADE. Pour cela, nous avons utilisé deux Arduinos que nous avons fait communiquer entre eux par l'intermédiaire de modules XBee.

Les Arduinos possèdent les caractéristiques suivantes :

- Processeur ATmega2560, fonctionnant à 16 MHz
- Mémoire Flash 256 ko
- SRAM 8 ko
- EEPROM 4 ko

L'algorithme de chiffrement auto-synchronisant a été implémenté en C dans l'Arduino.

De plus, les Arduinos sont équipés d'un *Arduino Wireless Proto Shield* et d'un module Xbee S1, comme on peut le voir en Figure 4.1.

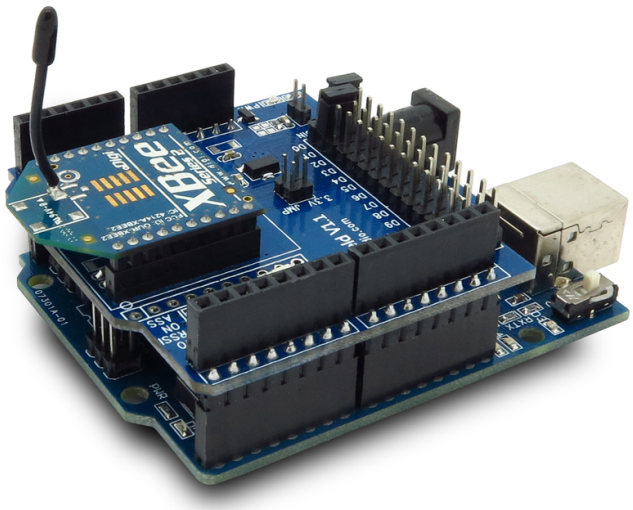


FIGURE 4.1 – Arduino et module Xbee

Le protocole sans fil utilisé est du IEEE 802.15.4. L'avantage de ce protocole assez simple est qu'il ne possède qu'une couche physique et une couche MAC définies, ce qui permet de créer une couche personnalisée par-dessus et de la chiffrer entièrement.
La taille maximale d'un paquet en suivant ce protocole est de 127 octets, ce qui nous laisse 102 octets pour le *payload*, la couche MAC occupant déjà 25 octets [1].

Afin de tester les capacités mémoire des Arduinos utilisés, nous avons préalablement implanté les algorithmes RC4 (chiffrement par flot non autosynchronisant), PRESENT (algorithme de chiffrement léger) et AES (algorithme de chiffrement standard) dans les Arduinos. Les trois implantations n'ont pas excédé les capacités mémoire des Arduinos, ce qui fût de bonne augure pour l'implantation d'algorithmes auto-synchronisants.

1. Il est également possible de chiffrer les données du *payload* en AES mode compteur (AES-CTR) par le module XBee. Il suffit de renseigner une clé de 128 bits lors de la configuration du module et de procéder à l'activation du chiffrement. Il est cependant inutile de l'activer dans notre cas d'utilisation car le *payload* sera déjà chiffré avec les algorithmes développés dans THE CASCADE.

Il s'est avéré à la suite de ces travaux préliminaires que les ressources présentes dans le processeur cible étaient bien suffisantes pour pouvoir contenir nos différentes versions d'algorithmes auto-synchronisants.

4.4 Démonstrations des propriétés des algorithmes développés

Afin de réaliser des démonstrations du fonctionnement de l'algorithme développé dans le cadre du projet sur Arduino, nous avons utilisé deux Arduinos, équipés chacun d'un module XBee permettant de communiquer entre ces deux cartes.
Un des Arduinos gère la partie chiffrement et l'autre gère la partie déchiffrement.

Le dispositif de démonstration complet est montré en Figure 4.2.

FIGURE 4.2 – Dispositif de démonstration complet

Afin d'anticiper la version finale de l'algorithme, nous avons commencé par réaliser à mi-projet l'implémentation d'une version réduite de ce dernier (de dimension 7). L'implémentation logicielle de référence a été réalisée en langage C : elle a permis non seulement de réaliser les démonstrations mais également de valider les implémentations matérielles de l'algorithme. Lors de la phase d'intégration du code C dans les Arduinos, une des phases les plus longues a consisté à gérer la transmission des messages entre les deux cartes.
Nous avons voulu que ces démonstrations sur Arduinos soient le plus visuelles possible afin de pouvoir permettre de vérifier et de prouver l'efficacité de l'algorithme. Il y a plusieurs points à vérifier :

- **Le fonctionnement de l'algorithme**, c'est-à-dire que l'Arduino récupérant un texte chiffré peut bien le déchiffrer s'il possède la même clé que le chiffreur.
- **La resynchronisation du système**. On doit pouvoir voir que quelque soit l'état dans lequel le déchiffreur est initialisé, il peut se resynchroniser en un certain nombre de quartets reçus.

— **La resynchronisation après erreur**. À la suite d'une erreur de transmission, il peut se resynchroniser en un certain nombre de quartets reçus.

Pour rendre ces démonstrations visuelles, nous avons choisi d'utiliser le message à chiffrer/déchiffrer "THECA5CADE THECA5CADE" pour un rendu vérifiable immédiatement.

Au vu des caractéristiques de l'algorithme, on sait que le système a besoin d'un maximum de 40 itérations pour se resynchroniser. Il faut donc garder à l'esprit que les 40 premiers quartets de message envoyés par l'Arduino réalisant le chiffrement seront perdus. Cette valeur peut, selon la clé, le message, ou l'état interne du système être inférieure en pratique (35 quartets nécessaires au minimum) mais la valeur nécessaire pour s'assurer d'une resynchronisation à tous les coups est de 40 quartets.

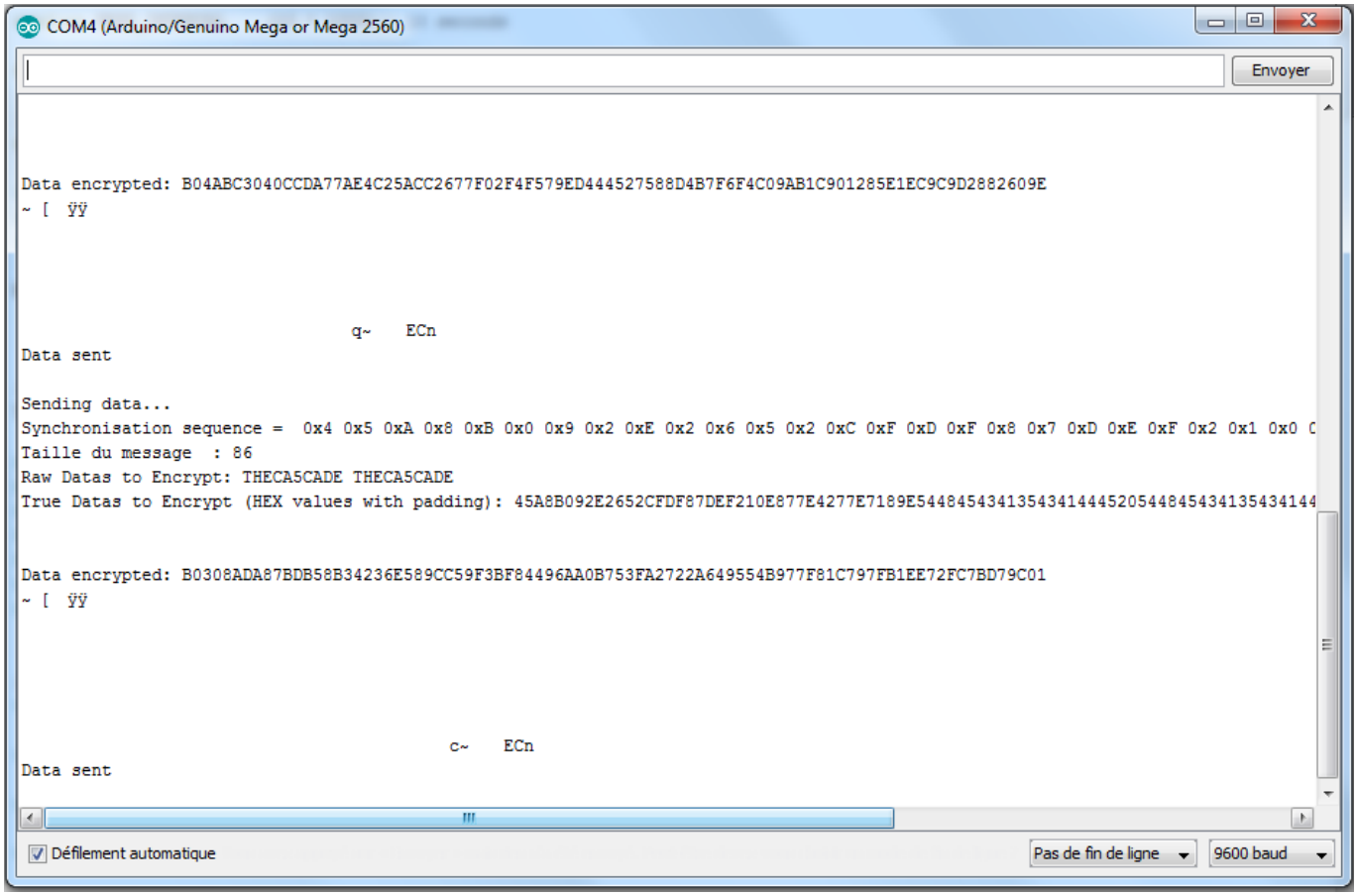


FIGURE 4.3 – Démonstration de chiffrement sur Arduino

Pour obtenir des résultats intéressants, une fois le système initialisé, c'est-à-dire une fois que les états internes initiaux ont été générés et que les sous-clés ont été calculées, l'Arduino réalisant le chiffrement va envoyer des messages chiffrés. Ces messages sont envoyés toutes les 10 secondes avec à chaque fois la partie nécessaire à la resynchronisation (les 40 premiers quartets) générée aléatoirement. Les états internes sont passés en variable globale dans le code C utilisé, pour que d'un chiffrement à l'autre les états internes soient réutilisés.

L'Arduino réalisant le déchiffrement va donc recevoir toutes les 10 secondes [2] un nouveau message à déchiffrer. Les états internes ne sont pas les mêmes que ceux utilisés par l'Arduino réalisant le chiffrement, pour montrer l'efficacité de la resynchronisation du système.

2. Ce temps est bien entendu paramétrable par l'utilisateur de la démonstration et il doit être représentatif du cas réel rencontré sur le terrain.

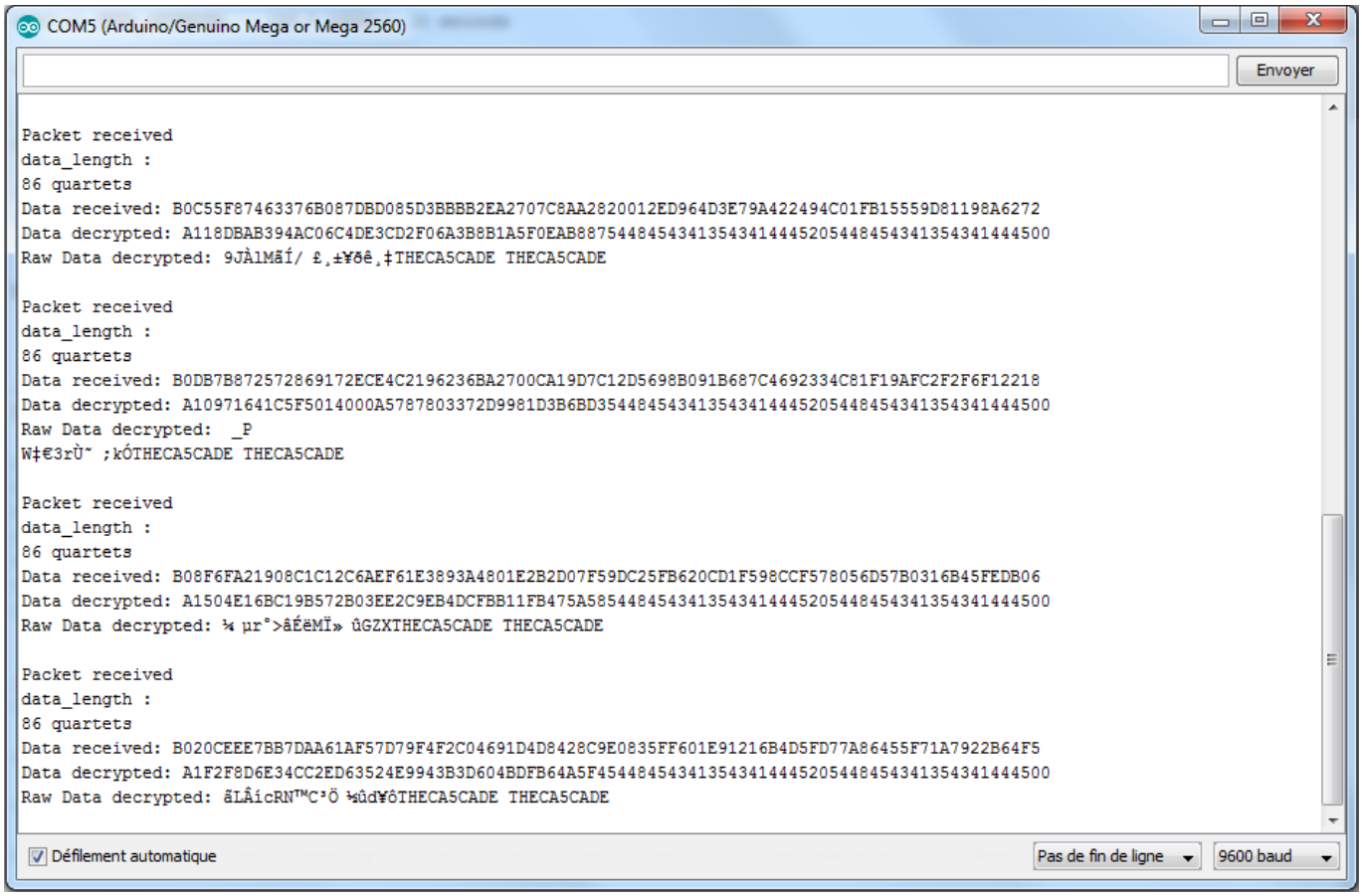


FIGURE 4.4 – Démonstration de déchiffrement sur Arduino

On observe bien que tous les 40 quartets reçus au plus, les deux Arduinos sont bien resynchronisés, et on déchiffre bien le message “THECA5CADE THECA5CADE”.

Les démonstrations ont été réalisées avec les différentes versions de l’algorithme pour vérifier le bon fonctionnement de celles-ci.

Plusieurs démonstrations ont été réalisées, en suivant les travaux décrits précédemment. Une démonstration consiste à envoyer simplement le même message plusieurs fois avec les mêmes états internes initiaux utilisés à la fois par le chiffreur et le déchiffreur (cf. Figure 4.3).
La seconde consiste à générer les états internes de façon aléatoire. Le chiffreur et le déchiffreur n’ont donc pas les mêmes états internes et on montre bien que le système va réussir tout de même à se resynchroniser au bout de 40 cycles d’horloge au plus (cf. Figure 4.4).
La dernière démonstration consiste à émuler une communication bruitée, c’est-à-dire qu’ici, tous les 50 caractères envoyés, un caractère est remplacé par le caractère `0x01` en hexadécimal (cf. Figure 4.5). Encore une fois, on remarque bien que le système arrive bien à se resynchroniser malgré les caractères erronés (cf. Figure 4.6).

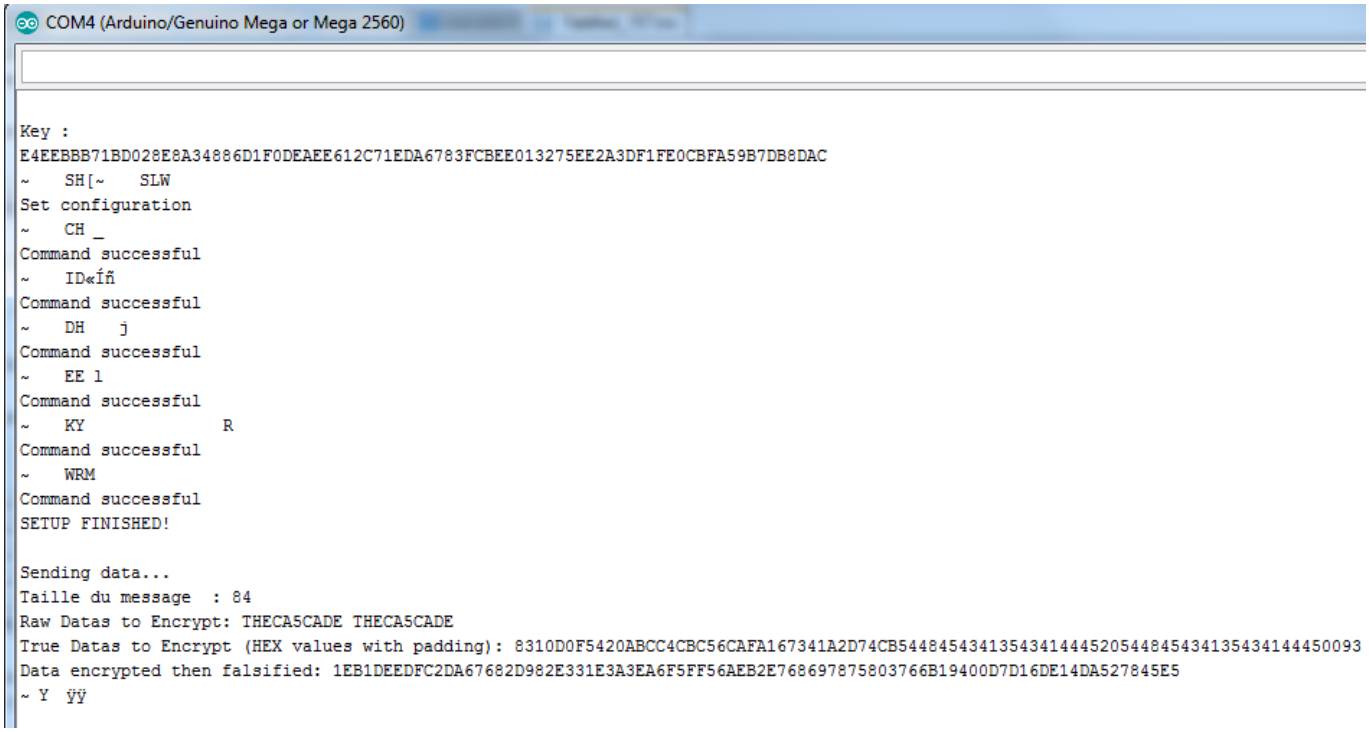


Figure 4.5 – Chiffreur de messages erronés

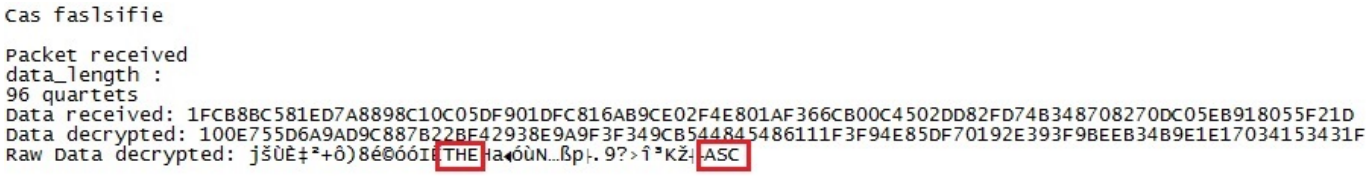


Figure 4.6 – Déchiffreur de messages erronés

4.5 Conclusion du chapitre

Ce chapitre a montré le déroulé des démonstrations réalisées sur Arduino et les propriétés intéressantes des algorithmes auto-synchronisants développés par THE CASCADE. Notamment, la resynchronisation très rapide en cas de bruit injecté sur le canal de communication, le tout sans ré-emission de paquets ou envoi de séquences explicites de synchronisation (exemples : numéro de trame, IV, etc.), est très pratique dans certaines applications, tels les ICS/SCADA (comme montré ici) ou les télécommunications. Il a ainsi été notamment montré que les différentes versions d'algorithme proposées par le consortium sont compatibles avec les contraintes des systèmes embarqués pour ce qui concerne le coût d'implémentation logicielle. Le coût d'implémentation matérielle est lui décrit dans le chapitre suivant.

5 Implémentations matérielles

5.1 Introduction du chapitre

Ce chapitre décrit l'étude du coût des implémentations matérielles de notre algorithme SSSC innovant. Nous avons montré au chapitre précédent que notre algorithme s'implémentait sans difficulté dans notre plate-forme logicielle ; nous devons faire le même type d'étude sur plate-formes matérielles.

Nous commencerons tout d'abord par décrire l'architecture matérielle que nous avons développée pour implémenter les différentes versions de notre algorithme SSSC, puis les plate-formes matérielles de test et enfin nous donnerons nos résultats d'implémentations que nous discuterons.

5.2 Architecture matérielle du SSSC

5.2.1 Création de la matrice

Pour réaliser la matrice de chiffrement qui va fournir, à chaque coup d'horloge, les états internes du système, nous sommes partis d'un composant de base appelé SB_KC (pour SBOX de clé et chiffré) qui permet de réaliser les fonctions Φ_i utilisées par la matrice.
De ce composant, découlent les différentes CELL (les cellules de la matrice), qui consistent en la multiplication des fonctions Φ_i par un état interne (cf. Figure 5.1).

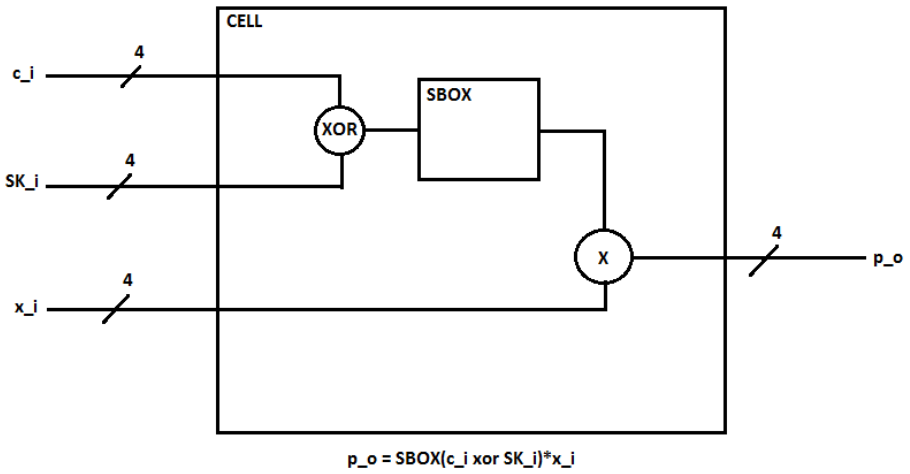


FIGURE 5.1 – Schéma d'une CELL

Nous avons ensuite créé les composants PROD_LINEX, permettant de réaliser la mise à jour des 40 états internes de la matrice. Ce sont des combinaisons linéaires de CELL qui prennent en entrée les 40 états internes, les sous-clés et le chiffré de l'état précédent et qui vont donner en sortie 40 quartets. Ensuite les composants FULL_LINEX, associés à chaque PROD_LINEX, vont permettre de donner chaque état interne en appliquant un XOR sur les 40 quartets récupérés en sortie de chaque PROD_LINEX. Les différentes FULL_LINEX sont ensuite "assemblés" dans le composant MATRIX qui va associer chaque composant aux bons états internes et aux bonnes sous-clés pour donner les états internes mis à jour à chaque coup d'horloge (cf. Figure 5.2).
D'une itération de l'algorithme à l'autre, les différents PROD_LINEX, ainsi que le composant MATRIX vont être modifiés pour s'adapter aux nouvelles spécifications. Le mappage des états internes et des sous-clés peuvent également varier d'une itération à l'autre.
Au sein d'une même version d'un algorithme, la différence entre le calcul du chiffré et du déchiffré réside dans la première ligne de la matrice utilisée pour la mise à jour des états internes ; dans le code VHDL que nous avons développé, cette différence se situe dans le composant PROD_LINE0 qui calcule cette première ligne de la matrice.

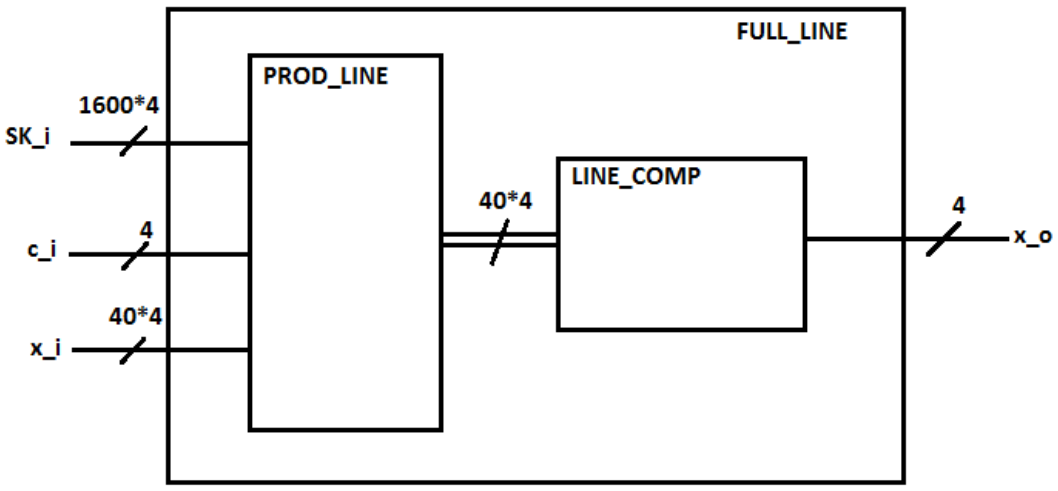


FIGURE 5.2 – Création de la matrice, ligne à ligne

5.2.2 Expansion de clés

Pour réaliser l'expansion de clés, on part d'un composant LFSR qui prend en entrée 88 bits, placés dans 88 registres et qui vont être décalés d'un registre à la fois à chaque coup d'horloge (cf. Figure 5.3). On récupère en sortie 8 bits qui vont être ajoutés au composant NLFSR, en plus des 88 registres de sortie pour rajouter la composante non-linéaire à notre génération de sous-clés. Ce composant NLFSR va donner 88 bits en sortie qui vont être réinjectés en entrée du LFSR. On récupère de plus un bit, correspondant au registre 0 de ce NLFSR, qui va donner l'ensemble des sous-clés.
Tous les quatre cycles d'horloge un signal en_quart_s va passer à 1, permettant de récupérer les quatre bits récupérés du NLFSR et de les placer dans un quartet. Pour chaque passage à 1 de ce signal, un compteur va être incrémenté pour connaître le nombre de quartets de sous-clés générés. Ces quartets vont ensuite être placés dans la matrice 40×40 aux places voulues grâce à un composant SK_TAB (pour table des sous-clés) associant à chaque valeur du compteur la place voulue dans la matrice. Ce composant change pour

chaque version de l'algorithme, le placement des sous-clés étant différent à chaque fois. On va ensuite pouvoir récupérer en sortie du composant PILOTED_SK_TAB les 1600 quartets de sous-clés générés.
Dans la version dite "*full*", 781 quartets de sous-clés sont générés, 80 pour les autres versions. Une fois le compteur de sous-clés arrivé à la valeur correspondante, un signal en_cipher_s est passé à 1, indiquant au système que le chiffrement ou le déchiffrement peut commencer.

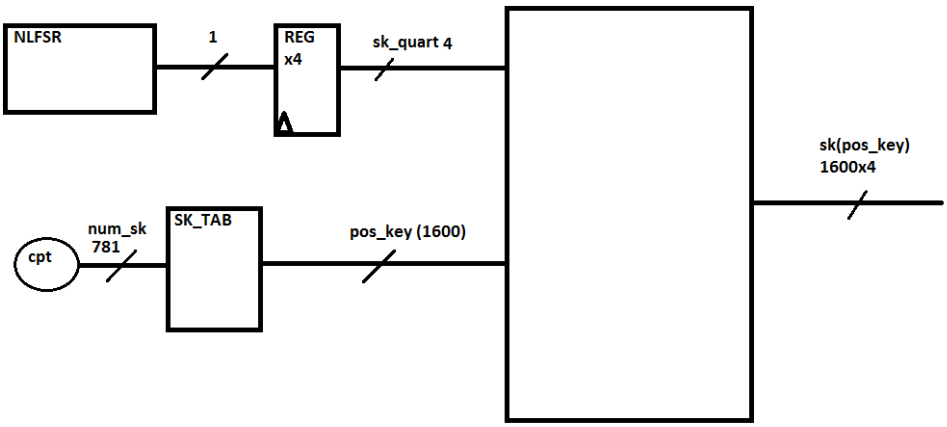


FIGURE 5.3 – Génération des sous-clés

5.2.3 Génération des états internes

Comme décrit précédemment, les états internes à l'initialisation du système doivent être générés de façon aléatoire (ou au moins pseudo-aléatoire). Pour cela, on reprend le composant LFSR utilisé pour la génération de clé, auquel on place en entrée les 88 bits de la clé maître, que l'on va faire tourner d'abord 90 tours à vide. Une fois ces 90 tours passées, on va continuer à décaler les registres et sortir un bit à chaque coup d'horloge. Tous les quatre cycles d'horloge, ces quatre bits récupérés vont être assemblés pour former un quartet. On doit récupérer 40 quartets d'états internes pour initialiser le système.

5.2.4 Machine à états

Pour piloter l'ensemble du système, nous avons recouru à une machine à états permettant de gérer à tout moment l'état du système et ainsi les principaux signaux de commande.
Le système peut être dans trois états différents (cf. Figure 5.4) :

- init : le signal de *reset* est à 0, le système est dans son état de départ, avec tous les signaux à 0.
- key_generation : le système va commencer à générer les sous-clés et les états internes. Si le signal d'entrée new_key est passé à 1, signe que l'utilisateur veut changer la clé maître, le système va repasser dans l'état key_generation.
- key_done : le système a généré toutes les sous-clés et les états internes, il peut donc passer au chiffrement/déchiffrement.

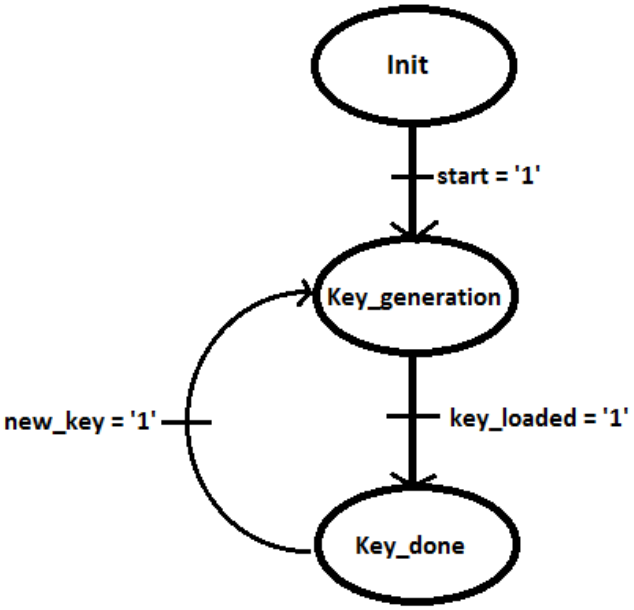


FIGURE 5.4 – Fonctionnement de la machine à états

5.2.5 Calcul du chiffré et du déchiffré

Une fois le système initialisé avec les états internes et les sous-clés, le composant principal MAIN peut commencer à chiffrer ou déchiffrer (cf. Figure 5.5). Tous les cycles d'horloge, le système va donc prendre un quartet en entrée et va donner en sortie un quartet de message chiffré (ou déchiffré).
Dans le cas du chiffrement, le calcul est assez simple puisque le chiffré correspond au second état interne. Dans le cas du déchiffrement, ce calcul est plus complexe et varie d'une version à l'autre. Ce calcul est réalisé par le composant MAIN.
On peut facilement passer du chiffrement au déchiffrement à l'aide d'un signal d'entrée binaire. S'il est à l'état haut, le système va chiffrer l'entrée, s'il est à l'état bas, le système va déchiffrer l'entrée.

5.2.6 Implémentation des algorithmes concurrents

Pour comparer les résultats obtenus avec les différentes versions de notre algorithme, le principe était d'implémenter également certains des algorithmes de chiffrement par flot présents dans la littérature. C'est pourquoi, le choix s'est porté sur Grain128 [45], Trivium [46] et Moutique [47], trois algorithmes présentés pour le projet eSTREAM, Moustique ayant été cassé mais restant le dernier algorithme auto-synchronisant reconnu par la communauté scientifique.
Nous avons pu réutiliser les codes C de ces derniers algorithmes sur le site du projet eSTREAM, et nous avons pu trouver les codes VHDL de Grain128 et Trivium en accès libre sur Internet.

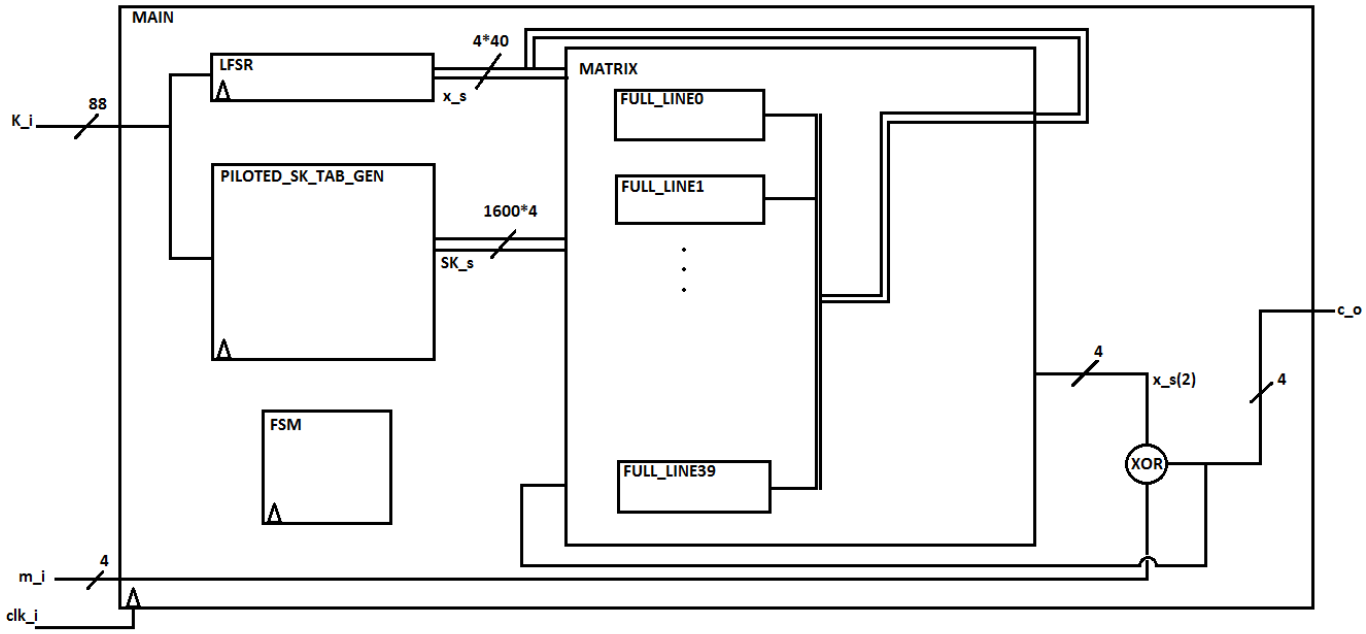


FIGURE 5.5 – Schéma simplifié du MAIN

5.3 Plate-formes d'implémentation matérielle

2 plate-formes d'implémentation matérielle ont été utilisées afin de pouvoir comparer les performances des algorithmes entre elles : un circuit programmable de type FPGA (*Field Programmable Gate Array*) et une technologie pour circuit intégré propre à une application (*Application-Specific Integrated Circuit*, ASIC).

5.3.1 Circuit programmable (FPGA)

N'ayant pas les ressources nécessaires au prototypage d'un ASIC complet et fonctionnel, nous avons tout d'abord prototypé nos implémentations matérielles sur FPGA. Plus précisément, nous avons utilisé la plate-forme SAKURA-G, qui embarque *in situ* un dispositif de prise de mesure bas bruit de courant électrique facilitant les attaques SCA. La SAKURA-G intègre deux FPGAs : l'un dit "de contrôle" qui s'occupe de s'interfacer avec l'extérieur de la carte et qui envoie au second FPGA des vecteurs d'entrée, l'autre dit "principal" qui est le circuit siège des prises de mesures de canaux auxiliaires. Voici les principales caractéristiques de cette plate-forme (cf. Figure 5.6) :

- 2 FPGAs Spartan-6 (principal : XC6SLX-75, de contrôle : XC6SLX-9)
- Fréquence de l'oscillateur : 48 MHz
- Point de mesure situé sur la broche VCCINT du FPGA principal
- Amplification : bande passante de 360 MHz et gain de 20 dB
- Interconnexion entre les deux FPGAs : bus de 51 bits
- Dimensions : 140mm × 120 mm

Nos résultats d'implémentation sur cette plate-forme sont décrits dans la Figure 7.6. Nos implémentations en VHDL ont été synthétisés grâce au logiciel ISE version 14.4, avec une priorité portée sur l'optimisation

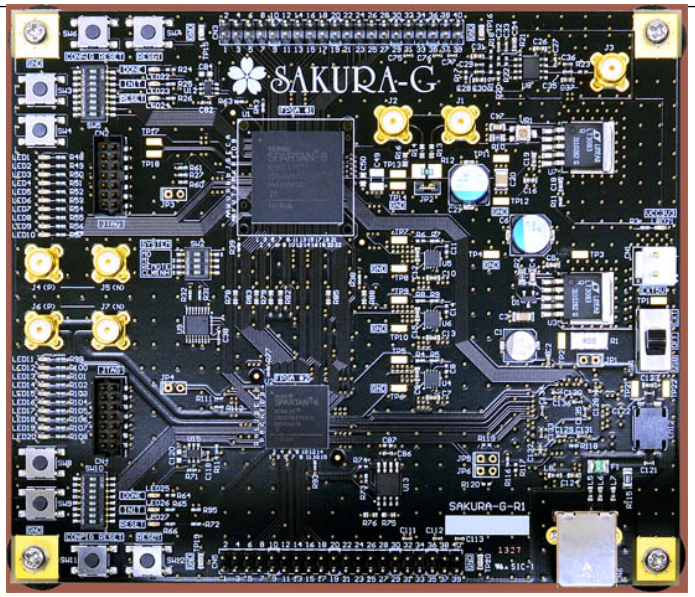


FIGURE 5.6 – SAKURA-G

de la surface occupée. Une fréquence de 100 MHz a été fournie comme contrainte aux outils de synthèse et de placement-routage du circuit.

Version	Slices	LUTs	Flip-Flops	Freq. Max. (MHz)	Débit (Mbps)	Cycles Init.	Cycles Resynchro.
Full	4932	11735	3505	67	268	3136	**40**
Red1	1074	2757	876	97	388	332	**40**
Red2	713	2254	876	98	**392**	332	**40**
Mul1	618	2005	876	96	384	332	**40**
Mul157	691	2293	876	87	348	332	**40**
Grain128	26	55	57	267	267	256	–
Trivium	54	212	288	165	165	1603	–
Moustique	169	385	670	266	266	104	105

FIGURE 5.7 – Résultats d'implémentation sur SAKURA-G

Nous avons implémenté régulièrement tout au long du projet plusieurs itérations de l'algorithme LPV-SSSC. Ici, une architecture utilisant une clé maître de 88 bits a été implémentée :

- La version "Full" voit sa matrice 40×40 remplie par des SBoxes à 781 endroits, les autres versions n'implémentant que 80 SBoxes dans leur matrice de chiffrement.
- Pour les versions "Red1" et "Red2", les éléments de leur matrice de chiffrement ne sont que potentiellement XORés entre eux.
- Pour les versions "Mul1" et "Mul157", les éléments de leur matrice de chiffrement sont également potentiellement XORés entre eux, par contre pour ce qui concerne les matrices de déchiffrement, les éléments de ces dernières sont potentiellement multipliés entre eux. Pour le déchiffrement de "Mul1", on conserve les 80 SBoxes de la matrice de chiffrement et on en utilise 40 qui seront multipliées au maximum une à une. Pour le déchiffrement de "Mul157", on reprend 77 SBoxes de la matrice de chiffrement, qui sont susceptibles d'être multipliés entre eux.

Nous résumons ci-après les conclusions de notre étude :

- Globalement, les algorithmes LPV-SSSC occupent plus d'espace que les algorithmes de chiffrement par flot standards et Moustique (exemple d'algorithme autosynchronisant). On peut constater un facteur 4 environ entre la surface de Moustique et les LPV-SSSC les plus compacts.
- Cette différence est provoquée par une plus grande utilisation des éléments constitutifs des FPGA (appelés "*slices*") : les composants combinatoires (LUTs) et séquentiels (bascules ou *Flip-Flops*). En effet, implémenter les SBoxes demandent des ressources importantes. On peut constater un facteur 10 environ entre le nombre de LUTs utilisées par Moustique et les LPV-SSSC les plus compacts.
- Cette impression générale est renforcée par l'analyse de la proportion LUT/Flip-Flop pour chaque type d'algorithme : environ 3/1 pour les LPV-SSSC implémentés, contre 1/1 pour Grain128 et Trivium, et 1/2 pour Moustique. On a donc bien respectivement une prédominance de l'utilisation d'éléments combinatoires pour les LPV-SSSC et d'éléments de mémorisation pour Moustique.
- Pour ce qui concerne les fréquences maximum de fonctionnement, on peut constater que Moustique et Grain128 fonctionnent environ à une fréquence 3 fois plus grande que les LPV-SSSC. Ceci s'explique car, pour LPV-SSSC, il y a un grand nombre de SBoxes à traverser sur le chemin critique, tandis que pour Moustique et Grain128, il suffit de traverser quelques portes logiques élémentaires (XORs, ANDs).
- Ceci étant dit, le débit produit par les LPV-SSSC sont nettement supérieurs à leurs concurrents : par exemple, la version "Red2" est presque 50% plus rapide que Moustique, et plus de 2 fois plus rapide que Trivium. **On voit ici l'intérêt d'avoir conçu dès le départ les LPV-SSSC afin qu'ils puissent traiter des quartets (avec des opérations effectuées dans GF(2^4)) et qu'ils puissent donc "sortir" 4 bits par itération/cycle d'horloge. Cela constitue l'un des résultats majeurs de nos travaux.**
- En terme d'initialisation, LPV-SSSC se trouve dans une position intermédiaire : quelques centaines de cycles environ pour LPV-SSSC, contre une centaine pour Moustique mais contre un millier de cycles pour Trivium, algorithme par flot standard.
- Pour ce qui concerne le nombre de cycles nécessaire à la resynchronisation, LPV-SSSC surpasse Moustique : seulement 40 cycles sont nécessaires pour LPV-SSSC contre 105 pour Moustique. **Là encore, le fait de considérer des quartets a été judicieux pour LPV-SSSC. C'est un autre résultat intéressant à mettre au crédit de notre LPV-SSSC.**

5.3.2 Technologie ASIC

Les résultats d'implémentation sur FPGA sont assez pratiques pour pouvoir comparer plus facilement deux implémentations différentes sur une même plate-forme, mais cette comparaison peut diverger par rapport à une implémentation sur technologie ASIC. En effet, une implémentation d'un algorithme sur FPGA peut instancier plus ou moins complètement les composants élémentaires d'un FPGA, tandis qu'en ASIC, une fonction logique présente est obligatoirement utile.

C'est pour cela que nous avons également décidé de donner des résultats d'implémentation ASIC, le tout sur une technologie CMOS "*High Vt*" 65nm basse consommation. Nous avons utilisé Synopsis Design Vision D-2010.03-SP5-2 pour la synthèse. Les valeurs de fonderie typiques ont été configurées : 1,2 V pour la tension du coeur et 25 degrés pour la température. Aucune scan *Flip-Flop* n'a été instanciée. La fréquence cible qui a été fournie aux outils de synthèse et de placement-routage a été de 100 MHz. La synthèse et le placement-routage ont été réalisés avec une optimisation de surface en priorité. Au lieu de fournir nos résultats en μm^2, nous avons pris le parti d'exprimer nos résultats en portes équivalentes (*Gate Equivalents*, GEs), c'est-à-dire en fait en unité de surface normalisée : pour obtenir la surface d'un circuit en nombre de GEs, on divise la surface du circuit en μm^2 par la surface d'une porte NAND 2 entrées dans la technologie

CMOS considérée. Ceci permet notamment de comparer plus facilement deux implémentations de circuits utilisant deux technologies ASIC différentes. Dans la technologie ASIC considérée, la surface d'une GE est égale à 2,08 μm^2.

Version	Niveaux de logique	Combin. (GEs)	Non-combin. (GEs)	Surface totale (GEs)	Chemin critique (ns)
Mul157	59	29580	34592	64172	14,2
Masquée	68	105660	34592	140252	16,1
Trivium	14	889	1148	2037	3,0
Grain128s	10	159	1365	1524	2,4
Grain128f	10	161	1381	1542	1,6
Moustique	2	2438	3390	5828	0,7

FIGURE 5.8 – Résultats d'implémentation sur ASIC en technologie 65 nm

Voici décrit ci-dessous la signification des différentes appellations utilisées dans la colonne "Version" du tableau 5.8 :

— "Mul157" et "Moustique" ont déjà été décrits précédemment.
— "Masquée" est une version masquée, c'est-à-dire protégée vis-à-vis des attaques par canaux auxiliaires. Nous avons un peu anticipé le chapitre 7, où les principes de ces attaques et des contre-mesures associées y seront décrites. Il est ici fourni les résultats d'implémentation de la méthode dite par table globale de correspondance (*Global LUT*, GLUT [48]).
— "Grain128s" et "Grain128f" sont respectivement des implémentations de l'algorithme Grain en version lente (non parallélisable) et rapide (parallélisable).

Nous résumons ci-après les conclusions de notre étude :

— On peut constater que les résultats d'implémentation sur ASIC sont globalement beaucoup moins flatteurs que les résultats obtenus sur FPGA. En effet, dans ce dernier cas, selon l'implémentation de l'algorithme considérée, on peut constater de fortes disparités dans l'utilisation des fonctions élémentaires des *slices* du FPGA (LUTs, Flip-Flops). Les comparaisons objectives de surface sont ainsi souvent plus difficiles à interpréter sur FPGA que sur ASIC. Dans ce dernier cas, le synthétiseur est sans appel : il implémente toutes les cellules dont il a besoin. Et dans notre cas, le rapport surface LPV-SSSC/surface Moustique s'aggrave au détriment de LPV-SSSC : d'un rapport d'environ 4 (pour ce qui concerne les *slices*), on passe à un rapport d'environ 11 (pour ce qui concerne les GEs). Il y a en effet plus de combinatoire à implémenter (environ 10 fois plus) à cause du coût unitaire d'une boîte-S (environ 23 GEs). Il ne faut pas non plus négliger le coût du routage dans la catégorie "non-combinatoire" : le routage de Moustique est assez simple car il est assez régulier, avec une topologie "en triangle", tandis que que pour LPV-SSSC, le routage est beaucoup plus irrégulier.
— LPV-SSSC doit traverser beaucoup plus de niveaux de logique sur son chemin critique qu'un Moustique, là encore c'est dû au grand nombre de portes logiques à traverser par les boîtes-S. Tandis que pour Moustique, lors du renouvellement de l'état interne, il suffit juste de traverser les fonctions g_0, g_1 ou g_2 qui ont chacune un chemin critique de 2 portes logiques seulement (2 XORs, ou 1 XOR + 1 NAND).
— Il faut également constater que le chemin critique pour LPV-SSSC est de 15 ns environ, ce qui veut donc dire que le circuit synthétisé ne respecte donc pas la contrainte imposée de 100 MHz (équivalente à un chemin critique de 10 ns). Le circuit généré travaillera donc plutôt autour de 65 MHz.
— Pour ce qui concerne la version masquée, c'est-à-dire incorporant des contre-mesures face aux attaques

par canaux auxiliaires, elles multiplient la surface de la partie combinatoire par un facteur d'environ 4, ce qui est assez important. Nous verrons dans le chapitre consacré aux protections face aux SCAs les raisons de ce surcoût prohibitif. Le chemin critique est cependant peu modifié.
— Un autre résultat intéressant montre que Moustique lui-même ne peut pas être considéré comme "léger" : sa surface dépasse largement les 3000 GEs, borne maximale souvent considérée par la littérature déterminant si un algorithme cryptographique est léger ou non.
— Avec cette même borne, en revanche, Grain et Trivium peuvent bien être considérés comme légers. C'est d'ailleurs l'avis général de la communauté cryptographique.

5.4 Conclusion du chapitre

Cette section nous a permis de décrire nos résultats d'implémentation de notre algorithme LPV-SSSC. Nous avons également pu nous comparer de manière juste et objective avec la concurrence. Cette étude est assez complète car étant réalisée à la fois sur technologies FPGA et ASIC, et avec plusieurs variantes de l'algorithme. En résumé, malgré un surcout matériel qui peut sembler rédhibitoire comparé à ses concurrents directs, tels Moustique, il faut cependant noter que sur le FPGA que nous avons considéré, LPV-SSSC produit un débit de traitement de l'information plus important, et un temps de resynchronisation nettement plus court. LPV-SSSC semble donc plus adapté sur FPGA Spartan-6 que Moustique dans des environnements fortement contraints tels les télécommunications ou encore les ICS/SCADA. Ce constat est nettement moins flatteur sur ASIC.

Il faut toutefois nuancer les écarts de performance défavorables pour LPV-SSSC de par l'argument suivant : Moustique est cassé. Le consortium a pris le pari d'adopter un nouveau paradigme de conception de SSSC, qui est intrinsèquement plus couteux que celui utilisé par Moustique, mais qui sera beaucoup plus résistant à l'avenir à la cryptanalyse que son défunt concurrent. L'écart de performances peut donc être vu comme "le prix à payer" afin d'obtenir un SSSC sécurisé.

6 Analyse de Corrélations sur l'Alimentation

6.1 Introduction du chapitre

Dans les chapitres 4 et 5, nous avons détaillé nos résultats d'implémentation en logiciel et matériel de plusieurs versions de notre algorithme SSSC innovant. Lorsqu'on évoque les implémentations d'algorithmes cryptographiques, nous devons également prendre en compte les attaques physiques sur composant. Ce chapitre 6 résume nos résultats d'analyse par canaux auxiliaires de notre algorithme SSSC sur carte à puce. Au chapitre 7, nous détaillerons des méthodologies de protection contre ce type d'attaques. Puis, au chapitre 8, nous regarderons l'impact d'un autre type d'attaque physique : les attaques dites actives, ou par injection de fautes.

L'analyse de corrélation sur l'alimentation (*Correlation Power Analysis*, CPA) est une méthode pour retrouver des données secrètes enfouies dans le silicium d'un composant électronique par mesure de la consommation pendant l'exécution d'un calcul qui invoque ces données. Cette méthode a été proposée en 2004 par Brier *et al.* [49].

Cette attaque fait suite aux travaux de Kocher *et al.* qui, dès 1999, ont proposé des attaques par mesure de la consommation : SPA (*Simple Power Analysis*) et DPA (*Differential Power Analysis*) [50].

La présentation originale de la CPA s'appuyait sur une analyse statistique. La présentation faite ici repose sur l'analyse de Fourier et est de nature plus algébrique. Nous verrons par la suite que ces travaux ont une portée plus large que les seuls SSSC, et peuvent ainsi concerner la plupart des algorithmes de chiffrement symétriques.

6.2 Préliminaires d'analyse spectrale

On rappelle avant tout, des notions sur l'analyse de Fourier nécessaires à la modélisation de l'attaque. Cette attaque exploite la consommation de courant d'un dispositif électronique qui traite des données binaires. Ainsi on modélise la consommation de courant comme une fonction réelle à valeurs sur l'ensemble des mots binaires.

On considère donc l'espace Φ des fonctions à valeurs réelles sur l'ensemble des mots binaires de taille n :

$$\Phi = \left\{ \varphi : \{0,1\}^n \to \mathbb{R} \right\}$$

Pour deux fonctions φ et ψ dans Φ, on définit le produit scalaire de φ et ψ par :

$$\langle \varphi, \psi \rangle = \sum_{x \in \{0,1\}^n} \varphi(x)\psi(x).$$

Ce produit scalaire est une forme bilinéaire symétrique qui confère à l'espace $\mathbf{\Phi}$ une structure d'espace vectoriel euclidien de dimension 2^n sur $\mathbb{R}$.

La norme associée à ce produit scalaire est définie par :

$$\|\varphi\| = \sqrt{\langle \varphi, \varphi \rangle} = \sqrt{\sum_{x \in \{0,1\}^n} \varphi(x)^2}$$

La valeur de la norme d'une fonction φ de $\mathbf{\Phi}$ s'appelle l'*énergie* de φ.

La notion de norme ou d'énergie sera essentielle plus tard pour pouvoir modéliser l'attaque CPA à partir d'une approche spectrale. On va donc définir la transformée de Fourier sur laquelle se base cette approche spectrale.

Définition 6.2.1 (Transformée de Fourier)**.** *La transformée de Fourier d'une fonction φ de $\mathbf{\Phi}$ est la fonction à valeurs réelles définie sur $\{0,1\}^n$ par :*

$$\widehat{f}(u) = \frac{1}{\sqrt{2^n}} \sum_{x \in \{0,1\}^n} f(x)(-1)^{u \cdot x}$$

La transformation de Fourier exprime un changement de base. Il s'agit donc d'une transformation linéaire vérifiant :

$$\begin{aligned} &\forall \varphi, \psi \in \mathbf{\Phi},\ \widehat{\varphi + \psi} = \widehat{\varphi} + \widehat{\psi} \\ &\forall \varphi \in \mathbf{\Phi},\ \forall \lambda \in \mathbb{R},\ \widehat{\lambda\varphi} = \lambda\widehat{\varphi} \end{aligned} \tag{6.1}$$

On a aussi l'égalité de Parseval qui exprime la loi de conservation de l'énergie :

Proposition 6.2.1 (Égalité de Parseval)**.** *Pour toute fonction φ de $\mathbf{\Phi}$, on a :*

$$\|\varphi\| = \|\widehat{\varphi}\|.$$

6.3 Modélisation de l'attaque

6.3.1 Principes physiques

Les circuits électroniques qui réalisent les calculs dans les processeurs sont réalisés dans une technologie appelée CMOS (*Complementary Metal Oxyde Silicium*). La caractéristique principale de cette technologie

réside dans l'étage de sortie des portes logiques qui est constitué de deux transistors à effet de champ de polarité opposée montés de manière symétrique de telle sorte que lorsque l'un est passant, l'autre est bloquant (architecture *push-pull*). Cette architecture permet de réaliser des transitions très rapides (voir Figure 6.1).

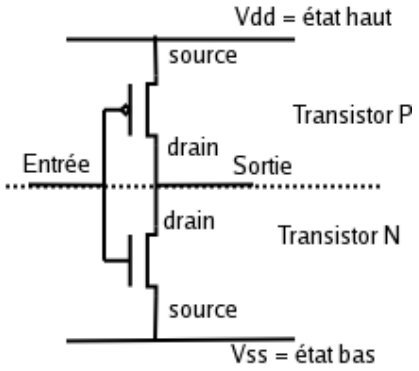


FIGURE 6.1 – Technologie CMOS

Il en résulte le principe de consommation suivant :

Hypothèse 1. *La consommation d'un circuit CMOS est proportionnelle au nombre de portes logiques qui transitent d'un état à un autre.*

6.3.2 Cible de l'attaque

La plupart des algorithmes de chiffrement en cryptographie symétrique comme le DES [51] ou encore l'AES [52] sont basés sur des successions de calculs de fonctions linéaires et non linéaires en plusieurs tours. Ces fonctions non linéaires sont souvent implémentées sous forme de boîtes-S définies sur $\{0,1\}^n$ vers $\{0,1\}^m$. En plus, ces fonctions réalisent un XOR entre une valeur $d \in \{0,1\}^n$ connue par l'attaquant et une valeur $k^\star \in \{0,1\}^n$ qui correspond à une sous-clé secrète et inconnue par l'attaquant. Cette fonction retourne une quantité $y \in \{0,1\}^m$:

$$y = f(d + k^\star).$$

Il existe différentes approches pour retrouver la sous-clé secrète $k^\star$ en réalisant une attaque CPA. Une des approches les plus classiques est décrite dans [53] et est basée sur le calcul de coefficient de corrélation de Pearson.

6.3.3 Description de l'approche générale d'une attaque CPA

L'approche consiste à réaliser l'attaque en 5 étapes comme illustrée par la Figure 6.2.

Étape 1 : choix des valeurs intermédiaires et des valeurs hypothétiques des sous clés et exécution de l'algorithme

Étape 2 : mesure de la consommation du courant du dispositif et enregistrement des traces

Étape 3 : calcul des valeurs hypothétiques à partir des valeurs intermédiaires

Étape 4 : correspondance des valeurs hypothétiques aux valeurs de consommation réelle

Étape 5 : corrélation des valeurs de consommation hypothétiques aux valeurs de consommation réelle (traces)

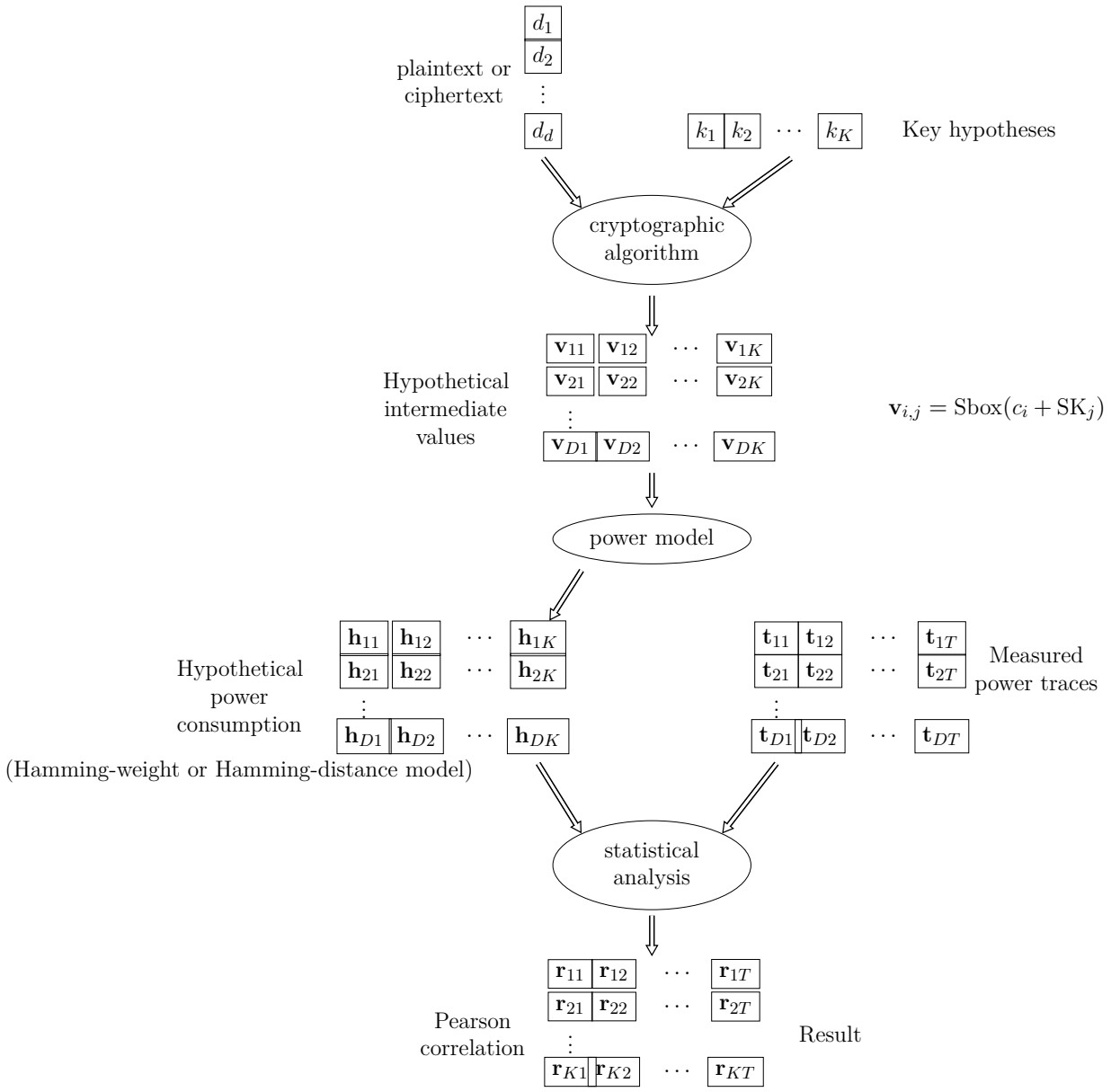


FIGURE 6.2 – Schéma des différentes étapes d'une attaque CPA.

À l'**Étape 1**, l'attaquant choisit D valeurs aléatoires $(d_1, d_2, \ldots, d_D)$ (clairs ou chiffrés) qui seront utilisés comme entrée de l'algorithme exécuté. Il choisit également les valeurs hypothétiques de sous-clés $k_1, \ldots, k_K$ qui seront corrélées à la valeur secrète réelle $k^\star$.

À l'**Étape 2**, à partir d'un oscilloscope, l'attaquant enregistre la consommation de courant ou "traces" associées à chaque valeur aléatoire d_i. Chaque trace est constituée de N échantillons notée $t_{i,j}$, $i = 1, \ldots, D$, $j = 1, ..., N$. Ce qui donne une matrice $T = (t_{i,j})$.

À l'**Étape 3**, l'attaquant calcule pour chaque valeur aléatoire d_i et chaque valeur hypothétique de sous-clé k_j, des valeurs hypothétiques $y_{i,j} = f(d_i + k_j)$.

À l'**Étape 4**, l'attaquant calcule des valeurs hypothétiques de consommation $h_{i,j}$ en utilisant soit le modèle du poids de Hamming soit le modèle de la distance de Hamming, avec $h_{i,j} = \mathrm{HW}(f(d_i + k_j))$ dans le cas du modèle du poids de Hamming ou $h_{i,j} = \mathrm{HD}(f(d_i + k_j))$ dans le cas du modèle de la distance de Hamming. Ceci donne une matrice $H = (h_{i,j})$.

Enfin à l'**Étape 5**, l'attaquant effectue une corrélation colonne par colonne entre les matrices H et T en appliquant la formule de corrélation de Pearson (6.2).

On rappelle que pour deux variables aléatoires $X = (X_1, \ldots, X_D)$ et $Y = (Y_1, \ldots, Y_D)$, la formule de corrélation de Pearson est donnée par :

$$\rho(X, Y) = \frac{\sum_{i=1}^{D}(X_i - \bar{X}) \cdot (Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{D}(X_i - \bar{X})^2 \cdot \sum_{i=1}^{D}(Y_i - \bar{Y})^2}} \tag{6.2}$$

où $\bar{X}$ désigne la moyenne de la variable aléatoire X.

Le coefficient de corrélation ainsi calculé à l'**Étape 5** est donc donné par $r_{i,j} = \rho(H_{.,i}, T_{.,j}),\ i = 1, \ldots, K, j = 1, \ldots, N$, où $H_{.,i}$ et $T_{.,j}$ désignent respectivement les colonnes i de H et les colonnes j de T.

L'approche spectrale présentée ici dans le cadre de la réalisation de l'attaque CPA permet de retrouver la valeur de $k^\star$ en calculant la transformée de Fourier de la consommation réelle notée φ et la consommation hypothétique notée g que l'on obtient à partir d'un modèle. On décrit en détail l'approche dans le paragraphe suivant.

6.3.4 Description de l'approche spectrale

Comme dans le cas général, l'attaque consiste à réaliser les opérations de chiffrement sur des entrées $d_1, d_2, \ldots, d_D$ choisies par l'attaquant puis à mesurer la consommation de courant pendant le calcul et à la fin à retrouver la valeur de la sous-clé $k^\star$.

En appliquant l'hypothèse 1, la consommation de courant mesurée lors du calcul de la boîte-S, pour une entrée d, est proportionnelle à la quantité $\varphi(d)$ donnée par :

$$\varphi(d) = \sum_{i=1}^{m} f_i(d + k^\star) + \varepsilon(d) + C, \tag{6.3}$$

ici :

- le premier terme est le modèle de fuite, c'est-à-dire le poids de Hamming de $f(d + k^\star)$, où f_i correspond à la i^{e} composante booléenne de la fonction f.
- le second terme noté $\varepsilon(d)$ correspond au bruit. Il provient des erreurs liées à la mesure de consommation ainsi que l'imprécision du modèle de fuite.

— le terme constant C correspond à la consommation du composant électronique qui ne dépend pas de la valeur d.

Notons par g la fonction $d \mapsto \sum_{i=1}^{m} f_i(d)$. L'Equation (6.3) peut être alors réécrite en :

$$\varphi(d) = g(d + k^\star) + \varepsilon(d) + C.$$

Ainsi, retrouver la valeur de la sous clé $k^\star$ revient à retrouver la valeur k qui minimise l'erreur donnée par :

$$\varepsilon_k(d) = \varphi(d) - g(d + k) - C. \tag{6.4}$$

Mieux, on essaye de minimiser l'erreur quadratique c'est-à-dire la valeur :

$$E(k)^2 = \|\varepsilon_k\|^2 = \sum_{d \in \{0,1\}^n} \varepsilon_k(d)^2.$$

A partir de l'égalité de Parseval (6.2.1), trouver la valeur de $k^\star$ équivaut à trouver la valeur de k qui est solution de :

$$\arg\min_k E(k)^2 = \sum_{u \in \{0,1\}^n} \widehat{\varepsilon}_k(u)^2 \tag{6.5}$$

À partir des propriétés sur la transformée de Fourier d'une fonction énumérées plus haut, l'Equation (6.5) s'exprime :

$$E(k)^2 = \sum_{u \in \{0,1\}^n} \widehat{\varphi}(u)^2 + \sum_{u \in \{0,1\}^n} \widehat{g}(u)^2 - 2 \sum_{u \in \{0,1\}^n} \widehat{\varphi}(u)\widehat{g}(u)(-1)^{u \cdot k} \tag{6.6}$$

Et donc finalement, trouver la valeur de k qui minimise l'énergie de l'erreur équivaut à trouver la valeur de k qui maximise la fonction $F(k)$ suivante :

$$F(k) = \sum_{u \in \{0,1\}^n} \widehat{\varphi}^\star(u)\widehat{g}^\star(u)(-1)^{u \cdot k}. \tag{6.7}$$

6.3.5 Estimation de la fiabilité de la valeur retrouvée

Lorsqu'on connait avec précision l'instant t où le calcul de $f(d + k^\star)$ est effectué alors la valeur de k qui est retrouvée à cet instant correspond exactement à la valeur de $k^\star$. Lorsque l'instant t n'est pas connu, on admet quand même qu'on connait un intervalle de temps Δ_t où le calcul est effectué. On estime donc la fiabilité de toutes les valeurs k (calculées en maximisant la fonction F de l'Equation (6.7)) en calculant le coefficient $r_t(k)$ suivant, pour tout instant t de Δ_t et pour tout valeur k retrouvée à cet instant t :

$$r_t(k) = \frac{F_t(k)}{\|\widehat{\varphi}_t\| \cdot \|\widehat{g}\|}. \tag{6.8}$$

6.4 Résultats expérimentaux

Le banc de mesure pour expérimenter l'attaque comprend une carte ATMega 163, muni d'un processeur AVR 8-bit. Cette carte a été programmée pour exécuter des opérations de la forme $f(d_i + k_i)$, où f est une boîte-S (par exemple celle de l'AES), k_i est une clé secrète de 4 ou 8 bits introduite préalablement dans la carte, les valeurs d_i sont des paramètres de 4 ou 8 bits.

6.4.1 Banc de test

Le banc de test (voir Figure 6.3) comprend une carte à puce, un lecteur de carte et un oscilloscope. L'oscilloscope est un Picoscope 5444b d'une bande passante de 200 MHz à un taux d'échantillonnage de 1 GS par seconde. Les broches de la carte à puce sont connectées à l'oscilloscope via un adaptateur (cf. Figure 6.4).

FIGURE 6.3 – Banc de test.

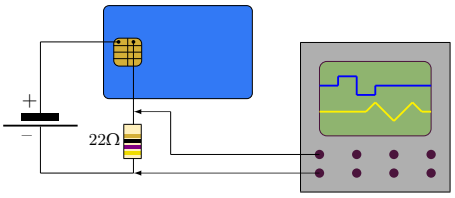


FIGURE 6.4 – Schéma de mesure de consommation.

6.4.2 Application de l'attaque sur une simple boîte-S

6.4.2.1 Procédure

L'attaque consiste à retrouver quatre clés secrètes k_1, k_2, k_3 et k_4 dans un intervalle de temps où la carte calcule $f(d + k_1)$, $f(d + k_2)$, $f(d + k_3)$ et $f(d + k_4)$ pour 256 valeurs, d prenant les valeurs 0 à 255.

La Figure 6.5 illustre la consommation d'une trace et le signal de consommation correspondant. L'intervalle de temps Δ_t correspond au calcul des quatre opérations successives $f(d + k_i)$ $(i = 1, \ldots, 4)$. Les 256 traces de consommation, chacune de 40000 échantillons, sont enregistrées dans cet intervalle de temps. La Figure 6.6 est un grossissement de la fenêtre Δ_t.

6.4.2.2 Résultats expérimentaux

On a considéré deux différents scénarios d'attaque :

i) lorsque la carte effectue le XOR entre la clé secrète k et la donnée d, c'est-à-dire l'opération $d + k$,

ii) lorsque la carte calcule la sortie de la boîte-S, c'est-à-dire l'opération $f(d + k)$.

Les Figures 6.7 et 6.8 montrent la fiabilité des valeurs k retrouvées à travers les coefficients de corrélation $r_t(k)$ de l'Equation (6.8), pour chaque instant t. Ceci prouve le succès de l'attaque. En effet, les pics de corrélation correspondent aux instants où les opérations sont effectées avec les vraies valeurs de sous-clé k_i.

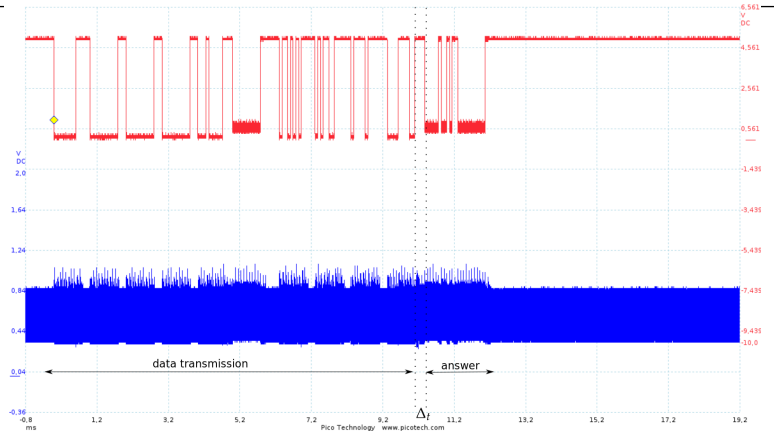


FIGURE 6.5 – Signal d'entrée sortie (en haut) et signal de consommation correspondant (en bas).

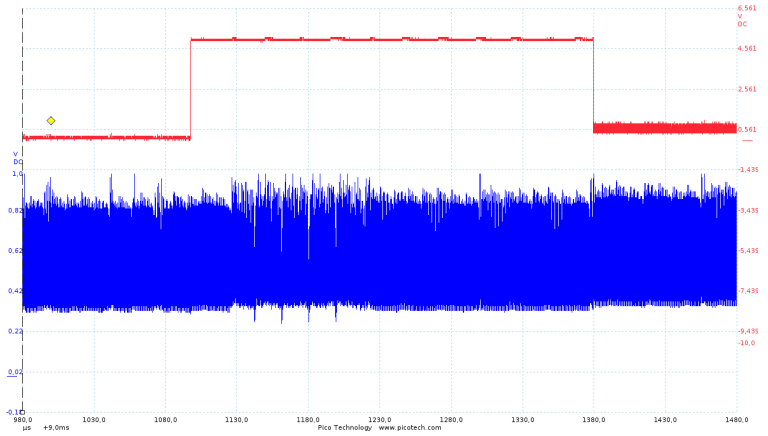


FIGURE 6.6 – Zoom sur l'intervalle de temps Δ_t. Le premier front descendant correspond à la réponse de la carte qui permet de synchroniser les traces.

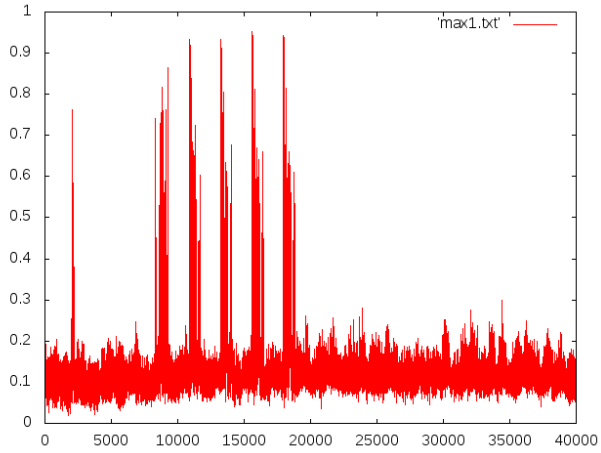


FIGURE 6.7 – Pics de corrélation correspondant au scénario *i*).

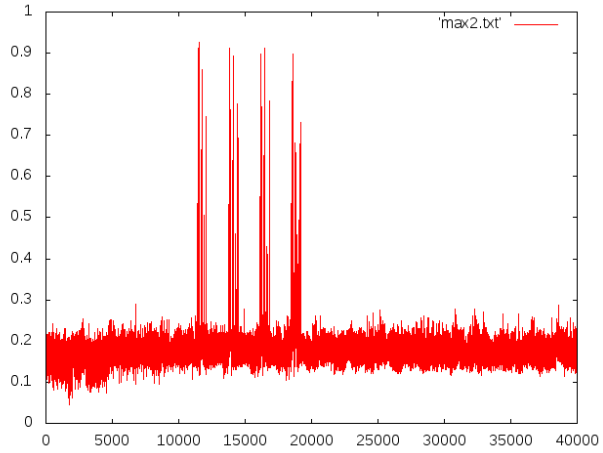


FIGURE 6.8 – Pics de corrélation correspondant au scénario *ii)*.

Remarque 6.4.1. *Il est à noter que dans le cas où l'attaque n'a pas abouti, suite à une mauvaise synchro-*

nisation des traces par exemple, on a pics de corrélation très faibles comme illustré par la Figure 6.9.

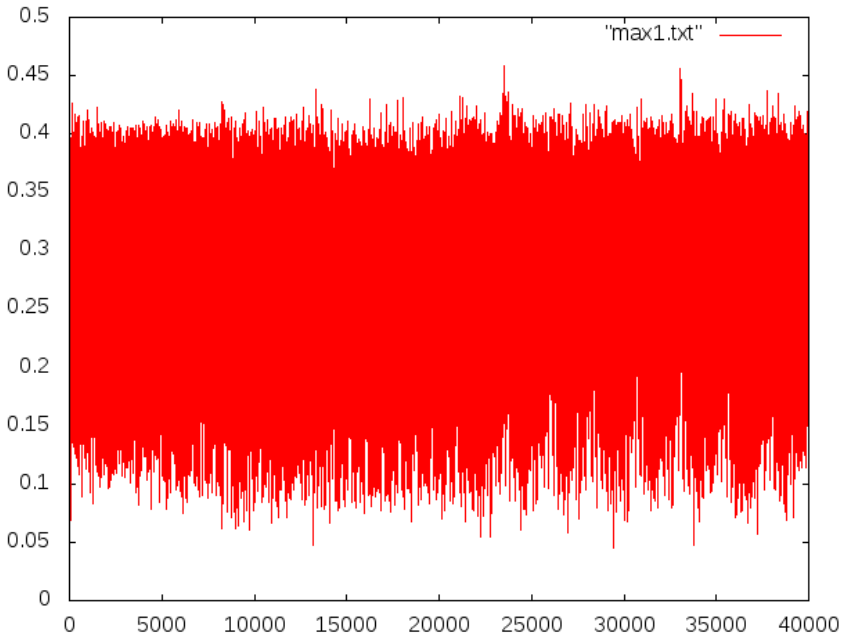


FIGURE 6.9 – Pics de corrélation lorsqu'aucune clé n'est retrouvée.

6.4.3 Application de l'attaque à l'algorithme THE CASCADE

On considère les équations de chiffrement et déchiffrement suivantes d'un SSSC obtenu à partir d'un système dynamique LPV plat. On note par m_k le texte clair à chiffrer à l'instant k et par $d_k = x_k[2]$ le chiffré correspondant. S correspond à la boîte-S utilisée dans l'algorithme, et SK_i, $i = 0, \ldots, 21$ aux sous-clés.

$$
\begin{array}{rcl}
x_{k+1}[0] & = & \mathrm{S}(d_k + \mathrm{SK}_0) \cdot x_k[0] + \mathrm{S}(d_k + \mathrm{SK}_1) \cdot x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_2) \cdot x_k[2] + \mathrm{S}(d_k + \mathrm{SK}_3) \cdot x_k[3] \\
 & & + \mathrm{S}(d_k + \mathrm{SK}_4) \cdot x_k[4] + \mathrm{S}(d_k + \mathrm{SK}_5) \cdot x_k[5] + \mathrm{S}(d_k + \mathrm{SK}_6) \cdot x_k[6] + m_k \\
x_{k+1}[1] & = & x_k[0] + x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_7) \cdot x_k[2] + x_k[6] \\
x_{k+1}[2] & = & x_k[1] + x_k[2] \\
x_{k+1}[3] & = & \mathrm{S}(d_k + \mathrm{SK}_8) \cdot x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_9) \cdot x_k[2] \\
x_{k+1}[4] & = & \mathrm{S}(d_k + \mathrm{SK}_{10}) \cdot x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_{11}) \cdot x_k[2] + \mathrm{S}(d_k + \mathrm{SK}_{12}) \cdot x_k[3] \\
x_{k+1}[5] & = & \mathrm{S}(d_k + \mathrm{SK}_{13}) \cdot x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_{14}) \cdot x_k[2] + \mathrm{S}(d_k + \mathrm{SK}_{15}) \cdot x_k[3] \\
 & & + \mathrm{S}(d_k + \mathrm{SK}_{16}) \cdot x_k[4] \\
x_{k+1}[6] & = & \mathrm{S}(d_k + \mathrm{SK}_{17}) \cdot x_k[1] + \mathrm{S}(d_k + \mathrm{SK}_{18}) \cdot x_k[2] + \mathrm{S}(d_k + \mathrm{SK}_{19}) \cdot x_k[3] \\
 & & + \mathrm{S}(d_k + \mathrm{SK}_{20}) \cdot x_k[4] + \mathrm{S}(d_k + \mathrm{SK}_{21}) \cdot x_k[5]
\end{array}
\tag{6.9}
$$

Pendant l'exécution de l'algorithme, le chiffré $x[2]$ obtenu à la première et seconde itération ne dépendent que de l'état initial. Ce n'est qu'à la troisième itéartion que le chiffré $x[2]$ est influencé par le clair, autrement dit le chiffré correspondant au clair m_k est donné par $x_{k+3}[2]$.

Ici les textes clairs et les chiffrés sont des quartets (symboles de 4 bits). Pour appliquer notre attaque basée sur le calcul de transformée de Fourier, on a donc besoin de 16 traces de consommation. Les mesures sont

donc réalisées pour des traces de 100000 échantillons pour couvrir en intégralité l'opération de chiffrement (6.9). Il faut bien noter que les entrées $d_i, i = 1, \ldots, D$ sont des textes clairs. Dans le cas de l'algorithme THE CASCADE, les entrées de la boîte-S ciblée sont par contre des chiffrés, contrairement à l'attaque sur une simple boîte-S.

Ainsi pour réaliser la mesure, on génère plusieurs textes aléatoires en appliquant la solution du collectionneur de coupons [54].

Les pics de corrélation obtenus sont illustrés par la Figure 6.10.

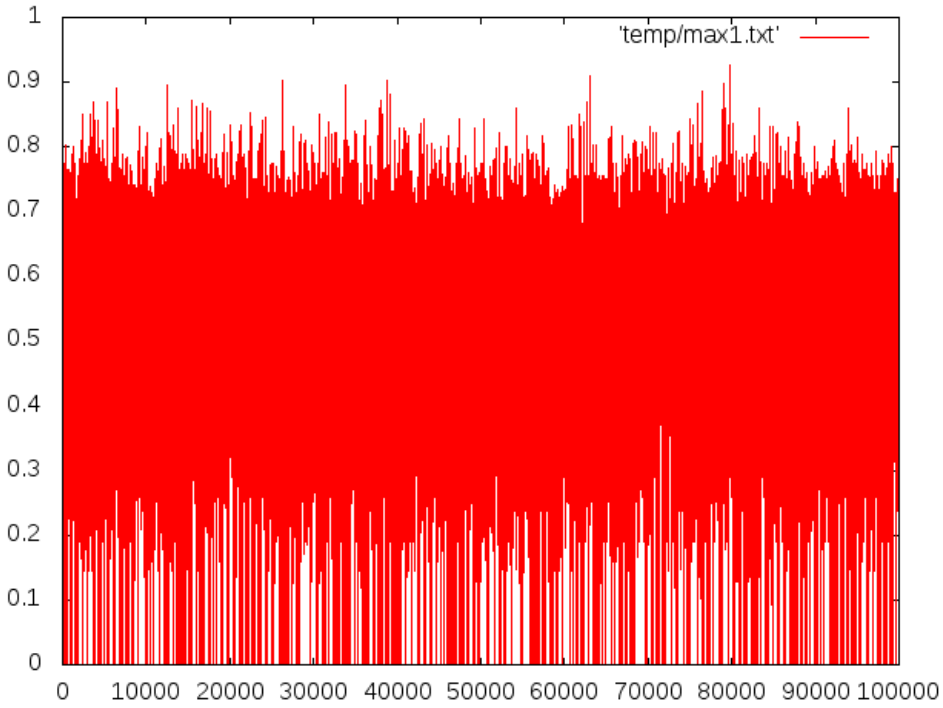


FIGURE 6.10 – Pics de corrélation en réalisant l'attaque CPA sur l'algorithme de chiffrement THE CASCADE.

Sur la figure, on observe des pics de corrélation élevés à des instants de calcul où les opérations sur la boîte-S n'interviennent pas. Et l'attaque retourne des valeurs erronées de clés secrètes à ces instants. Ces valeurs erronées sont probablement liées aux calculs intermédiaires comprenant des opérations de multiplication. En effet, ces opérations correspondent à des fonctions non linéaires qui peuvent bien correspondre à notre modèle pour des valeurs de clés secrètes qui ne sont pas nécessairement celles introduites dans le dispositif qui exécute l'algorithme. Notons également que ici, les opérations de multiplication ont été implémentées pour faciliter la synchronisation des traces, autrement cette synchronisation serait impossible à réaliser.

6.5 Conclusion du chapitre

L'approche spectrale appliquée à la CPA décrite ici de manière théorique, a été également appliquée sur un cas pratique d'implémentation de notre algorithme SSSC sur carte à puce. Cette attaque a une portée plus

importante que le seul cas des SSSCs, puisqu'elle peut s'appliquer sur n'importe quel algorithme symétrique pourvu qu'il utilise des fonctions non-linéaires telles des boîtes-S. Cela constitue un des résultats importants qui a d'ailleurs mené à une publication à date de livraison de ce livrable.

Cette étude pourrait être approfondie, par exemple en la testant sur des micro-contrôleurs plus complexes (16 bits, 32 bits) et sur plate-forme matérielle FPGA du type SAKURA-G, afin de vérifier si notre nouvelle formulation d'attaque CPA est toujours aussi efficace dans des environnements d'attaques plus ardus. Peut-être que pour ces plate-formes, utiliser le rayonnement électromagnétique de la puce, et pas seulement sa consommation électrique, pourrait être salutaire. Dans le même esprit, des tests sur des implémentations masquées (cf. Chapitre 7) et/ou désynchronisées pourraient être menées.

7 Résultats d'implémentations protégées contre les SCAs

7.1 Introduction du chapitre

Dans le chapitre précédent, nous avons montré que des attaques SCA étaient possibles sur les implémentations de notre algorithme SSSC innovant. Il nous faut donc maintenant estimer le coût des contre-mesures à ces attaques : c'est tout le but de ce chapitre.

Nous avons choisi de donner nos résultats d'implémentation sur SAKURA-G. Par ailleurs, nous avons également décidé d'étudier le surcout matériel de contre-mesures récentes qui étaient plutôt destinées au logiciel. Cette étude est également une première.

Après étude et lecture de plusieurs méthodes de masquage possibles pour augmenter la sécurité du système, notre choix s'est porté sur trois méthodes : la méthode *threshold* "classique", le "masquage grand ordre de tables de correspondance" (*Higher Order Masking of Look-up Tables*, HOMLUT), et la méthode utilisant une table (globale) de correspondance (*Global LUT*, GLUT).

7.2 La méthode *threshold* "classique"

Cette méthode repose sur la base de l'implémentation masquée dite *threshold*. On utilisera les notations suivantes dans cette partie : soit x une variable qu'on sépare en s partages additifs avec $x = \sum_i x_i$: on a donc le vecteur $\mathbf{x} = (x_1, x_2, ..., x_s)$. Pour implémenter $a = F(x, y, z, ...)$ de F_2^m dans F_2^n, la méthode TI (*Threshold Implementation*) doit satisfaire les trois propriétés suivantes :

- **La correctivité** : $a = F(x, y, z, ...) = \sum_i F_i(\mathbf{x, y, z, ...})$ pour tout **x, y, z, ...** satisfaisant $\sum_i x_i = x, \sum_i y_i = y, \sum_i z_i = z, ...$
- **La non-complétude** : Cette propriété est souvent traduite par "F_i doit être indépendante de $x_i, y_i, z_i, ...$". Chaque fonction est indépendante d'au moins un *share* des variables d'entrée x, y, z.
- **L'uniformité** : Pour tout $(a_1, a_2, ..., a_s)$ satisfaisant $\sum_i a_i = a$, le nombre de tuples $(\mathbf{x, y, z, ...}) \in F^{ms}$ pour lesquels $F_i(\mathbf{x, y, z, ...}) = a_i, 1 \leq j \leq s$, est égal à $2^{(s-1)(m-n)}$ fois le nombre de $(x, y, z, ...) \in F^m$ pour lesquels $a = F(x, y, z, ...)$. De plus, si F est une permutation de F^m, alors les fonctions F_i définissent ensemble une permutation de F^{ms}. Autrement dit, le partage préserve la distribution.

Pour une résistance aux attaques DPA d'ordre n, il faut pour chaque variable sensible utiliser $2n+1$ *shares*. Il faut de plus séparer la fonction S en trois sous-fonctions S_0, S_1 et S_2, telles que $S(x) = S_0(x) \oplus S_1(x) \oplus S_2(x)$.
Dans notre cas, on a $S(x) = 1/x + \alpha^2$ dans GF(16). On peut utiliser $S_0(x) = 1/x$, $S_1(x) = 1/x + a^2$ et $S_2(x) = 1/x + b^2$ avec $a^2 + b^2 = \alpha^2$.
Il faut alors générer 2 masques aléatoires de 4 bits par boîte-S, soit un total de 157×8 = **1256** bits d'aléa

par tour, ce qui représente une quantité importante de données de qualité à générer.

7.3 *Higher Order Masking of Look-up Tables*

La seconde méthode est décrite par Jean-Sébastien Coron [55] et est nommée “masquage grand ordre de tables de correspondance” (*Higher Order Masking of Look-up Tables*, HOMLUT).
Ce choix est basé sur la facilité d'utilisation de cette méthode puisqu'elle s'applique à n'importe quelle boîte-S utilisée, contrairement à de nombreuses autres méthodes décrites pour une implémentation masquée de l'AES ou du DES. De plus, pour une sécurité contre les attaques par analyses DPA d'ordre 2, c'est-à-dire combinant des traces de canaux auxiliaires exécutées à deux instants différents, elle ne nécessite que 5 partages aléatoires par variable critique. La figure 7.1 décrit le principe général de la méthode HOMLUT.

Xor operation. We consider a Xor operation $z = x \oplus y$. Taking as input the shares x_i and y_i such that $x = x_1 \oplus \cdots \oplus x_n$ and $y = y_1 \oplus \cdots \oplus y_n$, it suffices to compute the shares $z_i = x_i \oplus y_i$.

Linear operation. We consider a linear operation $y = f(x)$. Taking as input the shares x_i such that $x = x_1 \oplus \cdots \oplus x_n$, it suffices to compute the shares $y_i = f(x_i)$ separately.

Table Look-up. A table look-up $y = S(x)$ is computed using our previous Algorithm 1.

Input Encoding. Given x as input, we first encode x as $x_1 = x$ and $x_i = 0$ for $2 \leq i \leq n$. Secondly we let $(x_1, \ldots, x_n) \leftarrow \mathsf{RefreshMasks}(x_1, \ldots, x_n)$.

Output Decoding. Given $y_1, \ldots, y_n$ as input, we compute $y = y_1 \oplus \cdots \oplus y_n$ using Algorithm 3 below.

FIGURE 7.1 – *Higher Order Masking of Look-up Tables*

Les pseudo-codes des différents algorithmes utilisés sont décrits dans ce qui suit. La Figure 7.2 décrit l'algorithme qui permet de masquer une opération non-linéaire, dans notre cas, ce sera le calcul de la boîte-S.

Algorithm 1 Masked computation of $y = S(x)$

Input: $x_1, \ldots, x_n$ such that $x = x_1 \oplus \cdots \oplus x_n$
Output: $y_1, \ldots, y_n$ such that $y = S(x) = y_1 \oplus \cdots \oplus y_n$
1: **for all** $u \in \{0,1\}^k$ **do**
2: $\quad T(u) \leftarrow (S(u), 0, \ldots, 0) \in (\{0,1\}^{k'})^n$ ▷ $\oplus(T(u)) = S(u)$
3: **end for**
4: **for** $i = 1$ to $n-1$ **do**
5: $\quad$ **for all** $u \in \{0,1\}^k$ **do**
6: $\quad\quad$ **for** $j = 1$ **to** n **do** $T'(u)[j] \leftarrow T(u \oplus x_i)[j]$ ▷ $T'(u) \leftarrow T(u \oplus x_i)$
7: $\quad$ **end for**
8: $\quad$ **for all** $u \in \{0,1\}^k$ **do**
9: $\quad\quad T(u) \leftarrow \mathsf{RefreshMasks}(T'(u))$ ▷ $\oplus(T(u)) = S(u \oplus x_1 \oplus \cdots \oplus x_i)$
10: $\quad$ **end for**
11: **end for** ▷ $\oplus(T(u)) = S(u \oplus x_1 \oplus \cdots \oplus x_{n-1})$ for all $u \in \{0,1\}^k$.
12: $(y_1, \ldots, y_n) \leftarrow \mathsf{RefreshMasks}(T(x_n))$ ▷ $\oplus(T(x_n)) = S(x)$
13: **return** $y_1, \ldots, y_n$

FIGURE 7.2 – Calcul masqué

La Figure 7.3 décrit l'algorithme qui permet de mettre à jour les masques utilisés. Cette opération intervient plusieurs fois pour le masquage de la boîte-S.

Algorithm 2 RefreshMasks

Input: $z_1, \ldots, z_n$ such that $z = z_1 \oplus \cdots \oplus z_n$
Output: $z_1, \ldots, z_n$ such that $z = z_1 \oplus \cdots \oplus z_n$
1: **for** $j = 2$ to n **do**
2: $\quad tmp \leftarrow \{0,1\}^{k'}$
3: $\quad z_1 \leftarrow z_1 \oplus tmp$
4: $\quad z_j \leftarrow z_j \oplus tmp$
5: **end for**
6: **return** $z_1, \ldots, z_n$

FIGURE 7.3 – Mise à jour des masques

La Figure 7.4 décrit l'algorithme de recombinaison des partages générés. Une fois les opérations masquées, il faut démasquer l'ensemble pour continuer le chiffrement ou le déchiffrement.

Algorithm 3 Shares recombination

Input: $y_1, \ldots, y_n$
Output: y such that $y = y_1 \oplus \cdots \oplus y_n$
1: **for** $i = 1$ to n **do** $(y_1, \ldots, y_n) \leftarrow \mathsf{RefreshMasks}(y_1, \ldots, y_n)$
2: $c \leftarrow y_1$
3: **for** $i = 2$ to n **do** $c \leftarrow c \oplus y_i$
4: **return** c

FIGURE 7.4 – Recombinaison des partages masqués

Les différentes opérations ou variables à masquer dans notre algorithme sont celles utilisées pour la mise à jour des états internes du système. Elles consistent en un ensemble de OU-exclusifs entre les différentes cellules de la matrice pour chaque ligne. Les états internes mis à jour seront en clair.
On va donc ici chercher à masquer les boîtes-S avec les algorithmes ci-dessus. Ainsi toutes les cellules de la matrice seront masquées. Le point le plus important à prendre en compte pour réaliser ce masquage est le nombre de masques aléatoires à générer.
En effet, en prenant en compte la seconde version multipliée, il y a un total de 157 boîtes-S dans la matrice. Pour chaque boîte-S on va avoir besoin de 4 masques aléatoires pour masquer la variable critique **message** $\oplus$ **sous-clé**. Il faut de plus 4 masques aléatoires pour chaque opération *RefreshMasks* soit 4 opérations dans notre cas. Il en faut encore 4 pour l'algorithme *Shares recombination*.
Il faut au total quelques 5420 masques, soit **21680 bits aléatoires à générer pour chaque nouveau quartet de message entrant**. Et cela sans prendre en compte le masquage des XORs intervenant entre les boîtes-S qui demandent d'autres masques aléatoires.
On se rend donc compte que cette solution est difficilement utilisable pour le moment, car elle demande l'intégration d'un générateur aléatoire à très hautes performances au sein même du système. À ce titre, les oscillaters en anneaux à régime oscillant transitoire (*Transition Effect Ring Oscillator*, TERO [56]) peuvent être envisagés.

Pour arriver à un résultat plus probant, nous avons décidé de n'utiliser qu'un nombre réduit de masques en partant du principe qu'il suffisait que les boîtes-S d'une même ligne de la matrice soit masquée avec des masques différents. De plus, les masques ne sont pas modifiés à chaque quartet. Cette version n'est évidemment pas en accord avec les spécifications données dans l'article mais elles permettent de se faire une première idée de la place prise sur la carte. Le problème est que pour pouvoir créer cette architecture, nous avons dû implémenter un tableau contenant tous les masques. Les masques étant sur un quartet, il n'y a

donc qu'un total de 16 possibilités et le compilateur ne se prive pas de simplifier le tout pour donner des résultats qui ne correspondent pas à la réalité d'une telle implémentation si les masques étaient réellement générés aléatoirement. Cette solution ne convient donc pas. Nous avons donc néanmoins implémenté une version complète de cette version pour vérifier la taille prise sur notre FPGA cible.

7.4 Méthode GLUT

Pour finir, l'article [48] présente une méthode plus simple. Cette méthode ne peut être utilisée que pour résister à des attaques de premier ordre, mais au vu de la complexité de l'algorithme utilisé, ce masquage peut déjà convenir à nos applications.
Le principe va être d'utiliser une deuxième boîte-S que l'on appellera S'. Cette table doit tout d'abord être calculée pour les différentes valeurs du couple (valeur à masquer, masque), ce qui fait un tableau de 256 entrées.

Le masquage est décrit par la Figure 7.5 :

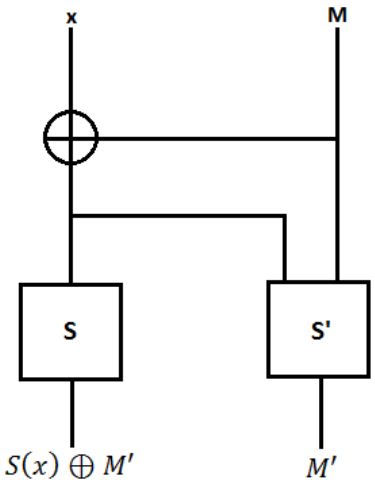


FIGURE 7.5 – Schéma de la méthode GLUT

Si l'on prend un masque de 4 bits par boîte-S, il faut un total de **628 bits d'aléa par tour**. Ce résultat est encore trop important. On peut alors penser à n'utiliser des masques différents que pour les boîtes-S qui sont sur une même ligne. On aurait alors au maximum besoin de 16 bits d'aléa pour masquer le chiffrement et de 132 dans le cas du déchiffrement.
Pour l'implémentation en VHDL on va choisir de ne démasquer les variables critiques que le plus tard possible.

7.5 Résultats d'implémentation des méthodes de masquage

Pour finir, voici en Figure 7.6 un tableau comparatif des implémentations des méthodes de masquage et la résistance qu'elles apportent. Nous sommes partis de notre composant de référence "Mul157" que nous avons masqué avec les 3 méthodes décrites précédemment. Nous avons utilisé comme plate-forme d'implémentation le FPGA SAKURA-G, dans les mêmes conditions que décrites dans le chapitre 5.

Version	Slices	LUTs	Flip-Flops	Freq. Max. (MHz)	Débit (Mbps)	Cycles Init.	Aléa (bits)	Résist. SCA
Mul157	691	2293	876	87	348	332	–	–
GLUT	1770	5911	876	58	232	332	628	1
Coron	764	2510	1766	74	296	332	5040	à définir
Coron (real)	3759	12199	9301	59	236	332	21680	2

FIGURE 7.6 – Résultats d'implémentation sur SAKURA-G

Voici les principales leçons à tirer de notre étude de performances :

- Pour ce qui concerne la méthode GLUT, on peut constater que la surface a été multipliée par 3 environ, ce surcout provenant essentiellement de la sur-utilisation des LUTs. Les tables ont donc été instanciées dans le FPGA sous la forme de LUTs, ce qui semble assez naturel. La fréquence maximale est tombée à 58 MHz, car le circuit obtenu est beaucoup plus complexe. Cette contre-mesure est donc très couteuse. La méthode nécessite 628 bits d'aléatoire par tour, ce qui est une quantité importante, mais beaucoup plus réalisable en pratique que les deux autres méthodes.
- La méthode Coron a été implémentée en deux versions : l'une réutilise des masques, l'autre renouvelle en permanence les masques utilisés.
- La première implémentation de la méthode Coron nécessite 5040 bits d'aléatoire par tour, ce qui est une quantité très importante. Le surcout provient globalement d'une sur-utilisation de Flip-Flops. Vu qu'initialement dans le composant Mul157, les *slices* utilisées n'instanciaient que très partiellement leurs Flip-Flops, la méthode masquée arrive à utiliser les Flip-Flops non-utilisées, ce qui fait que le surcout en *slices* est au final limité (environ 10%). La résistance aux SCAs de cette implémentation reste cependant à déterminer précisément, car la réutilisation des masques peut conduire à une perte de sécurité, passant de l'ordre 2 à 1.
- L'implémentation de la méthode Coron complète (cf. ligne **Coron (real)** Figure 7.6) est définitivement trop couteuse : le nombre de *slices* occupées est multiplié par un facteur d'environ 6, et le nombre de Flip-Flops d'environ 10. La quantité d'aléatoire requise par tour est lui-même énorme (21680), ce qui rend cette méthode très difficile à appliquer sur des cas industriels réels. En revanche, le grand avantage de cette méthode est qu'elle assure via une preuve de sécurité une sécurité d'ordre 2 vis-à-vis des attaques SCAs.

7.6 Conclusion du chapitre

Dans ce chapitre, nous avons détaillé le coût d'implémentation matérielle sur FPGA de contre-mesures SCAs appliquées sur notre algorithme SSSC innovant. Un comparatif de trois contre-mesures de l'état de l'art récent a été réalisé. Sans surprise, l'implémentation de ces contre-mesures induit un surcout important.

La méthode GLUT semble être la plus implémentable car le surcout reste relativement raisonnable, tandis que la quantité d'aléatoire à utiliser semble atteignable en pratique, à l'aide d'un générateur aléatoire de nombres très performant.

En revanche, la méthode de Coron semble très difficile à implémenter en pratique pour notre algorithme SSSC : le surcout d'implémentation est prohibitif, et surtout la quantité d'aléatoire à utiliser est très difficilement générable. Ce résultat ne semble pas sauter aux yeux immédiatement à la lecture de l'article concerné : cette contre-mesure semble donc être plutôt intéressante et élégante en théorie, mais difficilement praticable pour notre SSSC. Cette contre-mesure semble donc très adaptée à l'AES, mais moins sur des algorithmes cryptographiques plus complexes. La portée de ce constat dépasse donc le simple cadre des SSSCs et de THE CASCADE.

8 Attaque par injection de fautes en dimension réduite et complète

8.1 Introduction du chapitre

Dans le chapitre précédent, nous avons détaillé des architectures protégées contre les attaques physiques passives (SCAs) et fourni des résultats d'implémentation. Nous allons dans ce chapitre nous concentrer sur l'autre versant des attaques physiques : les attaques dites actives, ou par injection de fautes. Nous allons réaliser une analyse de sécurité sur un exemple jouet de notre algorithme SSSC afin de nous faciliter la récolte d'informations nous permettant de construire un algorithme SSSC en dimension complète intrinsèquement plus résistant face aux attaques en faute.

Dans ce chapitre seront présentées les différentes méthodes pratiques élaborées pour perturber le bon fonctionnement de circuits intégrés. Nous verrons ensuite les différents modèles de fautes généralement utilisés dans la littérature. Enfin, la dernière partie sera consacrée à certaines attaques spécifiques étudiées durant nos travaux.

8.2 Contexte des attaques en faute

Les perturbations des composants électroniques ont été étudiées pour la première fois dans les années 1970, lorsqu'il a été observé que les émissions de particules radioactives pouvaient provoquer des erreurs de fonctionnement sur les circuits intégrés. D'autres recherches ont par la suite été entreprises, notamment par les industries de l'aérospatiale, pour étudier les conséquences de telles perturbations sur les systèmes électroniques aéroportés.

C'est à partir des années 90 que l'on s'intéresse à l'exploitation de perturbations volontairement induites sur des dispositifs cryptographiques. L'analyse de fautes en cryptographie est utilisée pour la première fois en 1996 par Boneh *et al.*, qui montrent alors que la perturbation d'un cryptosystème permet de provoquer une fuite d'information critique. Leur article [57] propose différentes applications contre les implantations d'algorithmes de chiffrement à clé publique. Peu de temps après, Biham et Shamir [58] présentent des attaques contre des algorithmes de chiffrement symétrique, dont DES mais aussi des schémas plus généraux, partiellement inconnus. Par la suite, de nombreuses techniques sont généralisées et appliquées aux algorithmes de chiffrement asymétrique ou aux algorithmes de chiffrement par bloc, mais peu d'articles traitent encore des algorithmes de chiffrement par flot. Ce n'est qu'au début des années 2000 que commencent à paraître de tels articles. En 2004 notamment, Hoch et Shamir proposent des méthodes générales pour attaquer des algorithmes basés sur des registres à décalage à rétroaction linéaire [59].

8.2.1 Techniques d'injection de fautes

Dans cette section, nous détaillons les méthodes les plus couramment utilisées pour injecter des fautes [60] [61] [62]. Ces méthodes tentent de reproduire certains phénomènes naturels pouvant altérer le bon fonctionnement d'un dispositif, comme par exemple des variations de l'alimentation électrique, des interférences électromagnétiques ou encore une altération physique du circuit due à une usure à l'échelle nanoscopique.

Attaques par impulsions électriques. Le principe consiste à appliquer une variation brève sur la tension d'alimentation de la carte à puce ou sur la fréquence d'horloge.

Une première technique très simple est de sous-alimenter le dispositif. En effet, des erreurs peuvent apparaître en raison du ralentissement de la vitesse de transition des lignes du circuit consommant le plus d'énergie. Cette méthode ne demande que peu de moyens et connaissances techniques et peut facilement être mis en place sans laisser aucune trace sur le dispositif. Son principal inconvénient est son manque de précision dans le temps, il n'est en effet pas possible d'injecter une faute à un segment particulier de l'algorithme en cours.

Une amélioration consiste à provoquer soit des pics d'alimentation à des instants bien choisis, soit des micro-coupures temporaires. Il est alors possible de provoquer une mauvaise interprétation, ou même un saut d'instructions de la part du microprocesseur.

Une autre option pour un attaquant est de faire varier la fréquence d'horloge, pouvant ainsi perturber des lectures de données en mémoire comme l'exécution de micro-instructions. En effet, il est possible que le circuit tente de lire une valeur dans le bus de données trop tôt ou encore que le circuit exécute une instruction $n+1$ avant la fin de l'exécution de l'instruction n.

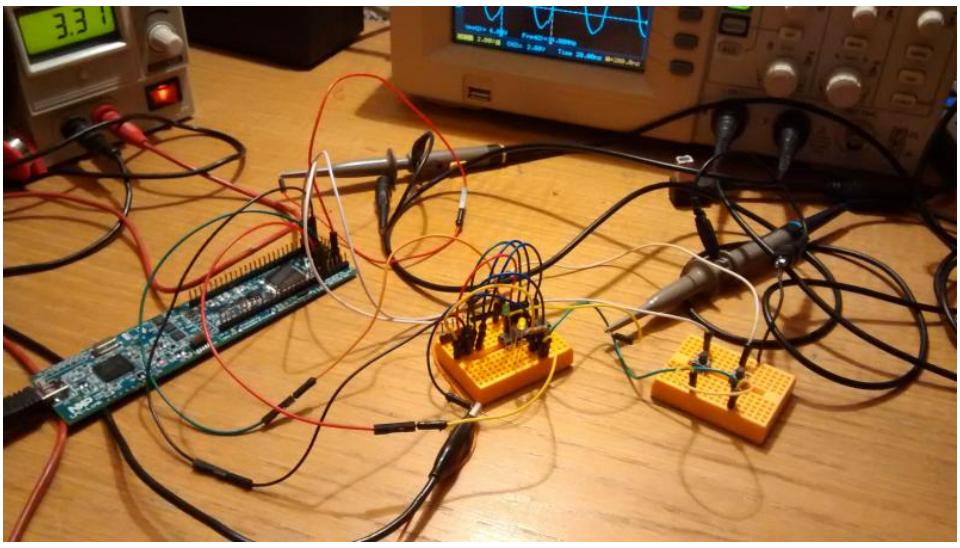


FIGURE 8.1 – Exemple de dispositif permettant de faire varier la fréquence d'horloge d'un microcontrôleur (Source : www.t4f.org)

Attaques thermiques. Le principe consiste à chauffer le composant de façon à ce qu'il se situe au-delà des températures de fonctionnement préconisées. Dans ce cas, deux effets sont principalement observés : modification aléatoire des données en RAM due à la chaleur et perturbations des opérations de lecture/écriture. En effet, ces deux opérations ont généralement des seuils de bon fonctionnement différents dans les

mémoires non volatiles. Ainsi, en plaçant le composant à une température entre ces deux seuils, des attaques sont possibles.

On peut par exemple utiliser une simple ampoule de 50W dirigée vers le composant et jouer sur la distance entre le composant et l'ampoule pour obtenir la température désirée. Cette méthode est donc à la portée de n'importe quel attaquant. L'inconvénient de cette technique est qu'elle peut corrompre une très grande quantité de données, en raison de son faible degré de précision. Il est également possible de détruire certaines parties trop sensibles du circuit en chauffant excessivement.

Attaques optiques. Il s'agit ici d'exploiter la sensibilité à la lumière des circuits électroniques, due à l'effet photoélectrique. Le courant provoqué par les photons peut être utilisé pour injecter des fautes si un circuit est exposé pendant un court instant à une source lumineuse puissante, telle un flash d'appareil photo (voir la figure 8.2) ou une lampe à UV. L'irradiation de la surface de silicium peut alors provoquer des effacements dans les cellules de mémoires EEPROM et Flash ou encore des erreurs dans les portes logiques illuminées. Cette méthode représente un moyen peu coûteux d'injection de fautes, elle nécessite cependant de mettre la puce à nu afin que le rayon lumineux émis puisse atteindre le composant et provoquer les perturbations escomptées. De nos jours, les attaques par illuminations sont sans doute les plus utilisées pour perturber les composants.

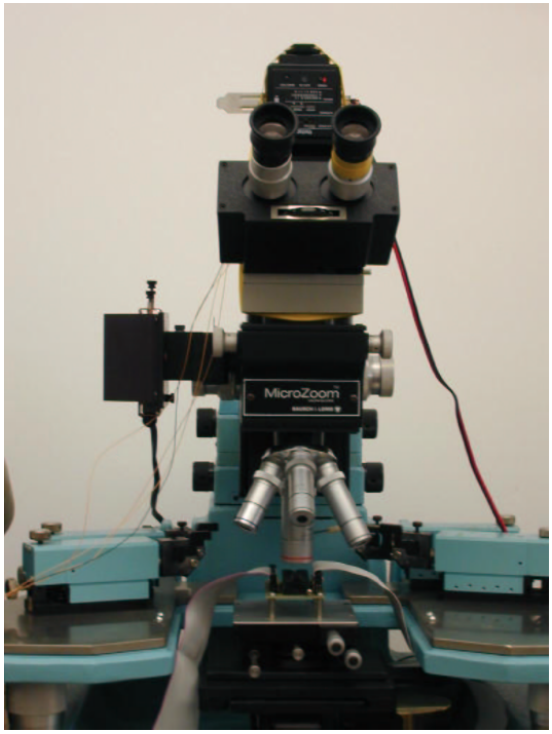


FIGURE 8.2 – Exemple de dispositif utilisant un flash d'appareil photo concentré à l'aide d'un microscope optique (Source : S. Skorobogatov, R. Anderson)

Une plus grande précision spatiale (zone de la puce à perturber) et temporelle (instant et durée d'une impulsion) peuvent être obtenues par l'utilisation de lasers. Ces attaques se montrent très efficaces, mais ne sont bien évidemment pas à la portée de tous du fait du coût élevé d'un banc laser.

Attaques par impulsions électromagnétiques. Une façon d'injecter des fautes sans avoir à être invasif sur le dispositif est de provoquer des interférences électromagnétiques à proximité du composant visé. Le champ électromagnétique crée des courants locaux au sein du circuit, connus sous le nom de courants de Foucault, pouvant provoquer des altérations temporaires au niveau des signaux, entraînant éventuellement l'utilisation d'une valeur erronée durant un calcul.

En pratique, il est possible d'utiliser un émetteur à étincelles (comme un allume-gaz piézoélectrique), en plaçant les deux extrémités entre lesquelles l'étincelle est produite à proximité de la surface du circuit. L'émetteur pourra être déclenché par un circuit connecté à l'horloge du composant attaqué.

Autres attaques. Il existe d'autres moyens d'induire une perturbation. Les rayons X, par exemple, peuvent aussi être employés, bien que cela soit peu commun. Cette méthode permet d'attaquer un dispositif sans avoir à extraire la puce.

Enfin, la microchirurgie par faisceau d'ions focalisés est la méthode d'injection de faute la plus précise, mais aussi la plus onéreuse. Elle permet à un attaquant de modifier la structure d'un circuit pour reconstruire des bus manquants ou couper des connexions électriques. On utilise généralement du gallium, qui permet une précision allant jusqu'à 0,135 nm.

8.2.2 Modèles de perturbation

8.2.2.1 Types de perturbations

On distingue deux grands types de perturbations : les perturbations éphémères (*transitoires*) et les perturbations destructives (*permanentes*). Les perturbations *permanentes* résultent d'une modification de la structure de la puce, qui peut alors concerner des variables en mémoire, du code contenu dans de la mémoire non volatile, ou encore le câblage du circuit.

Cependant, les perturbations peuvent aussi avoir des effets éphémères. On parle alors de perturbations *transitoires*. Ces perturbations ont pour effet de modifier temporairement l'exécution de code ou d'une opération. En particulier, le composant retrouve son comportement original après une réinitialisation, ou lorsque la source de la perturbation n'émet plus de signaux compromettants. Ce type de faute est certainement le plus étudié et le plus utilisé pour attaquer les implémentations d'algorithmes cryptographiques.

8.2.2.2 Modélisation des effets de perturbations

Un *modèle de faute* est un ensemble de propriétés d'une faute injectée. Il caractérise une attaque. Cette modélisation formalise les effets qu'il est possible d'obtenir sur un composant cryptographique. Le modèle doit, en outre, tenir compte des aptitudes de l'attaquant à contrôler les perturbations (localisation spatiale et temporelle, nombre de bits perturbés). Dans cette partie, nous allons présenter les modèles de fautes les plus utilisés dans la littérature.

Modification d'un bit

Bit aléatoire. Ce modèle de perturbation est le plus ancien, il a été introduit par Boneh, DeMillo et Lipton dans le premier article sur les attaques en fautes [57]. Il suppose que la faute affecte exactement un bit, dont la localisation est inconnue. Cette modification peut soit faire basculer la valeur de ce bit (*bit-flip fault model* en anglais) soit l'écraser pour la fixer à 0 ou à 1 (*stuck-at fault model*).

Bit choisi. Dans ce modèle plus contraint, on suppose que la faute affecte exactement un bit précis. Ce modèle suppose un adversaire particulièrement puissant, et est parfois considéré irréalisable en pratique. En effet, seule une grande expérience dans l'utilisation d'un laser permet d'injecter ce type de faute. Cependant, les modèles de fautes supposant le basculement d'un bit sont encore très largement utilisés pour attaquer les implantations d'algorithmes cryptographiques et particulièrement dans le cadre des algorithmes de chiffrement par flot [58].

Modification de quelques bits Un autre modèle suppose qu'un nombre limité de bits peut être affecté par une faute. Ce modèle prend en compte le fait que les composants actuels sauvegardent et chargent les données par bloc, dont la taille correspond aux spécificités de leur architecture. Cette taille est typiquement d'un octet, aussi considère-t-on souvent que les attaques supposant des fautes aléatoires sur un octet sont les plus réalistes, même s'il reste tout à fait possible de les étendre à de plus grandes tailles de bloc.

Plusieurs bits peuvent également être affectés par une faute visant un bit spécifique de la mémoire, mais dont les effets se propagent sur les bits voisins de manière non uniforme. Il reste cependant préférable de supposer une modification sur tout un octet, ce qui constitue un modèle universel de faute affectant un petit nombre de bits. Dans tous les cas, le nombre de bits touchés est supposé assez petit pour qu'un adversaire ne connaissant pas la localisation de ces bits puisse retrouver l'erreur induite par une recherche exhaustive.

Contrairement au modèle précédent, celui-ci est considéré comme réaliste car il reflète véritablement des effets de perturbations observés et permet d'exploiter des cas réels de perturbations. Il est particulièrement adapté aux perturbations intervenant pendant le stockage ou le chargement de données.

Perturbations multiples Un dernier aspect des modèles de faute concerne le nombre d'erreurs que l'adversaire peut injecter. On parle alors de l'*ordre* de l'attaque : une attaque du premier ordre suppose que l'attaquant ne peut induire qu'une seule erreur par exécution de l'algorithme visé. De même, dans une attaque de *second ordre*, l'adversaire pourra injecter deux erreurs par exécution, et ainsi de suite. Il existe plusieurs exemples d'attaques pratiques de second ordre dans la littérature. En 2007 notamment, Kim *et al.* proposent une attaque contre une implémentation protégée de RSA-CRT, en utilisant plusieurs impulsions électriques afin d'effectuer des sauts d'instructions à deux moments distincts de l'exécution [63].

8.2.3 Attaques classiques

Au cours de la première partie de notre étude sur les attaques en faute, nous nous sommes surtout concentrés sur les attaques en fautes qui concernaient des algorithmes de chiffrement par flot, en particulier RC4 et les algorithmes du portfolio eSTREAM, projet organisé en 2004 par le réseau ECRYPT (European Network of Excellence in Cryptology).

Si chaque attaque reste spécifique à la structure de l'algorithme concerné, la façon de procéder reste très similaire d'une attaque à l'autre. On distingue ainsi les phases suivantes :

— **Collecte de données.** L'attaquant obtient une première séquence chiffrée non perturbée, puis il injecte des fautes et obtient les séquences correspondantes.
— **Localisation des fautes.** L'attaquant analyse les différences entre la séquence chiffrée initiale et les séquences perturbées. Les motifs observés lui permettent de retrouver la position de la faute.
— **Constitution d'un système d'équations**. L'attaquant récolte des équations, généralement en fonction des bits de l'état interne, jusqu'à obtenir un système d'un rang [1] suffisamment grand.
— **Extraction de la clé.** Une fois le système résolu, l'attaquant connaît un état interne de l'algorithme. Il peut alors remonter dans son exécution jusqu'à l'état initial, qui dépend généralement de la clé.

Certaines attaques sont détaillées dans les sections suivantes.

8.2.3.1 Attaques contre RC4

Description de RC4 RC4 (*Rivest Cipher 4*) est un algorithme de chiffrement par flot conçu par Ronald Rivest en 1987. Bien que jamais publié officiellement, ses détails sont postés anonymement sur Internet en 1994.
RC4 prend en entrée une clé secrète K allant de 8 à 2048 bits. L'état interne se compose de :
— deux indices i et j, codés sur un octet
— un tableau S contenant 256 octets qui est une permutation des valeurs de 0 à 255.
Avant de générer les octets de la suite chiffrante, le contenu de S est initialisé par une procédure décrite par l'algorithme 8.2.3.1.

Algorithm 1 Initialisation de l'état interne de RC4 par la clé

Entrée : Clé secrète K de n octet(s)
Sortie : Etat interne S
pour $i = 0$ *à* 255 **faire**
  $S[i] \leftarrow i$

pour $i = 0$ *à* 255 **faire**
  $j \leftarrow (j + S[i] + K[i \bmod n]) \bmod 256$
  temp $\leftarrow S[i]$
  $S[i] \leftarrow S[j]$
  $S[j] \leftarrow$ temp

$i \leftarrow 0$
$j \leftarrow 0$

L'algorithme génère ensuite à chaque étape un octet qui est XOR-é à un octet de message clair pour former un symbole chiffré.

Attaques sur RC4 Dans la suite, nous allons voir trois attaques sur RC4 :

1. Le rang d'un système d'équations linéaires est le nombre d'équations que compte tout système échelonné équivalent. Dans le cas d'une matrice, le rang correspond au nombre maximal de vecteurs lignes (ou colonnes) linéairement indépendants.

Algorithm 2 Mise à jour de l'état interne de RC4

Entrée : Etat interne S
Sortie : Octet de sortie o
$i \leftarrow i + 1 \bmod 256$
$j \leftarrow j + S[i] \bmod 256$
temp $\leftarrow S[i]$
$S[i] \leftarrow S[j]$
$S[j] \leftarrow$ temp
retourner $o \leftarrow S[S[i] + S[j]]$

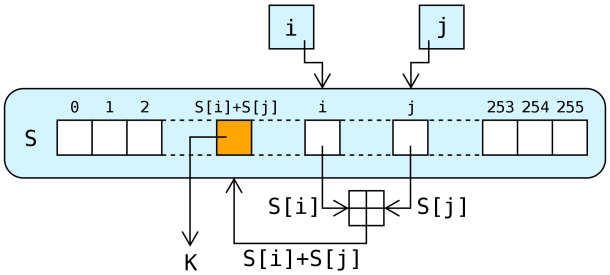


FIGURE 8.3 – Mise à jour de l'état interne de RC4

— La première a été proposée par Hoch et Shamir en 2004 [59]. Elle requiert en moyenne 2^{16} fautes et 2^{26} octets de suite chiffrante.
— La deuxième force l'algorithme à rentrer dans un état dit "impossible". Elle a été proposée en 2005 par Biham *et al.* [64]. Elle demande autant de fautes en moyenne que la première attaque mais le nombre d'octets à traiter est inférieur (2^{21}).
— La dernière attaque est une attaque différentielle, dont le nombre de fautes requis est de 2^{10} en moyenne. Elle est aussi décrite dans [64].

Attaque de Hoch et Shamir Cette attaque est l'une des premières attaques sur les algorithmes de chiffrement par flot. Ici, l'adversaire est supposé capable de modifier la valeur d'un octet de S juste après la phase d'initialisation (donc $i = j = 0$). L'attaque se déroule en trois étapes :

— Injection d'une faute dans S et génération d'une longue suite chiffrante (cette étape est répétée de nombreuses fois)
— Analyse des suites obtenues et génération d'équations en fonction des valeurs d'origine de S
— Résolution d'équations pour reconstruire S.

Après injection d'une faute, l'attaquant peut facilement déterminer le contenu de la case qu'il a fauté (mais pas l'indice de cette case). En effet, S contient toutes les valeurs possibles que peut prendre un octet. Ainsi, en fautant une case de S, l'attaquant remplace une des valeurs possibles, appelons-la a, par une autre déjà présente dans une autre case du registre, que l'on notera b. Dans ce cas, a n'apparaîtra donc jamais dans la suite chiffrante tandis que b apparaîtra en moyenne deux fois plus souvent. Les auteurs estiment qu'il faut une suite d'environ 10 000 octets pour identifier a et b.

L'étape suivante consiste à identifier les fautes sur $S[1]$, par observation du premier octet de sortie. En effet, si ce premier octet a été changé, nous nous trouvons dans l'un des trois cas suivants :

1. $S[1]$ a été fauté
2. $S[S[1]]$ a été fauté
3. $S[S[1] + S[S[1]]]$ a été fauté

Comme on connaît la valeur d'origine de $S[S[1] + S[S[1]]]$ (c'est le premier octet de la suite chiffrante non perturbée), on peut savoir si la faute a eu lieu à l'indice $S[1] + S[S[1]]$ en identifiant a et b. Si la faute a eu lieu à la case d'indice 1, *i.e.* $S[1]$ a été fauté, alors nécessairement, le deuxième octet de sortie $S[S[2] + S[S[1] + S[2]]]$ sera fauté, tandis que pour une faute sur la case d'indice $S[1]$, *i.e.* $S[S[1]]$ a été fauté, le deuxième octet de sortie reste inchangé avec une forte probabilité.

L'attaquant peut donc identifier les fautes en $S[1]$ ainsi que la valeur d'origine a et la nouvelle valeur b. Il sait donc que s'il parvient à fauter $S[1]$ après l'initialisation, le premier octet de sortie vaudra $S[b + S[b]]$. Il obtient ainsi pour chaque faute sur $S[1]$ une équation qu'il peut utiliser pour déterminer un octet de S.

Les auteurs ont estimé qu'il fallait injecter environ 2^{16} fautes sur S et analyser environ 2^{26} octets de sortie pour obtenir tout l'état interne de RC4.

Analyse des états dits "impossibles" de RC4 Cette attaque assez originale exploite les états impossibles de RC4 décrits par Finney *et al.* en 1994 [65], aussi appelés *états de Finney.*

Etat de Finney. On dit que l'état interne de RC4 se trouve dans un état de Finney lorsque :

$$j = i + 1 \text{ et } S[j] = 1 \tag{8.1}$$

Il est alors intéressant de remarquer qu'une fois un tel état atteint, i et j satisferont cette relation pour toute étape suivante, d'après l'algorithme 8.2.3.1. Mantin a également observé dans sa thèse [66] certaines propriétés de cet état impossible :

— Le contenu de S subit une permutation circulaire d'un octet toutes les 255 itérations.
— L'indice utilisé pour produire l'octet de sortie est le même toutes les 255 itérations.

Une conséquence de ces deux propriétés est alors que, dans de telles conditions, si on observe $255 \cdot 256$ octets de suite chiffrante, on obtient 255 copies entrelacées de l'état interne (ou du moins, une permutation circulaire de cet état interne).

Finney a montré qu'il était normalement impossible d'atteindre un tel état du fait de la procédure d'initialisation de RC4, qui remet i et j à 0 avant la phase de génération. Néanmoins, il est possible de forcer l'algorithme à rentrer dans un état de Finney au moyen d'une injection de faute.

Les auteurs ont donc supposé l'attaquant capable de fauter aléatoirement l'un des deux pointeurs i, j, à tout moment. Sous ce modèle, la probabilité qu'une faute provoque un état impossible est donc de 2^{-16} :

1. Si la faute touche i, la probabilité que $S[j] = 1$ est de 2^{-8}, et la probabilité que la nouvelle valeur de i soit $j - 1$ est aussi de 2^{-8}.
2. Si la faute touche j, la probabilité que $S[i + 1] = 1$ est de 2^{-8}, et la probabilité que la nouvelle valeur de i soit $i + 1$ est aussi de 2^{-8}.

Ainsi, on s'attend à tomber sur un état de Finney après injection de 2^{16} fautes environ.

Détection d'un état de Finney. L'intérêt de ce modèle d'attaque est qu'il est d'abord très simple d'identifier un état de Finney. Une approche naïve consisterait à observer $2 \cdot 256 \cdot 255$ octets de sortie postérieurs à l'injection de la faute, pour voir si nous nous trouvons bien dans un cycle de taille $256 \cdot 255$, ce qui demanderait donc de traiter environ 2^{17} octets. Afin de réduire cette quantité, les auteurs conseillent d'utiliser la fréquence de répétition des octets de sortie.

En effet, dans des conditions normales de fonctionnement, les octets de sortie sont choisis de manière pseudo-aléatoire au sein du registre S, ce qui d'après le paradoxe des anniversaires, donne une collision après environ 20 octets. En revanche, lorsque l'algorithme se trouve dans un état de Finney, il a été observé expérimentalement qu'une collision avait lieu après environ 80 octets. Selon les auteurs, si aucune collision n'a été observée sur les 30 octets de la suite chiffrante postérieurs à l'injection de la faute, alors on se trouve très probablement dans un état de Finney.

Reconstitution de l'état interne. Une fois que l'attaquant a obtenu une suite chiffrante résultant d'un état de Finney, il peut facilement reconstuire l'état interne de RC4 juste avant la perturbation. En effet, d'après les propriétés qui ont été mises en avant précédemment, il suffit d'observer $256 \cdot 255$ octets de sortie, et de prendre un des 255 flux entrelacés pour avoir une permutation circulaire de l'état interne. Pour aller plus vite, il est également possible de déduire certaines valeurs consécutives de l'état interne à partir de l'information donnée par l'ordre de sortie des octets.

L'attaquant obtient donc une permutation circulaire de l'état interne, il doit encore associer chaque valeur à sa véritable position dans S. Cela est possible, en exploitant le fait qu'il est facile d'identifier les cas où l'octet envoyé en sortie provient de la case d'indice i ou j, *i.e.* $S[i] + S[j] = i$ ou j. Par exemple, si $S[i] + S[j] = i$, alors l'octet produit vaut nécessairement 1. Une fois l'état interne à un instant t connu, il est simple de retrouver les valeurs de i et j puis de revenir à l'état initial, puis d'extraire la clé.

Cette attaque nécessite 2^{16} fautes, ce qui est autant que l'attaque de Hoch *et al.* [59]. En revanche, elle demande moins d'octets à analyser (2^{21} contre 2^{26}).

Attaque différentielle par fautes sur RC4 Cette attaque, plus standard, consiste à injecter des fautes à des positions précises de l'état interne, puis de déduire des suites chiffrantes perturbées la valeur des cases touchées. Elle utilise un nombre de fautes moins important que dans l'attaque précédente, mais le modèle de faute est assez contraint, car il demande un contrôle sur le moment d'injection et sur la position de la faute.

L'attaque exploite le fait qu'à chaque étape, seuls trois octets de S interviennent dans la sortie ($S[i]$, $S[j]$, $S[S[i] + S[j]]$). Elle contient une première phase de collection de données :

1. Obtention d'une suite chiffrante non perturbée :
   — On introduit la clé, inconnue ;
   — On itère 256 fois, en conservant le flux de sortie R pour une analyse ultérieure.
2. Pour l allant de 0 à 255 :
   — On introduit la clé, inconnue ;
   — On injecte une faute dans $S[l]$;

— On itère 30 fois, en conservant le flux de sortie, noté O_l, pour une analyse ultérieure.

Le premier octet de O_l est le même que celui de R pour toute valeur de l, sauf pour trois d'entre elles. Elles correspondent aux valeurs de i, j et $S[i] + S[j]$ mais nous ne sommes pas en mesure de toutes les distinguer. La valeur de i est toujours connue (sa valeur est uniquement incrémentée à chaque étape, donc ici, $i = 1$), il nous faut donc identifier j et $S[i] + S[j]$.

L'idée est de procéder ainsi pour tous les octets de R. On utilisera à chaque fois les valeurs précédemment trouvées pour déduire les nouvelles valeurs de j et $S[i] + S[j]$. Les auteurs ont appelé cette méthode *Cascade Guessing*.

Identification de j À toute position dans le flux R, on aura au moins trois valeurs de l pour laquelle l'octet sera différent de l'octet considéré. On remarque que si on soustrait à la valeur supposée de j le j trouvé pour le tour précédent, on obtient la valeur actuelle de $S[i]$. i étant toujours connu, il suffit de voir si cette valeur est dans le tableau pour savoir si la valeur supposée de j est la bonne.

Identification de $S[i] + S[j]$ L'autre valeur à identifier est celle de $S[i] + S[j]$. Connaissant la vraie valeur de l'octet de sortie (on dispose en effet du flux non perturbé R), on sait sur quelle valeur pointe $S[i] + S[j]$, ce qui nous permet également de tester nos hypothèses.

Les auteurs ont estimé qu'en observant environ 20 octets (donc près de 7 octets pour chaque flux de sortie), il était possible d'éliminer un candidat. On trouve ainsi dans un premier temps un nombre important d'octets de l'état interne avec une faible complexité.

Le problème est que plus on fait d'itérations, plus l'identification des trois valeurs devient difficile, à cause des effets de la propagation. Les auteurs proposent alors de se munir d'autres ensembles, issus de fautes injectées à des instants différents.

Une grande partie de l'état interne peut ainsi être retrouvée. Pour en obtenir la totalité, des méthodes d'analyses classiques peuvent ensuite être utilisées. Cette attaque utilise moins de 2^{16} octets de suite chiffrante.

8.2.3.2 Attaque différentielle sur Trivium

Description de Trivium Trivium est un algorithme de chiffrement par flot synchrone proposé par De Cannière *et al.* [67] et qui fait partie du portfolio eSTREAM. Il prend en entrée une clé secrète $K = (K_1, ..., K_{80})$ de 80 bits ainsi qu'un vecteur d'initialisation IV de 80 bits et utilise uniquement des opérations logiques (AND et XOR). L'état interne de Trivium, noté IS_t à l'instant t, se compose de 288 bits $(s_1, ..., s_{288})$, répartis sur 3 NLFSRs.

Une première étape initialise IS à partir de la clé secrète et du vecteur d'initialisation. Ensuite, l'algorithme met à jour l'état interne et produit à chaque tour un bit z_i qui est combiné à un bit de message clair via un XOR pour donner un bit chiffré.

Les fonctions d'initialisation et de mise à jour d'état interne sont décrites ci-dessous.

Algorithm 3 Initialisation de Trivium

Entrée : Clé secrète $K = (K_1, ..., K_{80})$, vecteur d'initialisation $IV = (IV_1, ..., IV_{80})$
Sortie : État interne $(s_1, ..., s_{288})$
$(s_1, ..., s_{93}) \leftarrow (K_1, ..., K_{80}, 0, ..., 0)$
$(s_{94}, ..., s_{177}) \leftarrow (IV_1, ..., IV_{80}, 0, ..., 0)$
$(s_{178}, ..., s_{288}) \leftarrow (0, ..., 0, 1, 1, 1)$ **pour** $i = 0$ *à* $4 \cdot 288$ **faire**
  $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
  $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
  $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
  $(s_1, ..., s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$
  $(s_{94}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$
  $(s_{178}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$

Algorithm 4 Génération de N bits de la suite chiffrante

Entrée : État interne de Trivium
Sortie : Suite chiffrante $\{z_i\}_{i=1}^{N}$
pour $i = 1$ *à* N **faire**
  $z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$
  $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
  $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
  $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
  $(s_1, ..., s_{93}) \leftarrow (t_3, s_1, ..., s_{92})$
  $(s_{94}, ..., s_{177}) \leftarrow (t_1, s_{94}, ..., s_{176})$
  $(s_{178}, ..., s_{288}) \leftarrow (t_2, s_{178}, ..., s_{287})$

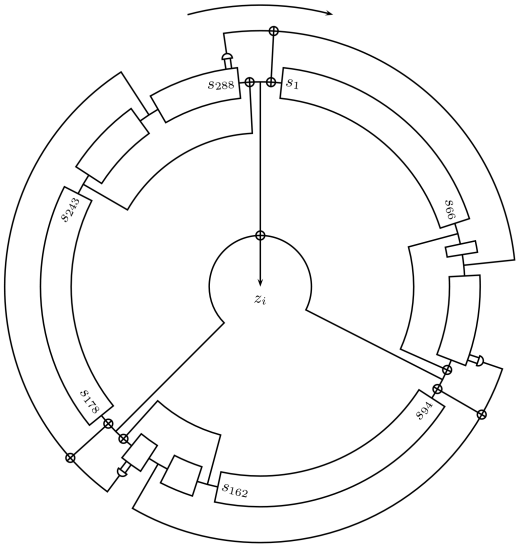


FIGURE 8.4 – Schéma de fonctionnement de Trivium

Modèle d'attaque Pour cette attaque, décrite dans [68] par Hojsík *et al.*, l'adversaire est supposé pouvoir :

- obtenir les N premiers bits consécutifs de la suite chiffrante $\{z_i\}$, produite à partir d'un état initial IS_{t_0} ;
- injecter une faute qui correspond au basculement d'un seul bit dans IS_{t_0}, à une position aléatoire (cet état perturbé est alors noté IS'_{t_0}) ;
- obtenir les N premiers bits consécutifs de la suite chiffrante altérée $\{z'_i\}$, produite à partir d'un état IS'_{t_0} ;
- effectuer de nouvelles de fautes sur le même état interne IS_{t_0}, m fois.

Description de l'attaque Le but de l'attaque est de déterminer tous les bits de l'état interne IS_{t_0}. En effet, les procédures d'initialisation et de mise à jour de IS pouvant être inversées, il suffit de connaître l'état interne à t_0 puis de remonter dans l'exécution de Trivium pour obtenir la clé K. L'adversaire va donc chercher à résoudre un système d'équations linéaires en les bits de l'état interne $IS_{t_0} = (s_1, ..., s_{288})$.

Un premier ensemble d'équations peut être obtenu directement à partir de la suite chiffrante, la relation entre les bits de sortie et les bits de l'état interne étant linéaire. En particulier, les 66 premiers bits de sortie sont combinaisons linéaires des bits de IS_{t_0} :

$$s_{67-i} \oplus s_{94-i} \oplus s_{163-i} \oplus s_{178-i} \oplus s_{244-i} \oplus s_{289-i} = z_i, \quad 1 \leq i \leq 66$$

Les 82 bits de sortie suivants présentent une dépendance quadratique. Le degré des expressions augmente ensuite.

Nous disposons donc de 66 équations, ce qui n'est pas suffisant. En partant de l'observation que $(s_i \oplus 1) \cdot s_j \oplus s_i \cdot s_j = s_j$, Hojsík *et al.* ont découvert que la basculement d'un bit dans l'état interne donnait soit directement la valeur de certains bits d'état en sortie, soit des équations en les bits d'état. Les auteurs ont donc choisi de se placer dans le cadre d'une attaque différentielle et d'utiliser les $\{d_i\}$, où $d_i = z_i \oplus z'_i$. Le tableau 8.1 donne les expressions obtenues à partir de $\{d_i\}$ pour une faute à la position 3.

TABLE 8.1 – Éléments non nuls de $\{d_i\}_{i=1}^{230}$ pour une faute en s_3

i	d_i
$64, 91, 148, 175, 199, 211, 214, 217$	1
$158, 175, 224$	s_4
$159, 174, 225$	s_3
212	$s_4 + s_{31} + s_{29} \cdot s_{30} + s_{109}$
213	$s_2 + s_{29} + s_{27} \cdot s_{28} + s_{107}$
227	$s_{178} + s_{223} + s_{221} \cdot s_{222} + s_4$
228	$s_{161} + s_{176} + s_{174} \cdot s_{175} + s_{263} + s_{221} + s_2 + s_{219} \cdot s_{220}$

Localisation de la faute. Au cours de l'attaque, l'adversaire n'a pas le contrôle sur la position du bit qu'il fait basculer, celle-ci est aléatoire. Pourtant, les différentes équations obtenues à partir d'une faute dépendent de la position du bit basculé. Les auteurs ont donc proposé une méthode très simple permettant de localiser la faute, basée sur l'observation des distances séparant les bits fautés (*i.e.* les bits pour lesquels $d_i = 1$).

La sortie est combinaison de six bits de l'état interne : deux de chaque registre. Or la distance séparant ces 2 bits est différente d'un registre à l'autre :

- sur le premier registre, les bits intervenant dans la sortie sont aux positions 66 et 93, ce qui fait un écart de 27 ;
- sur le deuxième registre, les bits concernés sont aux positions 162 et 177, d'où un écart de 15 ;
- enfin, sur le troisième registre, les bits s_{243} et s_{288}, ce qui fait un écart de 45.

Notons e la position de la faute, *i.e.* $s'_e = s_e + 1$. Pour se placer dans un cas simple, on fait l'hypothèse que $e \in \{1, ..., 66\} \cup \{94, ..., 162\} \cup \{178, ..., 243\}$. Notons a l'indice du premier bit chiffré fauté, *i.e.* $d_a = 1$ et $d_j = 0$ pour tout $1 \leq j \leq a$, alors si la faute a été injectée sur le premier registre, $d_{a+27} = 1$. De même, si la faute a été injectée sur le deuxième registre, on aura $d_{a+15} = 1$ et $d_{a+27} = 0$, et dans le dernier cas, $d_{a+45} = 1$ et $d_{a+27} = d_{a+15} = 0$. Ainsi, on est en mesure de déterminer sur quel registre la faute a eu lieu. Connaissant l'indice a du premier bit non nul dans la différence de suites chiffrantes, on sait à quelle distance était la faute du premier *tap* utilisé pour la sortie sur le registre identifié.

Dans le cas où e n'appartient pas à l'ensemble mentionné précédemment, le raisonnement reste le même ; il faut cependant prendre en compte les effets des termes de degré 2 en plus, qui peuvent générer d'autres différences non nulles.

Il est finalement possible de dresser des tables de correspondances permettant de déterminer de manière unique la position de la faute. Ces tables sont données ci-dessous (X représente un valeur non nulle différente de 1, s_{i+1} et s_{i-1}) :

TABLE 8.2 – Valeurs non nulles de $\{d_j\}$, pour une faute en s_i, $1 \leq i \leq 93$

2*Pos. de la faute	Valeur de d_j pour $j =$												
	67-i	94-i	136-i	151-i	161-i	162-i	163-i	176-i	177-i	178-i	202-i	220-i	242-i
$i = 1$	1	1		1	s_{i+1}	X		s_{i+1}	X	1	1		
$i = 2, ..., 66$	1	1		1	s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1	1		
$i = 67, ..., 69$		1	1		s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1		1	
$i = 70, ..., 90$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			1
$i = 91$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			
$i = 92$		1				s_{i-1}	1		s_{i-1}	1			
$i = 93$		1					1			1			

Déroulé de l'attaque. Cette attaque utilise uniquement les équations linéaires en les bits de l'état interne $(s_1, ..., s_{288})$, données par les 66 premiers bits chiffrés et par les différences entre suites chiffrantes. Elle est précédée par une phase de précalculs, durant laquelle l'adversaire exprime, pour toute position possible de la faute e, avec $1 \leq e \leq 288$, les potentiels bits non-nuls dans les suites $\{d_i\}$ en tant que polynômes de degré inférieur ou égal à 1 en les $(s_1, ..., s_{288})$, en se servant notamment du fait que $d_i = z_i \oplus z'_i$. Ces équations sont placées dans une table.

TABLE 8.3 – Valeurs non nulles de $\{d_j\}$, pour une faute en s_i, $94 \leq i \leq 177$

2*Pos. de la faute	Valeur de d_j pour $j =$											
	163-i	178-i	229-i	241-i	242-i	243-i	244-i	256-i	274-i	287-i	288-i	289-i
$i = 94$	1	1	1	1	s_{i+1}	X	1	1	1	s_{i+1}	X	1
$i = 95, ..., 162$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	1	s_{i+1}	s_{i-1}	1
$i = 163, ..., 171$		1		1	s_{i+1}	s_{i-1}	1	1		s_{i+1}	s_{i-1}	1
$i = 172, ..., 174$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 175$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 176$		1				s_{i-1}	1				s_{i-1}	1
$i = 177$		1					1					1

TABLE 8.4 – Valeurs non nulles de $\{d_j\}$, pour une faute en s_i, $178 \leq i \leq 288$

2*Pos. de la faute	Valeur de d_j pour $j =$											
	289-i	310-i	331-i	371-i	353-i	354-i	355-i	376-i	380-i	381-i	382-i	394-i
$i = 178$	1	1	1	1	s_{i+1}	X	1	1	s_{i+1}	X	1	1
$i = 179, ..., 243$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	1
$i = 244, ..., 264$	1		1		s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	
$i = 265, ..., 285$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 286$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 287$	1					s_{i-1}	1			s_{i-1}	1	
$i = 288$	1						1				1	

Pour chaque faute injectée, l'adversaire localise la position du bit touché, et va chercher dans la table précalculée les équations correspondant à la bonne position, en remplaçant le terme de droite par la véritable valeur observée. Les équations ainsi obtenues sont placées dans une matrice binaire M de 289 colonnes (une colonne pour chaque s_i, une colonne de plus pour le terme de droite dans l'équation). Une fois que le rang de M est suffisamment grand, il ne reste qu'à résoudre le système par méthode de Gauss.

Les auteurs ont déterminé qu'il valait mieux déterminer un nombre T de bits via cette attaque et procéder par recherche exhaustive sur les $288 - T$ restants pour réduire le nombre de fautes à injecter. En effet, si au début de l'attaque, chaque faute injectée nous donne des équations exploitables, il devient de plus en plus difficile de perturber le registre à des positions nouvelles. Pour déterminer tous les 288 bits, beaucoup de fautes injectées seraient donc gaspillées avant de toucher le registre à une position intéressante. Pour $T = 288$, les auteurs ont notamment observé qu'il fallait en moyenne 382 fautes, alors que pour $T = 260$, le nombre de fautes requis descend à 189. L'algorithme 8.2.3.2 résume cette attaque.

Première amélioration : utilisation de paires quadratiques Cette seconde attaque permet d'obtenir plus d'équations à partir d'une faute, réduisant donc le nombre total de fautes à injecter. Ainsi, elle utilise les équations linéaires vues précédemment, mais également les équations de degré 2 dont les termes quadratiques sont de la forme $s_i.s_{i+1}$, appelées *paires quadratiques* par De Cannière *et al.* Les auteurs font le choix de n'utiliser que les équations de ce type afin de ne pas obtenir une complexité trop grande.

Algorithm 5 Attaque différentielle par fautes sur Trivium

Entrée : Algorithme Trivium et possibilités d'injection de fautes, nombre T de bits à déterminer
Sortie : Clé secrète K
obtenir n bits consécutifs de la séquence $\{z_i\}$, générée depuis IS_{t_0}
  insérer dans M les équations obtenues à partir des 66 premiers bits $\{z_i\}$, en utilisant les vraies valeurs z_i
  tant que *rang(M)*$< T$ **faire**
    injecter une faute dans IS_{t_0}
      obtenir n bits consécutifs de la séquence perturbée $\{z'_i\}$
      calculer $d_i = z'_i + z_i$, $1 \leq i \leq n$
      déterminer la position e de la faute
      insérer dans M les équations précalculées correspondant à e
      appliquer le pivot de Gauss à M
répéter
    deviner les 288-T bits d'état interne
      résoudre le système linéaire donné par M et les bits devinés
      stocker la solution dans un état IS_s
      produire la suite chiffrante $\{\tilde{z}_i\}_{i=1}^{n}$ à partir de IS_s
jusqu'à $\forall i \in \{1, ..., n\}$, $\tilde{z}_i = z_i$;
remonter dans l'exécution de Trivium à partir de IS_s jusqu'à un état interne $IS_0 = (s_1^0, ...s_{288}^0)$ de la forme $(s_1^0, ..., s_{93}^0) = (a_1, ..., a_{80}, 0, ..., 0)$, $(s_{94}^0, ..., s_{177}^0) = (b_1, ..., b_{80}, 0, ..., 0)$, $(s_{178}^0, ..., s_{288}^0) = (0, ..., 0, 1, 1, 1)$
  retourner $K = (s_1^0, ...s_{288}^0)$

L'attaque est similaire à la première et son modèle reste le même. Elle est également précédée d'une phase de précalculs, durant laquelle on exprime pour chaque position possible de la faute les $\{d_i\}$ en fonction de $(s_1, ..., s_{288})$. Cette fois cependant, en plus des équations linéaires, sont également stockées les équations à paires quadratiques.

La différence principale a lieu durant la phase active de l'attaque : dès qu'il est possible de connaître la valeur d'une variable, on fait appel à une fonction notée *Substitution*(), qui remplace toute occurence de cette variable par sa vraie valeur, réduisant éventuellement à 1 le degré de certaines équations quadratiques.

Cette attaque demande deux fois plus de mémoire (la matrice M aura 576 colonnes : 288 pour chaque bit d'état, 287 pour toutes les paires possibles et 1 pour le terme de droite), mais diminue grandement le nombre de fautes à injecter. Nous passons en effet de 200 à 43 fautes en moyenne. L'algorithme 8.2.3.2 décrit cette attaque.

Seconde amélioration : utilisation du modèle "flottant" Cette attaque, que l'on doit toujours à Hojsík *et al.* [69], propose d'utiliser une autre représentation de l'algorithme : le *modèle flottant* (*Floating Model*). Le modèle d'attaque reste toujours le même. L'idée est cette fois de créer une nouvelle variable pour tout nouveau bit d'état, tout en conservant ses connexions avec les autres bits. On obtient alors des équations linéaires ou de degré très petit à partir de la suite chiffrante. Ensuite, comme dans l'attaque précédente, on injecte des fautes et on utilise les différences de suites chiffrantes pour obtenir plus d'équations. Après avoir

Algorithm 6 Attaque différentielle par fautes améliorée sur Trivium

Entrée : Algorithme Trivium et possibilités d'injection de fautes
Sortie : Clé secrète K
obtenir n bits consécutifs de la séquence $\{z_i\}$, générée depuis IS_{t_0}
 insérer dans M les équations obtenues à partir des 66 premiers bits $\{z_i\}$, en utilisant les vraies valeurs z_i
 tant que $(s_1, ...s_{288})$ *n'est pas entièrement déterminé* **faire**
  injecter une faute dans IS_{t_0}
   obtenir n bits consécutifs de la séquence perturbée $\{z'_i\}$
   calculer $d_i = z'_i + z_i$, $1 \leq i \leq n$
   déterminer la position e de la faute
   insérer dans M les équations précalculées correspondantes à e
   répéter
    faire *Substitution*()
  jusqu'à *ce que le système d'équations ne change plus*;
  appliquer le pivot de Gauss à M
   si *le pivot de Gauss donne de nouvelles variables* **alors**
    aller à 8.2.3.2
remonter dans l'exécution de Trivium à partir de IS_s jusqu'à un état interne $IS_0 = (s^0_1, ...s^0_{288})$ de la forme $(s^0_1, ..., s^0_{93}) = (a_1, ..., a_{80}, 0, ..., 0)$, $(s^0_{94}, ..., s^0_{177}) = (b_1, ..., b_{80}, 0, ..., 0)$, $(s^0_{178}, ..., s^0_{288}) = (0, ..., 0, 1, 1, 1)$
 retourner $K = (s^0_1, ...s^0_{288})$

traité les équations, l'adversaire cherche un ensemble de bits d'état pour un moment t qui soit exploitable (c'est-à-dire un état IS_t dont il connaît suffisamment de bits).

Représentation choisie. Les registres sont représentés par la séquence qu'ils produisent et non comme des NLFSRs de longueur finie. On note ainsi $\{x_n\}$, $\{y_n\}$ et $\{z_n\}$ les séquences produites par les trois NLFSRs et $\{o_n\}$ le flux de sortie.

À l'instant t le contenu de l'état interne est donc :

$$IS_t = (x_{t-1}, ...x_{t-93}, y_{t-1}, ...y_{t-84}, z_{t-1}, ...z_{t-111}).$$

On a les équations suivantes, décrivant les relations entre les bits de l'état interne :

$$\begin{aligned} x_n &= x_{n-69} + z_{n-66} + z_{n-111} + z_{n-110} \cdot z_{n-109} \\ y_n &= y_{n-78} + x_{n-66} + x_{n-93} + x_{n-92} \cdot x_{n-91} \\ z_n &= z_{n-87} + y_{n-69} + y_{n-84} + y_{n-83} \cdot y_{n-82} \end{aligned} \tag{8.2}$$

$$o_n = x_{n-66} + x_{n-93} + y_{n-69} + y_{n-84} + z_{n-6} + z_{n-111} \tag{8.3}$$

On note $\{\delta x_n\}$, $\{\delta y_n\}$, $\{\delta z_n\}$et $\{\delta o_n\}$ les différences entre les séquences perturbées et les séquences d'origine.

De (8.2) et (8.3), on a :

$$\delta o_n = \delta x_{n-66} + \delta x_{n-93} + \delta y_{n-69} + \delta y_{n-84} + \delta z_{n-6} + \delta z_{n-111} \tag{8.4}$$

$$\delta x_n = \delta x_{n-69} + \delta z_{n-66} + \delta z_{n-111} + \delta z_{n-110} \cdot z_{n-109} + z_{n-110} \cdot \delta z_{n-109} + \delta z_{n-110} \cdot \delta z_{n-109} \tag{8.5}$$

$$\delta y_n = \delta y_{n-78} + \delta x_{n-66} + \delta x_{n-93} + \delta x_{n-92} \cdot x_{n-91} + x_{n-92} \cdot \delta x_{n-91} + \delta x_{n-92} \cdot \delta x_{n-91} \tag{8.6}$$

$$\delta z_n = \delta z_{n-87} + \delta y_{n-69} + \delta y_{n-84} + \delta y_{n-83} \cdot y_{n-82} + y_{n-83} \cdot \delta y_{n-82} + \delta y_{n-83} \cdot \delta y_{n-82} \tag{8.7}$$

Durant l'attaque, l'adversaire calcule donc la séquence $\{\delta o_n\}$ pour chaque faute, et exprime les bits en fonction des $\{x_n\}$, $\{y_n\}$ et $\{z_n\}$ en utilisant les équations (8.4), (8.5), (8.6) et (8.7), que nous appellerons *équations delta*.

L'adversaire dispose donc de trois ensembles d'équations qu'il pourra résoudre :
- les équations linéaires de la forme (8.3), provenant de o_n ;
- les équations quadratiques de la forme (8.2), décrivant les relations entre les x_i, y_i et z_i ;
- les *équations delta* résultant des injections de fautes.

Différences avec le modèle classique, dit "statique". Dans le modèle statique, à un instant t, une variable x_t est exprimée en fonction des bits de l'état interne initial $IS_0 = (x_{-1}, ...x_{-93}, y_{-1}, ...y_{-84}, z_{-1}, ..., z_{-111}) = (s_1, ..., s_{288})$. Certaines des *équations delta*, qui seraient linéaires si elles étaient exprimées en fonction des variables flottantes x_t, y_t ou z_t, deviennent alors, dans cette représentation, des polynômes en les variables de l'état initial statique $(s_1, ..., s_{288})$ de plus haut degré. En revanche, on dispose de beaucoup plus de variables dans le modèle flottant : $3N + 288$ (288 variables pour l'état initial, 3 de plus à chaque étape).

Autre différence notable, nous cherchons dans le modèle statique à déterminer un nombre fixe de variables, qui correspondent aux bits de l'état interne IS_{t_0} au moment de l'injection de la faute. Dans le modèle flottant, on décide de l'état interne IS_t que l'on veut calculer, en fonction des équations disponibles. On attend donc que la faute se propage dans tout l'état interne, et seulement à ce moment, on essaye de déterminer l'état interne à un moment t qui est le plus intéressant (c'est-à-dire un instant t pour lequel on est en mesure de connaître suffisamment de bits d'état interne).

Ainsi, dans ce modèle flottant, on essaye de déterminer les valeurs de tous les $\{x_n\}, \{y_n\}$, et $\{z_n\}$ et on attend d'avoir assez de variables connues pour un état interne, quelle que soit sa position dans le temps.

Description de l'attaque. L'attaque est décrite par l'algorithme 8.2.3.2. Elle reprend les étapes des attaques précédentes. Après avoir injecté différentes fautes et déterminé leur position, l'attaquant calcule les *équations delta* correspondantes et les ajoute à son système d'équations, qui contient déjà des équations de la forme (8.5), (8.6), et (8.7). Il essaye alors de résoudre ce système, en utilisant le pivot de Gauss sur les équations linéaires et la fonction $Substitution()$ déjà employée dans l'attaque précédente, qui permet de réduire le degré d'équations non-linéaires en substituant les variables déjà déterminées par leur vraie valeur.

Cette attaque demande en moyenne 3,2 injections de faute et 800 bits de sortie, pour une complexité d'environ $2^{37,7}$ opérations élémentaires si on choisit $N = 800$. Dans les meilleurs cas, seulement deux fautes

Algorithm 7 Attaque de Trivium en représentation flottante

obtenir N bits consécutifs de la séquence $\{o_n\}$, générée depuis IS_0
ajouter les équations de suite chiffrante et les relations entre les bits au système d'équations
tant que $\forall$ *instant* t, IS_t *inconnu* **faire**
    réinitialiser Trivium à l'état IS_0
    injecter faute dans IS_0
    obtenir N bits consécutifs de la séquence $\{o'_n\}$
    $\delta o_n \leftarrow o_n + o'_n, \quad n = 0, ..., N-1$
    déterminer la position e de la faute
    générer les équations delta pour e et les ajouter au système d'équations
    répéter
        faire $Substitution()$
    jusqu'à *ce que le système d'équations ne change plus*;
    appliquer la méthode du pivot de Gauss au système
    si *le pivot de Gauss donne de nouvelles variables* **alors**
        aller à 8.2.3.2
remonter dans l'exécution de Trivium à partir de IS_t jusqu'à un état interne $IS_0 = (s^0_1, ..., s^0_{288})$ de la forme $(s_1, ..., s_{93}) = (a_1, ..., a_{80}, 0, ..., 0)$, $(s_{94}, ..., s_{177}) = (b_1, ..., b_{80}, 0, ..., 0)$, $(s_{178}, ..., s_{288}) = (0, ..., 0, 1, 1, 1)$
retourner $K = (s^0_1, ...s^0_{288})$

suffisent. Cette attaque est donc très intéressante, car elle montre l'importance du choix de représentation d'un système en cryptanalyse.

8.2.3.3 Attaques sur d'autres algorithmes

Nous avons choisi de nous attarder sur les attaques contre RC4 et Trivium car nous nous sommes principalement inspirés de ces attaques pour étudier les vulnérabilités de l'algorithme auto-synchronisant du projet, du point de vue des fautes. D'autres attaques ont cependant été considérées, notamment celles portant sur les autres algorithmes du portfolio eSTREAM :

- Attaque différentielle sur HC-128 [70],
- Attaques différentielles sur Rabbit [71][72],
- Attaques différentielles de Grain-128 [73] et de toute la famille Grain [74],
- Attaque différentielle de la famille MICKEY [75],
- Attaque différentielle sur Sosemanuk [76].

Le tableau 8.5 résume les résultats observés.

Une attaque en fautes sur Sprout [77] (algorithme basé sur Grain-128a proposé en 2015 par Armknecht *et al.*) a aussi été étudiée, l'algorithme ayant la particularité d'utiliser des sous-clés, à l'instar de notre algorithme à attaquer.

Les algorithmes de chiffrement par flot pouvant aussi être considérés comme des algorithmes de chiffrement par bloc en mode CFB (*Cipher Feedback Mode*), nous avons également regardé ce qui avait été fait sur les schémas SPN (*Substitution-Permutation Network*), notamment l'attaque de Piret *et al.* [78] et celle sur l'AES de Fuhr *et al.* [79], qui utilise uniquement les symboles chiffrés perturbés. Malheureusement, nous

TABLE 8.5 – Résultats des différentes attaques en fautes sur les algorithmes du portfolio eSTREAM

Ref.	Type de faute	Position de la faute	#Fautes	#Candidats restants
Trivium				
[68]	1 bit basculé	Aléatoire	43	1
[69]	1 bit basculé	Aléatoire	3,2	1
HC-128				
[70]	Aléatoire sur 1 mot (32 bits)	Aléatoire	7968	2
Rabbit				
[71]	1 bit basculé	Aléatoire	128 - 256	1
[71]	1 bit basculé	Choisie	24	1
[72]	1 ADD changé en XOR	Aléatoire	32	1
Grain v1				
[74]	1 bit basculé	Aléatoire	150	1
Grain-128				
[73]	1 bit basculé	Aléatoire	24	1
[74]	1 bit basculé	Aléatoire	312	1
Grain-128a				
[74]	1 bit basculé	Aléatoire	384	1
MICKEY v1				
[75]	1 bit basculé	Aléatoire	240	1
MICKEY-128				
[75]	1 bit basculé	Aléatoire	384	1
MICKEY 2.0				
[75]	1 bit basculé	Aléatoire	300	1
MICKEY-128 2.0				
[75]	1 bit basculé	Aléatoire	480	1
Sosemanuk				
[76]	Aléatoire sur 1 mot (32 bits)	Aléatoire	6144	2^{48}

n'avons pas pu adapter ces attaques à notre algorithme.

Enfin, des attaques non-physiques, purement cryptanalytiques sur des algorithmes auto-synchronisants ont été étudiées :

- Attaque par clairs choisis sur la version auto-synchronisante de Sober (*Self-Synchronizing Sober*)[80],
- Attaque différentielle sur Knot [81],
- Attaque par clairs choisis sur Mosquito [82],
- Attaque par corrélation de clés sur Moustique [83].

Ces attaques étaient néanmoins trop spécifiques aux algorithmes ciblés pour être reproduites.

8.2.4 Contremesures et méthodes de conception des algorithmes

Contremesures. Très peu de contremesures ont été proposées pour les algorithmes de chiffrement par flot. Les auteurs de l'attaque sur Grain-128 proposent dans leur papier [73] de dupliquer les LFSRs et de comparer constamment ces deux registres dupliqués. Si jamais leurs contenus diffèrent, le circuit peut alors lever une interruption et arrêter l'algorithme.

Il est également suggéré l'ajout de redondance au LFSR de telle sorte que l'opération de rétroaction du LFSR soit remplacée par une transformation linéaire incluant une méthode de détection d'erreurs adapté.

La duplication de l'état interne (et donc des calculs) serait trop coûteuse pour la plupart des algorithmes, où l'état interne est le composant le plus grand. En effet, pour les implémentations matérielles, il faudrait doubler l'espace requis tandis que pour les implémentations logicielles, il faudrait 100% de temps de calcul en plus. Compte tenu du fait que les algorithmes de chiffrement par flot sont essentiellement utilisés pour leur vitesse et leur taille réduite, une telle contremesure ne semble pas très intéressante.

Concernant l'ajout de redondance, nous risquons d'être limités par la capacité de détection du code correcteur utilisé. Les attaques en fautes tolérant généralement des fautes sur plusieurs bits, l'implémentation de méthodes de détection d'erreurs basées sur de la redondance risque d'être aussi coûteuse que la duplication des calculs.

Conception des algorithmes de chiffrement par flot. Dans [61], les auteurs suggèrent d'utiliser des fonctions non-linéaires de haut degré pour la rétroaction, afin d'éviter toute relation linéaire entre la sortie et l'état interne, diminuant grandement l'information différentielle exploitable.

Une autre idée proposée par les auteurs est d'utiliser des couches non-linéaires dépendant de la clé, telles des boîtes-S, tout en veillant à ce que ces couches ne constituent pas une faille exploitable.

Enfin, il est suggéré dans [60] d'ajouter de l'aléa dans l'exécution d'un algorithme : si l'ordre d'exécution des opérations de l'algorithme est fait dans un ordre aléatoire, il devient difficile pour l'attaquant de prédire ce que fait la machine à tout cycle. Pour la plupart des attaques, cette contremesure ralentira l'adversaire, même si la faute finira bien par atteindre l'instruction visée. En revanche, elle peut être efficace contre des attaques requérant des injections de fautes à des emplacements spécifiques ou dans un ordre spécifique.

Le contexte général des attaques en fautes et les différentes attaques étudiées ont donc été présentés. Dans le chapitre suivant, nous introduisons l'algorithme sur lequel nous avons travaillé, nommé dans ce qui suit "LPV SSSC", et les analyses par injection de fautes que nous avons réalisées sur celui-ci.

8.3 Description de la version réduite de LPV SSSC étudiée

La structure de LPV SSSC repose sur un registre de n semi-octets, noté Rx. L'algorithme prend en entrée un IV et une clé de $\frac{4\times(n+s+r)}{2}$ bits, où s désigne le nombre de symboles chiffrés intervenant dans la fonction de mise à jour, et r le retard, soit la case de Rx utilisée pour produire les symboles chiffrés.

Nous nous concentrerons sur le cas $n = 7$, avec $r = 2$ et $s = 1$, côté chiffreur. Nous rappelons ici que nous étudions une version "jouet" de notre algorithme final, ce dernier étant en dimension $n = 40$. Les notations utilisées dans la suite du chapitre sont indiquées ci-dessous :

- Le coup d'horloge est noté t
- L'état du registre Rx au coup d'horloge t est noté $x_t[0], ..., x_t[6]$
- La clé secrète utilisée est notée $K = (k_0, ..., k_{21})$
- Les 22 sous-clés dérivées de K sont notées SK_i, $i \in [\![0, 21]\!]$
- La boîte-S 4×4 bijective utilisée dans le schéma est notée S, et son inverse, S^{-1}
- Le symbole clair au coup d'horloge t est noté m_t
- Le symbole chiffré au coup d'horloge t est noté c_t
- $+$ et $\cdot$ désignent l'addition et la multiplication dans $\mathbb{F}_{16}$ respectivement, avec $\mathbb{F}_{16} \simeq \mathbb{F}_2/(x^4 + x + 1)$
- L'inverse dans $\mathbb{F}_{16}^{\times}$ est noté $(.)^{-1}$

Pour $n = 7$, $s = 1$ et $r = 2$ la mise à jour de Rx s'effectue de la manière suivante :

$$
\begin{aligned}
x_{t+1}[0] &= \textstyle\sum_{j=0}^{6} \mathrm{S}(c_t + \mathrm{SK}_j) \cdot x_t[j] + m_t \\
x_{t+1}[1] &= x_t[0] + x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_7) \cdot x_t[2] + x_t[6] \\
x_{t+1}[2] &= x_t[1] + x_t[2] \\
x_{t+1}[3] &= \mathrm{S}(c_t + \mathrm{SK}_8) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_9) \cdot x_t[2] \\
x_{t+1}[4] &= \mathrm{S}(c_t + \mathrm{SK}_{10}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{11}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot x_t[3] \\
x_{t+1}[5] &= \mathrm{S}(c_t + \mathrm{SK}_{13}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{14}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot x_t[3] \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot x_t[4] \\
x_{t+1}[6] &= \mathrm{S}(c_t + \mathrm{SK}_{17}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{18}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot x_t[3] \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot x_t[4] + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot x_t[5]
\end{aligned}
$$

Le symbole chiffré suivant est ensuite produit :

$$c_{t+1} = x_{t+1}[2]$$

8.4 Modèle de faute

Nous avons choisi de nous placer dans le cadre d'une attaque différentielle. Ainsi, au cours de l'attaque, des perturbations sont injectées. Les différences entre les séquences de symboles chiffrés fautés et de symboles chiffrés non altérés sont ensuite exploitées afin d'obtenir les sous-clés.

Le degré de contrôle de l'adversaire qui a été supposé est donc le suivant :

1. L'adversaire peut injecter des fautes dans l'état interne aux positions de son choix.

2. La faute touche exactement un semi-octet du registre interne, et ce, juste avant la mise à jour de Rx (ainsi, au moment t de l'injection, le symbole chiffré c_t produit n'est pas affecté par la faute).
3. L'adversaire peut réinitialiser l'implémentation, avec le même vecteur d'initialisation.

8.5 Analyse différentielle des perturbations

Une étude préliminaire des perturbations a été d'abord été faite afin de voir quelles informations nous pouvions en tirer et si une attaque était possible. Nous avons donc regardé tous les cas de perturbations possibles et la façon dont celles-ci se propagent dans l'état interne durant l'exécution. Ainsi, dans cette section seront présentés les différents effets de la propagation d'une faute en $x[i]$, $0 \leq i \leq 6$, au coup d'horloge t.

La figure 8.5 donne un exemple de propagation d'une faute sur $x_t[3]$. Le registre Rx à l'instant t est à gauche et à droite se trouve ce même registre à l'instant $t+1$. Les Φ^i_j représentent les termes en $x_t[i]$ dans la formule de mise à jour de $x_{t+1}[j]$ *i.e.* ce sont les $\mathrm{S}(c_t + \mathrm{SK}) \cdot x_t[i]$ dans le calcul de $x_{t+1}[j]$. Ainsi, si $x_t[3]$ est fauté, toutes les valeurs Φ^3_j sont erronées à l'instant t, perturbant alors certaines valeurs de Rx à $t+1$. Les valeurs fautées sont représentées en rouge.

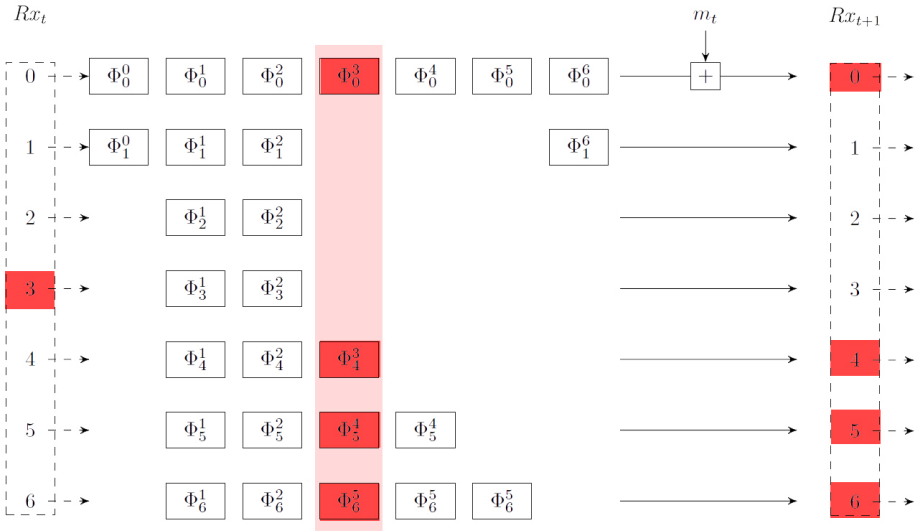


FIGURE 8.5 – Effets sur un tour d'une faute sur $x_t[3]$

Notons $\Delta x[i]$ la différence entre la valeur fautée (notée $x'[i]$) et la valeur correcte du $x[i]$ à tout instant, *i.e.* $\Delta x[i] = x[i] + x'[i]$. Afin d'alléger les équations, la différence initiale (c'est-à-dire celle au moment t de l'injection de la faute) entre les valeurs fautée et non fautée de $x[i]$ sera notée δ_i, *i.e.* $\delta_i = \Delta x_t[i] = x_t[i] + x'_t[i] \neq 0$.

On peut remarquer que, du fait de la structure de l'algorithme et de sa nature auto-synchronisante, à partir du moment où un symbole chiffré est affecté par la propagation de la faute, à un coup d'horloge qui sera par exemple noté t_c, l'observation des différences entre les symboles chiffrés corrects et les symboles chiffrés perturbés au-delà de l'instant $t_c + 2$ n'est plus intéressante. Les équations obtenues contiendraient en effet des termes en les $x[i]$, sur lesquels l'adversaire n'a aucune information. Nous ne disposons donc que d'un

nombre limité d'équations exploitables.

Dans la suite, les équations auxquelles l'adversaire a accès ont été encadrées.

8.5.1 Faute en $x_t[0]$

À l'instant $t+1$:

$$\begin{aligned}
&\Delta x_{t+1}[0] = \delta_0 \cdot \mathrm{S}(c_t + \mathrm{SK}_0) \\
&\Delta x_{t+1}[1] = \delta_0 \\
&\boxed{\Delta x_{t+1}[2] = 0} \\
&\Delta x_{t+1}[3] = \Delta x_{t+1}[4] = \Delta x_{t+1}[5] = \Delta x_{t+1}[6] = 0
\end{aligned} \tag{8.8}$$

À $t+2$:

$$\begin{aligned}
&\Delta x_{t+2}[0] = \delta_0 \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1)\Big) \\
&\Delta x_{t+2}[1] = \delta_0 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_0) + 1\Big) \\
&\boxed{\Delta x_{t+2}[2] = \delta_0} \\
&\Delta x_{t+2}[3] = \delta_0 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_8) \\
&\Delta x_{t+2}[4] = \delta_0 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{10}) \\
&\Delta x_{t+2}[5] = \delta_0 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{13}) \\
&\Delta x_{t+2}[6] = \delta_0 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{17})
\end{aligned} \tag{8.9}$$

À $t+3$:

$$\begin{aligned}
\Delta x_{t+3}[1] = \delta_0 \cdot \Big(&\Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) + 1\Big) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) \\
&+ \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) \\
&+ x_{t+2}[2] \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big) \\
\boxed{\Delta x_{t+3}[2] = \delta_0 \cdot \mathrm{S}(c_t + \mathrm{SK}_0)}&
\end{aligned} \tag{8.10}$$

À $t+4$:

$$\boxed{\begin{aligned}
\Delta x_{t+4}[2] = \delta_0 \cdot \Big(&\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) \\
&+ \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) \\
&+ x_{t+2}[2] \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)
\end{aligned}} \tag{8.11}$$

8.5.2 Faute en $x_t[1]$

À l'instant $t+1$:

$$\begin{aligned}
\Delta x_{t+1}[0] &= \delta_1 \cdot \mathrm{S}(c_t + \mathrm{SK}_1) \\
\Delta x_{t+1}[1] &= \delta_1 \\
\boxed{\Delta x_{t+1}[2] = \delta_1} & \\
\Delta x_{t+1}[3] &= \delta_1 \cdot \mathrm{S}(c_t + \mathrm{SK}_8) \\
\Delta x_{t+1}[4] &= \delta_1 \cdot \mathrm{S}(c_t + \mathrm{SK}_{10}) \\
\Delta x_{t+1}[5] &= \delta_1 \cdot \mathrm{S}(c_t + \mathrm{SK}_{13}) \\
\Delta x_{t+1}[6] &= \delta_1 \cdot \mathrm{S}(c_t + \mathrm{SK}_{17})
\end{aligned} \tag{8.12}$$

À $t+2$:

$$\begin{aligned}
\Delta x_{t+2}[1] = \delta_1 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_1) + \mathrm{S}(c_t + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1\Big) \\
&+ x_{t+1}[2] \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big) \\
\boxed{\Delta x_{t+2}[2] = 0} &
\end{aligned} \tag{8.13}$$

À $t+3$:

$$\boxed{\begin{aligned}
\Delta x_{t+3}[2] = \delta_1 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_1) + \mathrm{S}(c_t + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1\Big) \\
&+ x_{t+1}[2] \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big)
\end{aligned}} \tag{8.14}$$

8.5.3 Faute en $x_t[2]$

À l'instant $t+1$:

$$\begin{aligned}
\Delta x_{t+1}[0] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_2) \\
\Delta x_{t+1}[1] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_7) \\
\boxed{\Delta x_{t+1}[2] = \delta_2} & \\
\Delta x_{t+1}[3] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_9) \\
\Delta x_{t+1}[4] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_{11}) \\
\Delta x_{t+1}[5] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_{14}) \\
\Delta x_{t+1}[6] &= \delta_2 \cdot \mathrm{S}(c_t + \mathrm{SK}_{18})
\end{aligned} \tag{8.15}$$

À $t+2$:

$$\begin{aligned}
\Delta x_{t+2}[1] = \delta_2 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_2) + \mathrm{S}(c_t + \mathrm{SK}_{18}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_t + \mathrm{SK}_7)\Big) \\
&+ x_{t+1}[2] \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big) \\
\boxed{\Delta x_{t+2}[2] = \delta_2 \cdot (\mathrm{S}(c_t + \mathrm{SK}_7) + 1)} &
\end{aligned} \tag{8.16}$$

À $t+3$:

$$\boxed{\begin{aligned}\Delta x_{t+3}[2] = \delta_2 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_2) + \mathrm{S}(c_t + \mathrm{SK}_{18}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1\Big) \\ &+ x_{t+1}[2] \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big)\end{aligned}} \tag{8.17}$$

8.5.4 Faute en $x_t[3]$

À l'instant $t+1$:

$$\begin{aligned}\Delta x_{t+1}[0] &= \delta_3 \cdot \mathrm{S}(c_t + \mathrm{SK}_3) \\ \Delta x_{t+1}[1] &= 0 \\ \boxed{\Delta x_{t+1}[2] = 0} \\ \Delta x_{t+1}[3] &= 0 \\ \Delta x_{t+1}[4] &= \delta_3 \cdot \mathrm{S}(c_t + \mathrm{SK}_{12}) \\ \Delta x_{t+1}[5] &= \delta_3 \cdot \mathrm{S}(c_t + \mathrm{SK}_{15}) \\ \Delta x_{t+1}[6] &= \delta_3 \cdot \mathrm{S}(c_t + \mathrm{SK}_{19})\end{aligned} \tag{8.18}$$

À $t+2$:

$$\begin{aligned}\Delta x_{t+2}[0] = \delta_3 \cdot \Big(&\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_3) + \mathrm{S}(c_{t+1} + \mathrm{SK}_4) \cdot \mathrm{S}(c_t + \mathrm{SK}_{12}) \\ &+ \mathrm{S}(c_{t+1} + \mathrm{SK}_5) \cdot \mathrm{S}(c_t + \mathrm{SK}_{15}) + \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \cdot \mathrm{S}(c_t + \mathrm{SK}_{19})\Big) \\ \Delta x_{t+2}[1] = \delta_3 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_3) + \mathrm{S}(c_t + \mathrm{SK}_{19})\Big) \\ \boxed{\Delta x_{t+2}[2] = 0} \\ \Delta x_{t+2}[3] = \Delta x_{t+2}[4] &= 0 \\ \Delta x_{t+2}[5] = \delta_3 \cdot \Big(&\mathrm{S}(c_{t+1} + \mathrm{SK}_{16}) \cdot \mathrm{S}(c_t + \mathrm{SK}_{12})\Big) \\ \Delta x_{t+2}[6] = \delta_3 \cdot \Big(&\mathrm{S}(c_{t+1} + \mathrm{SK}_{20}) \cdot \mathrm{S}(c_t + \mathrm{SK}_{12}) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21}) \cdot \mathrm{S}(c_t + \mathrm{SK}_{15})\Big)\end{aligned} \tag{8.19}$$

À $t+3$:

$$\begin{aligned}\Delta x_{t+3}[1] = \delta_3 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_3) \cdot (\mathrm{S}(c_{t+1} + \mathrm{SK}_0) + 1) + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_6) + 1\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_4) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{20})\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big)\Big) \\ \boxed{\Delta x_{t+3}[2] = \delta_3 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_3) + \mathrm{S}(c_t + \mathrm{SK}_{19})\Big)}\end{aligned} \tag{8.20}$$

À $t+4$:

$$\boxed{\begin{aligned}\Delta x_{t+4}[2] = \delta_3 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_3) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_4) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{20})\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big)\Big)\end{aligned}} \tag{8.21}$$

8.5.5 Faute en $x_t[4]$

À l'instant $t+1$:

$$\begin{aligned}\Delta x_{t+1}[0] &= \delta_4 \cdot \mathrm{S}(c_t + \mathrm{SK}_4) \\ \Delta x_{t+1}[1] &= 0 \\ \boxed{\Delta x_{t+1}[2] = 0} & \\ \Delta x_{t+1}[3] &= \Delta x_{t+1}[4] = 0 \\ \Delta x_{t+1}[5] &= \delta_4 \cdot \mathrm{S}(c_t + \mathrm{SK}_{16}) \\ \Delta x_{t+1}[6] &= \delta_4 \cdot \mathrm{S}(c_t + \mathrm{SK}_{20})\end{aligned} \tag{8.22}$$

À $t+2$:

$$\begin{aligned}\Delta x_{t+2}[0] = \delta_4 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_4) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_5) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6)\Big) \\ \Delta x_{t+2}[1] = \delta_4 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_4) + \mathrm{S}(c_t + \mathrm{SK}_{20})\Big) \\ \boxed{\Delta x_{t+2}[2] = 0} & \\ \Delta x_{t+2}[3] = \Delta x_{t+2}[4] &= \Delta x_{t+2}[5] = 0 \\ \Delta x_{t+2}[6] = \delta_4 \cdot \mathrm{S}(c_t + \mathrm{SK}_{16}) &\cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\end{aligned} \tag{8.23}$$

À $t+3$:

$$\begin{aligned}\Delta x_{t+3}[1] = \delta_4 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_4) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) + 1\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_6) + 1\Big)\Big)\end{aligned} \tag{8.24}$$

$$\boxed{\Delta x_{t+3}[2] = \delta_4 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_4) + \mathrm{S}(c_t + \mathrm{SK}_{20})\Big)} \tag{8.25}$$

À $t+4$:

$$\boxed{\begin{aligned}\Delta x_{t+4}[2] = \delta_4 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_4) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big) \\ &+ \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6)\Big)\end{aligned}} \tag{8.26}$$

8.5.6 Faute en $x_t[5]$

À l'instant $t+1$:

$$\begin{aligned}
&\Delta x_{t+1}[0] = \delta_5 \cdot \mathrm{S}(c_t + \mathrm{SK}_5) \\
&\Delta x_{t+1}[1] = 0 \\
&\boxed{\Delta x_{t+1}[2] = 0} \\
&\Delta x_{t+1}[3] = \Delta x_{t+1}[4] = \Delta x_{t+1}[5] = 0 \\
&\Delta x_{t+1}[6] = \delta_5 \cdot \mathrm{S}(c_t + \mathrm{SK}_{21})
\end{aligned} \tag{8.27}$$

À $t+2$:

$$\begin{aligned}
&\Delta x_{t+2}[0] = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6)\Big) \\
&\Delta x_{t+2}[1] = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_t + \mathrm{SK}_{21})\Big) \\
&\boxed{\Delta x_{t+2}[2] = 0} \\
&\Delta x_{t+2}[3] = \Delta x_{t+2}[4] = \Delta x_{t+2}[5] = \Delta x_{t+2}[6] = 0
\end{aligned} \tag{8.28}$$

À $t+3$:

$$\begin{aligned}
\Delta x_{t+3}[0] = \delta_5 \cdot \Big(&\mathrm{S}(c_{t+2} + \mathrm{SK}_1) \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_t + \mathrm{SK}_{21})\Big) \\
&+ \mathrm{S}(c_{t+2} + \mathrm{SK}_0) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \cdot \mathrm{S}(c_t + \mathrm{SK}_{21})\Big)\Big) \\
\Delta x_{t+3}[1] = \delta_5 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) + 1\Big) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_6) + 1\Big)\Big) \\
\boxed{\Delta x_{t+3}[2] = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_t + \mathrm{SK}_{21})\Big)} \\
\Delta x_{t+3}[6] = \delta_5 \cdot \mathrm{S}(c_{t+2} + \mathrm{SK}_{17}) \cdot &\Big(\mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_t + \mathrm{SK}_{21})\Big)
\end{aligned} \tag{8.29}$$

À $t+4$:

$$\begin{aligned}
\Delta x_{t+4}[1] = \delta_5 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_1) + \mathrm{S}(c_{t+2} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7) \\
&+ \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_0) + 1\Big) + 1\Big) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_1) \\
&+ \mathrm{S}(c_{t+2} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_0) + 1\Big) + 1\Big)\Big) \\
&+ x_{t+3}[2] \cdot \Big(\mathrm{S}(c_{t+3} + \mathrm{SK}_7) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7)\Big)
\end{aligned}$$

$$\boxed{\Delta x_{t+4}[2] = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6)\Big)} \tag{8.30}$$

À $t+5$:

$$\boxed{\begin{aligned}\Delta x_{t+5}[2] = \delta_5 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_1) + \mathrm{S}(c_{t+2} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7) \\ &+ \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_{t+2} + \mathrm{SK}_0) + 1\Big) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \Big(\mathrm{S}(c_{t+2} + \mathrm{SK}_1) \\ &+ \mathrm{S}(c_{t+2} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \cdot \mathrm{S}(c_{t+2} + \mathrm{SK}_0) + 1\Big)\Big) \\ &+ x_{t+3}[2] \cdot \Big(\mathrm{S}(c_{t+3} + \mathrm{SK}_7) + \mathrm{S}(c'_{t+3} + \mathrm{SK}_7)\Big)\end{aligned}} \tag{8.31}$$

8.5.7 Faute en $x_t[6]$

À l'instant $t+1$:

$$\begin{aligned}&\Delta x_{t+1}[0] = \delta_6 \cdot \mathrm{S}(c_t + \mathrm{SK}_6) \\ &\Delta x_{t+1}[1] = \delta_6 \\ &\boxed{\Delta x_{t+1}[2] = 0} \\ &\Delta x_{t+1}[3] = \Delta x_{t+1}[4] = \Delta x_{t+1}[5] = \Delta x_{t+1}[6] = 0\end{aligned} \tag{8.32}$$

À $t+2$:

$$\begin{aligned}&\Delta x_{t+2}[0] = \delta_6 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_6) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1)\Big) \\ &\Delta x_{t+2}[1] = \delta_6 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_6) + 1\Big) \\ &\boxed{\Delta x_{t+2}[2] = \delta_6} \\ &\Delta x_{t+2}[3] = \delta_6 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_8) \\ &\Delta x_{t+2}[4] = \delta_6 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{10}) \\ &\Delta x_{t+2}[5] = \delta_6 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{13}) \\ &\Delta x_{t+2}[6] = \delta_6 \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_{17})\end{aligned} \tag{8.33}$$

À $t+3$:

$$\begin{aligned}\Delta x_{t+3}[1] = \delta_6 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_6) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) + 1\Big) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) \\ &+ \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) + x_{t+2}[2] \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big) \\ \boxed{\Delta x_{t+3}[2] = \delta_6 \cdot \mathrm{S}(c_t + \mathrm{SK}_6)}\end{aligned} \tag{8.34}$$

À $t+4$:

$$\boxed{\begin{aligned}\Delta x_{t+4}[2] = \delta_6 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_6) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) \\ &+ \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) + x_{t+2}[2] \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)\end{aligned}} \tag{8.35}$$

Remarque : D'après l'observation faite en début de section, nous aurions pu aller jusqu'à $\Delta x_{t+5}[2]$ pour les cas de fautes sur $x_t[3]$ et $x_t[4]$, comme cela a été fait pour $x_t[5]$. Ces équations ont été calculées mais n'ont pas été utilisées dans l'attaque que nous allons décrire. Nous avons donc choisi de ne pas les inclure dans ce livrable, afin de ne pas le surcharger inutilement.

8.6 Description de l'attaque

L'attaque s'inspire principalement des attaques différentielles sur Trivium [68] et RC4 [64] présentées dans le chapitre précédent. Cependant, contrairement à ces attaques, nous ne cherchons pas à reconstruire l'état interne, celui-ci étant indépendant de la clé dans LPV SSSC.

Notre attaque se concentre uniquement sur les sous-clés SK_i. En effet, celles-ci ne changent jamais durant le chiffrement, elles ne sont générées qu'une seule fois, au moment où la clé est introduite. L'idée est donc d'obtenir des équations linéaires en les sous-clés. Ainsi, en fautant le registre Rx à différentes positions, une première phase consistera à calculer les $\Delta x[i]$ correspondants et utiliser les équations les plus simples pour éliminer certaines variables. On pourra alors dans un second temps utiliser ces nouvelles variables calculées pour réduire le degré des équations quadratiques et obtenir des équations linéaires supplémentaires.

8.6.1 Phase 1 : Obtention de SK_0, SK_6, SK_7

En observant les équations présentées dans la section 8.5, nous avons constaté qu'il nous était possible d'avoir directement la valeur de certaines sous-clés, en particulier SK_0, SK_7 et SK_6, et ce, en fautant une seule fois $x_t[0]$, $x_t[2]$ et $x_t[6]$ respectivement, ce qui amène donc à un total de trois fautes.

8.6.1.1 Obtention de SK_0

Pour une faute en $x_t[0]$, on peut utiliser les équations (8.9) et (8.10) :

$$\begin{cases} \Delta c_{t+2} = \delta_0 \\ \Delta c_{t+3} = \delta_0 \cdot \mathrm{S}(c_t + \mathrm{SK}_0) \end{cases}$$

Ce système donne directement la valeur de SK_0 :

$$\mathrm{SK}_0 = \mathrm{S}^{-1}\Big(\Delta c_{t+3} \cdot (\Delta c_{t+2})^{-1}\Big) + c_t$$

L'obtention de SK_6 et de SK_7 se fait de manière similaire.

8.6.1.2 Obtention de SK_7

En fautant $x_t[2]$, on obtient un système intéressant, formé des équations (8.15) et (8.16) :

$$\begin{cases} \Delta c_{t+1} = \delta_2 \\ \Delta c_{t+2} = \delta_2 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_7) + 1\Big) \end{cases}$$

ce qui donne valeur de SK_7 :

$$\mathrm{SK}_7 = \mathrm{S}^{-1}\Big(\Delta c_{t+2} \cdot (\Delta c_{t+1})^{-1} + 1\Big) + c_t$$

8.6.1.3 Obtention de SK_6

Avec une faute sur $x_t[6]$, on a, d'après (8.33) et (8.34) :

$$\begin{cases} \Delta c_{t+2} = \delta_6 \\ \Delta c_{t+3} = \delta_6 \cdot \mathrm{S}(c_t + \mathrm{SK}_6) \end{cases}$$

D'où :

$$\mathrm{SK}_6 = \mathrm{S}^{-1}\Big(\Delta c_{t+3} \cdot (\Delta c_{t+2})^{-1}\Big) + c_t$$

À la fin de cette phase, nous disposons donc des valeurs de SK_0, SK_6, SK_7, δ_0, δ_2 et δ_6. La prochaine étape va consister à réinjecter ces nouvelles valeurs dans d'autres équations pour réduire soit leur degré, soit leur nombre d'inconnues (ce qui implique une réduction du nombre d'équations requises pour les déterminer, et donc du nombre de fautes à injecter).

8.6.2 Phase 2 : Informations partielles sur (SK_1, SK_{17}) et (SK_2, SK_{18})

Durant cette phase, on exploite les trois fautes injectées précédemment (en $x_t[0]$, $x_t[2]$ et $x_t[6]$) pour restreindre l'ensemble des valeurs possibles de (SK_1, SK_{17}) et (SK_2, SK_{18}). Nous considérons ces sous-clés par paire, car en les permutant, les équations restent inchangées.

Pour pouvoir se ramener à seulement deux candidats pour (SK_2, SK_{18}), il nous faudra fauter $x[2]$ au moins une seconde fois. Concernant (SK_1, SK_{17}), les deux fautes déjà provoquées en $x_t[0]$ et $x_t[6]$ peuvent nous permettre de trouver seulement deux possibilités ; elle ne sont néanmoins pas toujours suffisantes.

8.6.2.1 Réutilisation de la faute en $x_t[0]$

On se sert de l'équation (8.11) :

$$\begin{aligned} \Delta c_{t+4} = \delta_0 \cdot \Big(&\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) \\ &+ c_{t+2} \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big) \end{aligned}$$

Ce qui implique :

$$\begin{aligned} \mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) = \delta_0^{-1} \cdot &\Big(\Delta c_{t+4} + c_{t+2} \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)\Big) \\ &+ \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + 1 + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) \end{aligned}$$

Le membre de droite de l'équation peut être calculé, les valeurs de δ_0, SK_0 et SK_7 ayant en effet été retrouvées durant la première phase.

$$\begin{aligned}\text{Si on pose } A = \delta_0^{-1} \cdot \Big(\Delta c_{t+4} + c_{t+2} \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)\Big) \\ + \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_0) + 1 + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7)\end{aligned}$$

alors,

$$\mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) = A \in \mathbb{F}_{16} \simeq GF(2)/(x^4 + x + 1).$$

Ainsi, seize couples $(\mathrm{S}(c_{t+1} + \mathrm{SK}_1), \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}))$ satisfont cette relation. Comme S est bijective, on obtient de même seize valeurs possibles pour $(\mathrm{SK}_1, \mathrm{SK}_{17})$.

8.6.2.2 Réutilisation de la faute en $x_t[6]$

On a d'après (8.35) :

$$\begin{aligned}\Delta c_{t+4} = \delta_6 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_6) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) \\ &+ \mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + 1\Big) \\ &+ c_{t+2} \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)\end{aligned}$$

Ce qui donne :

$$\begin{aligned}\mathrm{S}(c_{t+1} + \mathrm{SK}_1) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{17}) = \ & \delta_6^{-1} \cdot \Big(\Delta c_{t+4} + c_{t+2} \cdot \Big(\mathrm{S}(c'_{t+2} + \mathrm{SK}_7) + \mathrm{S}(c_{t+2} + \mathrm{SK}_7)\Big)\Big) \\ &+ \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \cdot \mathrm{S}(c_t + \mathrm{SK}_6) + 1 + \mathrm{S}(c'_{t+2} + \mathrm{SK}_7)\end{aligned}$$

On obtient de nouveau seize valeurs possibles pour $(\mathrm{SK}_1, \mathrm{SK}_{17})$.

Par intersection des deux ensembles de valeurs possibles obtenus en exploitant les fautes en $x[0]$ et en $x[6]$, on trouve un nombre réduit de candidats qui est supérieur ou égal à 2. Dans le cas où l'on a plus de deux candidats, il est encore possible d'affiner l'ensemble des valeurs possibles en utilisant des ensembles supplémentaires obtenus via des fautes en $x[0]$ ou $x[6]$, ou encore $x[1]$. La dernière option est décrite ci-dessous.

8.6.2.3 Faute en $x_t[1]$

Si on faute $x_t[1]$, alors on a l'équation suivante à $t+3$:

$$\begin{aligned}\Delta c_{t+3} = \delta_1 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_1) + \mathrm{S}(c_t + \mathrm{SK}_{17}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1\Big) \\ &+ c_{t+1} \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big)\end{aligned}$$

Cette équation donne

$$\begin{aligned}\mathrm{S}(c_t + \mathrm{SK}_1) + \mathrm{S}(c_t + \mathrm{SK}_{17}) = \delta_1^{-1} \cdot \Big(&\Delta c_{t+3} + c_{t+1} \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big)\Big) \\ &+ \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1,\end{aligned}$$

ce qui nous permet à nouveau de trouver seize solutions possibles pour (SK_1, SK_{17}).

L'attaque a été simulée en langage C. Expérimentalement, il a été constaté qu'en moyenne, trois ensembles de valeurs (donc trois fautes) étaient nécessaires pour restreindre les valeurs possibles au nombre de deux.

8.6.2.4 Réutilisation de la faute en $x_t[2]$

On procède comme pour (SK_1, SK_{17}) : on utilise des fautes sur $x_t[2]$ pour se ramener à deux candidats pour (SK_2, SK_{18}).

On a d'après (8.17) :

$$\begin{aligned}\Delta c_{t+3} = \delta_2 \cdot \Big(&\mathrm{S}(c_t + \mathrm{SK}_2) + \mathrm{S}(c_t + \mathrm{SK}_{18}) + \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1\Big) \\ &+ c_{t+1} \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big).\end{aligned}$$

Cela donne :

$$\begin{aligned}\mathrm{S}(c_t + \mathrm{SK}_2) + \mathrm{S}(c_t + \mathrm{SK}_{18}) = \delta_2^{-1} \cdot \Big(&\Delta c_{t+3} + c_{t+1} \cdot \Big(\mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + \mathrm{S}(c_{t+1} + \mathrm{SK}_7)\Big)\Big) \\ &+ \mathrm{S}(c'_{t+1} + \mathrm{SK}_7) + 1.\end{aligned}$$

Avec une faute sur $x_t[2]$, on obtient seize candidats pour (SK_2, SK_{18}). En moyenne, on arrive à deux candidats restants en fautant trois fois le registre en $x[2]$.

En résumé, par l'utilisation de fautes en $x_t[0]$ et $x_t[6]$, et éventuellement en $x_t[1]$, il est possible de trouver deux couples de valeurs possibles pour (SK_1, SK_{17}). Ces deux couples sont réciproques l'un de l'autre (on ne sait pas distinguer SK_1 de SK_{17}).

De même, à l'aide de fautes en $x_t[2]$, il nous est possible de trouver deux candidats pour (SK_2, SK_{18}).

8.6.3 Phase 3 : Obtention de SK_3, SK_4, SK_5, SK_{12}, SK_{15}, SK_{16}, SK_{19}, SK_{20}, SK_{21}

Pour cette phase, intéressons-nous à des fautes sur $x[3]$, $x[4]$ et $x[5]$. Contrairement à ce qui a été vu précédemment pour des fautes en $x[0]$, $x[1]$, $x[2]$ ou $x[6]$, il nous faut ici essayer toutes les valeurs non nulles que peut prendre la différence initiale δ. En effet, il n'est pas possible d'obtenir sa véritable valeur par directe observation des différences entre les symboles chiffrés fautés et les symboles chiffrés d'origine, comme le montrent les équations vues dans la section 8.5.

Cette phase exploite la structure triangulaire des dernières lignes la fonction de mise à jour de Rx :

$$\begin{aligned}
x_{t+1}[4] &= \mathrm{S}(c_t + \mathrm{SK}_{10}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{11}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot x_t[3] \\
x_{t+1}[5] &= \mathrm{S}(c_t + \mathrm{SK}_{13}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{14}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot x_t[3] \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot x_t[4] \\
x_{t+1}[6] &= \mathrm{S}(c_t + \mathrm{SK}_{17}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{18}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot x_t[3] \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot x_t[4] + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot x_t[5]
\end{aligned}$$

En effet, on peut voir que du fait de cette structure particulière, une faute sur $x_t[3]$ contaminera le registre Rx plus rapidement qu'une faute sur $x_t[4]$, et encore plus qu'une faute sur $x_t[5]$. Plus particulièrement, si on faute $x[3]$ à t, alors $x[4]$ et $x[5]$ seront affectés à $t+1$, et si on faute $x[4]$ à t, alors $x[5]$ sera affecté à $t+1$. Les équations de la section 8.5, page 84 montrent alors une conséquence de ce phénomène : les sous-clés qui interviennent dans les équations Δc_{t+3} pour des fautes sur $x[4]$ et $x[5]$ (équations (8.25) et (8.29)) interviennent également dans la différence Δc_{t+4} issue d'une faute sur $x[3]$ (équation 8.21). De même, les sous-clés intervenant dans la différence Δc_{t+3} provoquée par une faute sur $x_t[5]$ (équation (8.29)) se retrouvent également dans l'équation correspondant à Δc_{t+4} pour une faute sur $x_t[4]$ (équation (8.26)). On va donc procéder *en cascade*.

Pour une faute à t, intéressons-nous donc aux différences sur les chiffrés, à $t+4$, pour des fautes sur $x[3]$, $x[4]$ et $x[5]$, ce qui correspond aux équations (8.21), (8.26), et (8.30) respectivement :

$$\begin{aligned}
\Delta c_{t+4} &= \delta_3 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_3) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_4) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{20}) \Big) \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21}) \Big) \Big) \\
\Delta c_{t+4} &= \delta_4 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_4) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21}) \Big) \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \Big) \\
\Delta c_{t+4} &= \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \\
&\quad + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \Big)
\end{aligned}$$

Dans un premier temps, nous déterminerons SK_5 et SK_{21} à l'aide de fautes en $x_t[5]$, puis en combinant ces nouvelles valeurs trouvées avec des fautes en $x[4]$, nous déterminerons SK_4, SK_{16} et SK_{20}. Enfin, de la même façon, nous obtiendrons SK_3, SK_{12}, SK_{15} et SK_{19} en fautant $x[3]$.

8.6.3.1 Fautes en $x_t[5]$

$$\begin{cases}
\Delta c_{t+3} = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) + \mathrm{S}(c_t + \mathrm{SK}_{21}) \Big) \\
\Delta c_{t+4} = \delta_5 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_5) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \\
\qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \Big)
\end{cases}$$

En supposant que $\Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_0) \neq 1$ et $\mathrm{S}(c_{t+1} + \mathrm{SK}_6) \neq 1\Big)$, alors pour un δ_5 fixé, le système ci-dessus est linéaire et a une unique solution. Ainsi, en prenant les 15 valeurs non nulles de δ_5, on obtient donc un ensemble de 15 solutions pour (SK_5, SK_{21}). Par intersections de plusieurs tels ensembles, on trouve une seule solution. Il est à noter qu'une fois SK_5 et SK_{21} déterminés, la valeur de δ_5 pour chaque faute injectée est également connue : il s'agit du δ_5 associé au couple solution.

Expérimentalement, injecter une faute en $x[5]$ à trois reprises en moyenne suffit pour obtenir SK_5 et SK_{21}.

8.6.3.2 Fautes en $x_t[4]$

On dispose à présent des valeurs de SK_5 et SK_{21}. Le système suivant est donc linéaire, pour un δ_4 et un SK_{16} fixés.

$$\begin{cases} \Delta c_{t+3} = \delta_4 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_4) + \mathrm{S}(c_t + \mathrm{SK}_{20})\Big) \\ \Delta c_{t+4} = \delta_4 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_4) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \\ \qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big) \\ \qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6)\Big) \end{cases}$$

Comme dans la sous-section précédente, on calcule les valeurs possibles de SK_4 et SK_{20} pour chaque (δ_4, SK_{16}). Pour un δ_4 et un SK_{16} fixés, le système a une unique solution lorsque les deux équations sont linéairement indépendantes. Au total, pour une faute sur $x[4]$, on obtient donc un ensemble de $15 \times 16 = 240$ valeurs possibles pour (SK_4, SK_{16}, SK_{20}) (15 valeurs non nulles pour δ_4, 16 valeurs possibles pour SK_{16}).

Avec trois ensembles en moyenne, l'intersection donne une unique solution. Comme pour les fautes en $x[5]$, pour chaque faute injectée, la valeur de δ_4 est celle qui est associée au triplet solution.

8.6.3.3 Fautes en $x_t[3]$

On connaît à présent SK_4, SK_5, SK_{20} et SK_{21}. D'après (8.20) et (8.21) :

$$\begin{cases} \Delta c_{t+3} = \delta_3 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_3) + \mathrm{S}(c_t + \mathrm{SK}_{19})\Big) \\ \Delta c_{t+4} = \delta_3 \cdot \Big(\mathrm{S}(c_t + \mathrm{SK}_3) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_0) \\ \qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot \mathrm{S}(c_{t+1} + \mathrm{SK}_6) \\ \qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_4) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{20})\Big) \\ \qquad\qquad + \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot \Big(\mathrm{S}(c_{t+1} + \mathrm{SK}_5) + \mathrm{S}(c_{t+1} + \mathrm{SK}_{21})\Big)\Big) \end{cases}$$

À nouveau, on utilise les valeurs déterminées à l'étape précédente.
On a 15 valeurs non nulles pour δ_3, 16 valeurs possibles pour SK_{12} et pour SK_{15}. Donc, pour chaque triplet (δ_3, SK_{12}, SK_{15}), il existe une unique solution pour (SK_3, SK_{19}). Ainsi, on obtient pour chaque faute injectée $15 \times 16 \times 16 = 3840$ valeurs possibles pour (SK_3, SK_{12}, SK_{15}, SK_{19})

Nos tests ont montré qu'il fallait injecter une faute 5 fois en moyenne pour déterminer une unique solution.

Pour les six sous-clés restantes, on procède par recherche exhaustive, ce qui revient à essayer 2^{24} bits, dans le pire des cas. Pour avoir toutes les sous-clés, il faut donc essayer 2^{26} possibilités (on a aussi deux possibilités pour (SK_1, SK_{17}) et deux possibilités pour (SK_2, SK_{18})), et donc autant d'exécutions de l'algorithme.

On retrouve ainsi en fin de processus l'ensemble des bits de sous-clés en injectant environ 17 fautes, avec une complexité calculatoire réduite.

8.7 Conclusion du chapitre

Dans ce chapitre, nous avons à la fois détaillé un état de l'art sur les attaques en fautes sur les algorithmes de chiffrement par flot, et analysé les faiblesses d'une version jouet de notre algorithme SSSC innovant afin d'en tirer les leçons d'une manière constructive afin de bâtir un algorithme SSSC à grande dimension.

À notre connaissance, nous avons réalisé la première étude de sécurité vis-à-vis des attaques en faute sur un SSSC. Notre SSSC est intrinsèquement assez complexe à analyser, d'autant plus à grande dimension, car l'attaquant est en présence d'un système où respectivement la contrôlabilité et l'observabilité est très limitée : l'injection d'une faute conduit très vite à des équations très complexes (l'attaquant affronte donc une diffusion de sa faute très rapide) et l'attaquant ne peut observer qu'un quartet de sortie à la fois. La connaissance de l'état interne via l'utilisation d'observateurs pourrait lui permettre d'obtenir plus d'informations afin de lui faciliter son analyse, mais ce n'est pas le cas ici. On se retrouve donc au fond ici dans un cas classique d'indétermination en théorie du contrôle, dans ce qu'il est convenu d'appeler une estimation conjointe état interne/paramètres (représentés ici par la clé de chiffrement).

Plusieurs leçons intéressantes ont été tirées de notre analyse :

- Suivant le placement des boîtes-S et de la sortie, une faute injectée peut se propager plus facilement vers la sortie, et fournir la valeur de la faute injectée au cycle précédent à l'attaquant. Ce point est décisif, car si l'attaquant retrouve la valeur de la faute injectée, alors une inconnue est retirée de son système d'équations et l'analyse en fautes peut progresser de proche en proche.
- Un chemin linéaire de la faute jusqu'à la sortie facilite l'analyse. Une leçon à tirer est donc d'éviter de réaliser un simple XOR des composantes de la matrice sur la ligne du quartet de sortie. **Ainsi, lors de la conception de notre algorithme LPV-SSSC en dimension 40, les $r-1$ premières lignes, qui sont en fait les lignes par lesquelles passerait une faute potentiellement injectée avant d'être "sortie", ne contiennent plus de 1,** à part là où c'est obligatoire [2] afin de satisfaire la propriété de platitude.
- Une structure triangulaire de la matrice de chiffrement peut permettre une résolution d'équations "à la Gauss", et donc une structure irrégulière lui est préférable.
- Même si un algorithme de chiffrement LPV-SSSC peut être vu comme intrinsèquement assez complexe à analyser en fautes, cela ne doit pas empêcher un concepteur d'algorithmes LPV-SSSC de réaliser une étude complète de son système afin notamment de trouver le placement idéal pour ses boîtes-S et sa sortie.

2. Pour être plus précis, une sous-diagonale de la matrice de chiffrement.

9 Conclusion générale

9.1 Résumé des travaux réalisés

Ce livrable, qui est chronologiquement le dernier à livrer pour le projet THE CASCADE, permet de récapituler l'ensemble des résultats du projet.

Tout d'abord, les conditions de construction d'un système LPV-SSSC utilisable en cryptographie ont pu être décrites et ont donné lieu à une instanciation pratique sous forme matricielle. La construction de cette matrice de la version finale de notre algorithme a elle-même été justifiée.

Par la suite, les caractéristiques remarquables pour un algorithme auto-synchronisant ont pu être illustrées sur notre plate-forme de démonstration Arduino, notamment :

- À la suite d'une désynchronisation (accidentelle ou provoquée par un attaquant) lors d'une communication, le destinataire du message chiffré mettra un temps maximal borné pour se resynchroniser, et ce, sans avoir besoin de paramètres additionnels tels des vecteurs d'initialisation (*Initialization Vector*, IV) ou de compteurs de symboles.
- Chiffrer le même message clair avec deux étais internes initiaux différents conduisent au même message chiffré après synchronisation.

Cet algorithme, une fois implanté sur technologie ASIC 65 nm, conduit à des résultats en défaveur de notre algorithme en terme de nombre de portes logiques utilisées. Cet écart significatif avec des algorithmes aux propriétés équivalentes (Moustique) s'explique par le paradigme de construction du mécanisme de synchronisation de notre LPV-SSSC qui est en fait une matrice 40×40, qui certes n'a pas besoin d'instancier à chaque élément de la matrice un élément combinatoire (SBox ou Xor), mais qui demande tout de même intrinsèquement nettement plus de ressources qu'un Moustique. Ce dernier étant cassé, et n'ayant pas eu de successeur crédible depuis son cassage, on peut en conclure que cet écart de performance est peut-être le "prix à payer" pour pouvoir utiliser un algorithme auto-synchronisant sécurisé. Notons également que les résultats d'implémentations sur FPGA Xilinx sont beaucoup plus flatteurs en surface, car ils reflètent en fait l'excellente utilisation/occupation des *slices* du FPGA, les boîtes-S ayant une taille adaptée au nombre d'entrées-sorties de ses cellules élémentaires. Par ailleurs, le débit de LPV-SSSC est plus important sur FPGA que sur Moustique, et pour le coup, c'est un excellent résultat à mettre au crédit du consortium. Le fait que l'algorithme ait été conçu dès le départ pour travailler sur des quartets (avec des opérations effectuées dans $\mathrm{GF}(2^4)$) et qu'il puisse donc "sortir" 4 bits par itération/cycle d'horloge a été un choix judicieux du consortium.

Le consortium a également trouvé un nouveau type de formulation de la CPA qu'elle a pu appliquer sur LPV-SSSC. Ce résultat original dépasse le simple cadre des auto-synchronisants car cette attaque peut être mise en oeuvre sur un grand nombre d'algorithmes de chiffrements symétriques.

Une étude approfondie des méthodes de masquage des implémentations matérielles de LPV-SSSC a également amené des résultats intéressants en terme d'estimation du rapport implémentations masquées/implémentations non-masquées. Ce rapport est clairement prohibitif pour la méthode proposée par Coron. Là encore, c'est un résultat qui a un impact plus large que le seul cadre des algorithmes auto-synchronisants,

car à la connaissance du consortium, aucune étude de performance n'avait été préalablement effectuée sur les méthodes de masquage de l'état de l'art dans le monde matériel. C'est désormais chose faite, et il peut être constaté que certains articles de l'état de l'art sont plus dédiés à des implémentations logicielles et que leur transposition dans le domaine matériel est malaisé. Notamment, le débit d'aléatoire requis est souvent rédhibitoire pour pouvoir être concrètement appliqué à des cas industriels réels.

Enfin, les attaques en faute sur notre algorithme ont été également intensivement étudiées. Les attaques de l'état de l'art ne s'appliquant pas directement (cf. attaque de Piret *et al.*), il nous a fallu injecter "manuellement" des fautes, et constater à la sortie si ces fautes nous donnaient un résultat exploitable via une DFA. Intrinsèquement, la rapide diffusion de l'algorithme combinée au fait que nous avons une observabilité très faible sur le système compliquent considérablement la tâche de l'attaquant, ce qui est une très intéressante propriété pour notre algorithme. Nous avons donc commencé à mi-projet à étudier une version réduite de l'algorithme (dite "de dimension 7"). Cette première étude, détaillée dans ce livrable, nous a permis de trouver des conditions plus favorables que d'autres à l'exploitation des fautes injectées. Ce retour d'expérience nous a permis de proposer une version finale de LPV-SSSC compliquant la tâche de l'attaquant.

Avant de décrire dans la sous-section suivante les pistes de recherche, le lecteur pourra ainsi apprécier la grande diversité des plate-formes utilisées pendant le projet (ASIC, FPGA, carte à puce, Arduinos), le nombre d'itérations de notre algorithme étudiées et implémentées, ainsi que la combinaison des études des attaques actives et passives.

9.2 Pistes de recherche

Plusieurs pistes de recherche sont concevables à la suite de ces travaux.

9.2.1 Optimisation des performances

Une première série de sujets concernent l'optimisation des implémentations, afin de tenter de combler quelque peu l'écart de performances entre LPV-SSSC et son défunt concurrent Moustique. Concernant l'implémentation de la matrice d'état interne LPV-SSSC, il reste le problème ouvert de trouver une matrice contenant moins de 80 éléments mais garantissant les mêmes propriétés statistiques que pour 80. De même, il faudrait idéalement pouvoir trouver une matrice pour lesquels les éléments sont disposés de telle sorte que le routage inter-éléments soit minimal. Une meilleure mutualisation entre chiffrement et déchiffrement est également envisageable mais demanderait une modification de la construction des matrices du système LPV-SSSC. D'autres alternatives, utilisant une matrice de dimension moins importante (ex. : 20) mais avec des SBoxes plus grandes (ex. : 8×8 bits) pourraient également être investiguées.

9.2.2 Protection contre les attaques physiques sur composant

Les algorithmes auto-synchronisants ont des propriétés intrinsèquement intéressantes pour se protéger contre les attaques physiques : les attaquants ont besoin d'une synchronisation parfaite pour pouvoir aligner leurs relevés de canaux auxiliaires ou bien injecter au moment opportun une faute. Or, un algorithme auto-synchronisant est susceptible de ne pas prendre constamment le même temps de synchronisation suivant son état initial, sa clé et les symboles en cours de traitement. Ce comportement est même amplifié avec

l'utilisation de chiffreurs dits "statistiques" [84]. Une étude complète pourrait être menée afin de trouver un chiffreur statistique facilement implémentable et ayant une large plage de temps de synchronisation possibles. Il faut noter qu'à elle seule, ce type de contre-mesure (appelée "*hiding/shuffling*") ne suffit pas à elle seule à contrer définitivement les attaques SCAs, mais elle oblige l'attaquant à prendre plus de mesures [1] ou à injecter plus de fautes.

Pour ce qui concerne la protection contre les attaques actives, on pourrait également tenter d'aller un cran plus loin dans la combinaison entre théorie du contrôle et cryptographie. Par exemple, on pourrait imaginer tirer parti du fait que le concepteur du circuit (le défenseur) puisse contrôler l'évolution du système via l'utilisation d'observateurs (considérés ici au sens de la théorie du contrôle) focalisés sur l'état interne du système et permettant de détecter des fautes injectées. De même, un lien entre diagnostic (considéré là aussi ici au sens de la théorie du contrôle) des systèmes et détection d'attaques en faute pourrait être approfondi.

1. Une première estimation donne que, sans pré-traitement particulier sur les courbes de canaux auxiliaires obtenues, il devra traiter approximativement n^2 mesures, où n est le nombre de moments de synchronisation possibles pour l'attaquant, afin d'obtenir une corrélation équivalente au cas où il n'y a pas de désynchronisation.

Bibliographie

[1] P. Apkarian, P. Gahinet, and G. Becker, "A convex characterization of gain-scheduled H_∞ Controllers," *IEEE Transactions on Automatic Control*, vol. 40, pp. 853–864, May 1995.

[2] M. Farhood and G. Dullerud, "Control of nonstationary LPV systems," *Automatica*, vol. 44, pp. 2108–2119, 2010.

[3] S. M. Lee and J. H. Park, "Output feedback model predictive control for LPV systems using parameter-dependent Lyapunov-function," *Applied Mathematics and Computation*, vol. 190, pp. 671–676, 2007.

[4] D. Leith and W. Leithead, "Survey of gain scheduling analysis and design," *International Journal of Control*, vol. 73, pp. 1001–1025, 2000.

[5] A. Packard and G. Balas, "Theory and application of linear parameter-varying control techniques," *Proceedings of the American Control Conference, Tutorial workshop*, 1997.

[6] C. Scherer, "LPV control and full block multipliers," *Automatica*, vol. 37, pp. 361–375, 2001.

[7] G. I. Bara, J. Daafouz, F. Kratz, and J. Ragot, "Parameter-dependent state observer design for affine LPV systems," *International Journal of Control*, vol. 74, no. 16, pp. 1601–1611, 2001.

[8] M. Heemels, J. Daafouz, and G. Millérioux, "Observer-based control of discrete-time LPV systems with uncertain parameters," *IEEE Trans. Autom. Contr.*, vol. 55, pp. 2130–2135, September 2010.

[9] M. Sato, "Filter design for LPV systems using quadratically parameter-dependent lyapunov functions," *Automatica*, vol. 42, pp. 2017–2023, 2006.

[10] R. Toth, F. Felici, P. Heuberger, and P. M. J. V. den Hof, "Discrete time LPV I/O and state-space representations, differences of behavior and pitfalls of interpolation," in *Proc. of the European Control Conference, (ECC 2007)*, (Kos, Greece), IEEE, 2007.

[11] W. Rugh and J. Shamma, "Research on gain scheduling," *Automatica*, vol. 36, pp. 1401–1425, 2000.

[12] O. Sename, P. Gaspar, and J. Bokor, eds., *Robust Control and Linear Parameter Varying Approaches, Application to Vehicle Dynamics*, vol. 437 of *Lecture Notes in Control and Information Sciences*. Springer, 2013.

[13] F. Bruzelius, *Linear Parameter-Varying Systems : an approach to gain scheduling*. PhD thesis, Department of Signals and Systems, Chalmers University of Technology, Göteborg, Sweden, 2004.

[14] G. Millerioux, F. Anstett, and G. Bloch, "Considering the attractor structure of chaotic maps for observer-based synchronization problems," *Mathematics and Computers in Simulation*, vol. 68, pp. 67–85, February 2005.

[15] A. Chamseddine, Y. Zhang, C. Rabbath, C. Join, and D. Theillol, "Flatness-based trajectory planning/replanning for a quadrator unmanned aerial vehicle," *IEEE Trans. on Aerospace and electronic systems*, 2012.

[16] T. Meurer, "Flatness-based trajectory planning for diffusion-reaction systems in a parallelepipedon - a spectral approach," *Automatica*, vol. 47, no. 5, pp. 935 – 949, 2011.

[17] J. De Donà, F. Suryawan, M. Seron, and J. Lévine, "A flatness-based iterative method for reference trajectory generation in constrained NMPC," in *Nonlinear Model Predictive Control* (L. Magni, D. M. Raimondo, and F. Allgöwer, eds.), vol. vol. 384 of *Lecture Notes in Control and Information Sciences*, pp. 325–333, Springer Berlin Heidelberg, 2009.

[18] M. Fliess and R. Marquez, "Toward a module theoretic approach to discrete-time linear predictive control," *International Journal of Control*, vol. 73, pp. 606–623, 2000.

[19] C. Kandler, S. Ding, T. Koenings, and N. Weihold, "A differential flatness based model predictive control approach," in *Proc. of IEEE International Conference on Control Applications (CCA)*, (Dubrovnik), 2012.

[20] G. Millérioux and J. Daafouz, "Flatness of switched linear discrete-time systems," *IEEE Trans. on Automatic Control*, vol. 54, pp. 615–619, March 2009.

[21] J. Parriaux and G. Millérioux, "Nilpotent semigroups for the characterization of flat outputs of switched linear and lpv discrete-time systems," *Systems and Control Letters*, vol. 62, no. 8, pp. 679–685, 2013.

[22] B. Dravie, T. Boukhobza, and G. Millérioux, "A mixed algebraic/graph-oriented approach for flatness of SISO LPV discrete-time systems," in *American Control Conference - ACC 2017- The 2017 American Control Conference, Seattle, WA, USA, May 24-26, 2017*, 2017.

[23] V. D. Blondel and J. N. Tsitsiklis, "When is a pair of matrices mortal ?," *Information Processing Letters*, vol. 63, pp. 283–286, 1997.

[24] O. Bournez and M. Branicky, "The mortality problem for matrices of low dimensions," *Theory of Computing Systems*, vol. 35, no. 4, pp. 433–448, 2002.

[25] M. S. Paterson, "Unsolvability in 3×3 matrices," *Studies in Applied Mathematics*, vol. 49, no. 1, pp. 105–107, 1970.

[26] H. Radjavi and P. Rosenthal, *Simultaneous Triangularization*. Springer, 2000.

[27] T. Boukhobza and G. Millérioux, "Graphical characterization of the set of all flat outputs for structured linear discrete-time systems," in *6th Symposium on System Structure and Control, (SSSC 2016)*, (Istanbul, Turkey), June 2015.

[28] U. M. Maurer, "New approaches to the design of self-synchronizing stream ciphers," in *Advances in Cryptology - EUROCRYPT '91, Workshop on the Theory and Application of of Cryptographic Techniques, Brighton, UK, April 8-11, 1991, Proceedings* (D. W. Davies, ed.), vol. vol. 547 of *Lecture Notes in Computer Science*, pp. 458–471, Springer, 1991.

[29] J. Daemen and P. Kitsos, "The self-synchronizing stream cipher mosquito : estream documentation, version 2," tech. rep., e-Stream Project, 2005. Available at :www.ecrypt.eu.org/stream/p3ciphers/mosquito/mosquito.pdf.

[30] J. Daemen and P. Kitsos, "The self-synchronizing stream cipher moustique," in *New Stream Cipher Designs - The eSTREAM Finalists* (M. J. B. Robshaw and O. Billet, eds.), vol. vol. 4986 of *Lecture Notes in Computer Science*, pp. 210–223, Springer, 2008.

[31] P. Hawkes, M. Paddon, G. G. Rose, and W. V. Miriam, "Primitive specification for sss," tech. rep., e-Stream Project, 2004. Available at : http ://www.ecrypt.eu.org/stream/ciphers/sss/sss.pdf.

[32] P. Sarkar, "Hiji-bij-bij : A new stream cipher with a self-synchronizing mode of operation," *IACR Cryptology ePrint Archive*, vol. vol. 2003, p. 14, 2003.

[33] J. Daemen, J. Lano, and B. Preneel, "Chosen ciphertext attack on SSS," *eSTREAM, ECRYPT Stream Cipher Project, Report 2005/044*, June 2005. Available online at http ://www.ecrypt.eu.org/stream/papers.html/044.pdf.

[34] A. Joux and F. Muller, "Chosen-ciphertext attack against mosquito," *Lecture Note in Computer Science*, 2006.

[35] A. Joux and F. Muller, "Loosening the KNOT," in *Fast Software Encryption, 10th International Workshop, FSE 2003, Lund, Sweden, February 24-26, 2003, Revised Papers*, pp. 87–99, 2003.

[36] A. Joux and F. Muller, "Two attacks against the HBB stream cipher," in *Fast Software Encryption : 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers*

(H. Gilbert and H. Handschuh, eds.), vol. vol. 3557 of *Lecture Notes in Computer Science*, pp. 330–341, Springer, 2005.

[37] V. Klíma, "Cryptanalysis of hiji-bij-bij (HBB)," *IACR Cryptology ePrint Archive*, vol. vol. 2005, p. 3, 2005.

[38] A. Klimov and A. Shamir, "New cryptographic primitives based on multiword t-functions," in *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers* (B. K. Roy and W. Meier, eds.), vol. vol. 3017 of *Lecture Notes in Computer Science*, pp. 1–15, Springer, 2004.

[39] B. Dravie, P. Guillot, and G. Millérioux, "Design of self-synchronizing stream ciphers : a new control-theoretical paradigm," in *International Federation of Automatic Control - IFAC 2017- The 20th IFAC World Congress, Toulouse, France, July 9-14, 2017*, 2017.

[40] T. P. Berger and M. Minier, "Some results using the matrix methods on impossible, integral and zero-correlation distinguishers for feistel-like ciphers," in *Progress in Cryptology - INDOCRYPT 2015 - 16th International Conference on Cryptology in India, Bangalore, India, December 6-9, 2015, Proceedings* (A. Biryukov and V. Goyal, eds.), vol. 9462 of *Lecture Notes in Computer Science*, pp. 180–197, Springer, 2015.

[41] T. Suzaki and K. Minematsu, "Improving the generalized feistel," vol. 6147 of *Fast Software Encryption – FSE*, pp. 19–39, Springer, 2010.

[42] F. Arnault, T. P. Berger, M. Minier, and B. Pousse, "Revisiting LFSRs for cryptographic applications," *IEEE Trans. Information Theory*, vol. vol. 57, no. 12, pp. 8095–8113, 2011.

[43] T. P. Berger, M. Minier, and G. Thomas, "Extended generalized Feistel networks using matrix representation," in *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers* (T. Lange, K. E. Lauter, and P. Lisonek, eds.), vol. vol. 8282 of *Lecture Notes in Computer Science*, pp. 289–305, Springer, 2013.

[44] M. E. Hellman, "A cryptanalytic time-memory trade-off," *IEEE Trans. Information Theory*, vol. 26, no. 4, pp. 401–406, 1980.

[45] M. Hell, T. Johansson, and W. Meier, "Grain : a stream cipher for constrained environments," *IJWMC*, vol. vol. 2, no. 1, pp. 86–93, 2007.

[46] C. D. Cannière, "Trivium : A stream cipher construction inspired by block cipher design principles," in *Information Security, 9th International Conference, ISC 2006, Samos Island, Greece, August 30 - September 2, 2006, Proceedings*, pp. 171–186, 2006.

[47] J. Daemen and P. Kitsos, "The self-synchronizing stream cipher moustique," *eSTREAM, ECRYPT Stream Cipher Project*, June 2005. Available online at http ://www.ecrypt.eu.org/stream.

[48] J. Daemen, G. Rouvroy, and J.-J. Quisquater, "FPGA Implementations of the DES and Triple-DES Masked Against Power Analysis Attacks," *International Conference on Field Programmable Logic and Applications (FPL)*, 2006.

[49] E. Brier, C. Clavier, and F. Olivier, "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems - CHES 2004 : 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pp. 16–29, 2004.

[50] P. C. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings* (M. J. Wiener, ed.), vol. vol. 1666 of *Lecture Notes in Computer Science*, pp. 388–397, Springer, 1999.

[51] N. DES, "Fips publication 46-3 - Data Encryption Standard," October 25, 1999.

[52] N. AES, "FIPS publication 197 - advanced encryption standard," November 26, 2001.

[53] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[54] M. Ferrante and N. Frigo, "A note on the coupon - collector's problem with multiple arrivals and the random sampling." arXiv :1209.2667 [math.PR], 12 Sep 2012.

[55] J. Coron, "Higher order masking of look-up tables," in *EUROCRYPT*, pp. 441–458, 2014.

[56] M. Varchola and M. Drutarovsky, "New high entropy element for fpga based true random number generators," in *Cryptographic Hardware and Embedded Systems - CHES*, pp. 351–365, 2010.

[57] D. Boneh, R. A. Demillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," in *EUROCRYPT '97*, 1997.

[58] E. Biham and A. Shamir, "Differential Fault Analysis of Secret Key Cryptosystems," *Advances in Cryptology (CRYPTO '97)*, 1997.

[59] J. J. Hoch and A. Shamir, "Fault Analysis of Stream Ciphers," in *CHES*, pp. 240–253, 2004.

[60] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan, "The Sorcerer's Apprentice Guide to Fault Attacks," *IACR Cryptology ePrint Archive*, vol. 2004, p. 100, 2004.

[61] M. Joye and M. Tunstall, eds., *Fault Analysis in Cryptography*. Information Security and Cryptography, Springer, 2012.

[62] M. Otto, *Fault Attacks and Countermeasures*. PhD thesis, University of Paderborn, 2005.

[63] C. H. Kim and J.-J. Quisquater, *"Fault Attacks for CRT Based RSA : New Attacks, New Results, and New Countermeasures"*, pp. 215–228. Berlin, Heidelberg : Springer Berlin Heidelberg, 2007.

[64] E. Biham, L. Granboulan, and P. Nguyen, "Impossible Fault Analysis of RC4 and Differential Analysis of RC4," in *FSE*, pp. 359–367, 2005.

[65] H. Finney, "An RC4 Cycle That Can't Happen," September 1994.

[66] I. Mantin, *Analysis of the Stream Cipher RC4*. PhD thesis, The Weizmann Institute of Science, 2001.

[67] C. D. Cannire and B. Preneel, "Trivium - A Stream Cipher Construction Inspired by Block Cipher Principles," *eSTREAM submitted papers*, 2006.

[68] M. Hojsík and B. Rudolf, *Differential Fault Analysis of Trivium*, pp. 158–172. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008.

[69] M. Hojsík and B. Rudolf, *Floating Fault Analysis of Trivium*, pp. 239–250. Berlin, Heidelberg : Springer Berlin Heidelberg, 2008.

[70] A. Kircanski and A. M. Youssef, *Differential Fault Analysis of HC-128*, pp. 261–278. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010.

[71] A. Kircanski and A. Youssef, *Differential Fault Analysis of Rabbit*, pp. 197–214. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009.

[72] A. Berzati, C. Canovas-Dumas, and L. Goubin, "Fault Analysis of Rabbit : Toward a Secret Key Leakage," in *Proceedings of the 10th International Conference on Cryptology in India : Progress in Cryptology*, INDOCRYPT '09, (Berlin, Heidelberg), pp. 72–87, Springer-Verlag, 2009.

[73] A. Berzati, C. Canovas, G. Castagnos, B. Debraize, L. Goubin, A. Gouget, P. Paillier, and S. Salgado, "Fault analysis of GRAIN-128," in *Hardware-Oriented Security and Trust, 2009. HOST '09. IEEE International Workshop on*, pp. 7–14, July 2009.

[74] S. Karmakar and D. R. Chowdhury, "Fault Analysis of Grain Family of Stream Ciphers," *IACR Cryptology ePrint Archive*, vol. 2014, p. 261, 2014.

[75] S. Karmakar and D. R. Chowdhury, "Differential Fault Analysis of MICKEY Family of Stream Ciphers," *IACR Cryptology ePrint Archive*, vol. 2014, p. 262, 2014.

[76] Y. E. Salehani, A. Kircanski, and A. M. Youssef, "Differential Fault Analysis of Sosemanuk," in *AFRICACRYPT*, 2011.

[77] D. Roy and S. Mukhopadhyay, "Fault analysis and weak key-IV attack on Sprout," *IACR Cryptology ePrint Archive*, vol. 2016, p. 207, 2016.

[78] G. Piret and J. J. Quisquater, "A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD," in *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, vol. 2779 of *Lecture Notes in Computer Science*, pp. 77–88, Springer, 2003.

[79] T. Fuhr, E. Jaulmes, V. Lomné, and A. Thillard, "Fault Attacks on AES with Faulty Ciphertexts Only," in *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pp. 108–118, Aug 2013.

[80] J. Daemen, J. Lano, and B. Preneel, "Chosen Ciphertext Attack on SSS," *eSTREAM submitted papers*, Juin 2005.

[81] J. Antoine and F. Muller, *Loosening the KNOT*, pp. 87–99. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003.

[82] A. Joux and F. Muller, "Chosen-Ciphertext Attacks Against MOSQUITO," in *Proceedings of the 13th International Conference on Fast Software Encryption*, FSE'06, (Berlin, Heidelberg), pp. 390–404, Springer-Verlag, 2006.

[83] E. Käsper, V. Rijmen, T. E. Bjørstad, C. Rechberger, M. Robshaw, and G. Sekar, "Correlated keystreams in moustique," in *Proceedings of the Cryptology in Africa 1st International Conference on Progress in Cryptology*, AFRICACRYPT'08, (Berlin, Heidelberg), pp. 246–257, Springer-Verlag, 2008.

[84] H. W. Heys, "An analysis of the statistical self-synchronization of stream ciphers," in *Proc. of INFOCOM*, pp. 897–904, 2001.

A Miscellaneous

A.1 Schémas

A.1.1 Répartition des sous-clés

Le tableau A.1 décrit la position des sous-clés SK_i et des coefficients 1 dans la matrice A. La case i correspond à l'indice des lignes. Chaque valeur entre parenthèses correspond à l'indice de la colonne d'une sous-clé SK.

Par exemple :
SK0, (8) sur la ligne 0 signifie que la sous-clé SK0 est à la ligne 0 colonne 8.
SK11, (5) sur la ligne 7 signifie que la sous-clé SK11 est à la ligne 7 colonne 5.
1, (0) sur la ligne 1 signifie qu'on a un coefficient constant 1 sur la ligne 0 colonne 0.
Les sous-clés sont notées par ordre croissant suivant les lignes et à partir de la ligne 0.

A.1.2 Équations

Équation chiffreur :

j \ i									
0	1,(0) 1,(9) 1,(19) 1,(28) 1,(37)	1,(1) 1,(10) 1,(20) 1,(29) 1,(38)	1,(2) 1,(11) 1,(21) 1,(30) 1,(39)	1,(3) 1,(12) 1,(22) 1,(31) SK0,(8)	1,(4) 1,(13) 1,(23) 1,(32) SK1,(18)	1,(5) 1,(14) 1,(24) 1,(33)	1,(6) 1,(15) 1,(25) 1,(34)	1,(7) 1,(16) 1,(26) 1,(35)	 1,(17) 1,(27) 1,(36)
1	1,(0)	SK2,(7)	SK3,(38)	SK4,(39)					
2	1,(1)	SK5,(2)							
3	1,(1)	SK6,(2)							
4	1,(2)	SK7,(3)							
5	1,(3)	SK8,(4)							
6	1,(4)	SK9,(5)							
7	SK10,(1)	SK11,(5)	SK12,(6)						
8	1,(6)	SK13,(7)							
9	1,(7)	SK14,(1)	SK15,(8)						
10	1,(8)	SK16,(9)							
11	1,(9)	SK17,(1)	SK18,(10)						
12	1,(10)	SK19,(11)							
13	1,(11)	SK20,(12)							
14	1,(12)	SK21,(13)							
15	1,(13)	SK22,(5)	SK23,(7)	SK24,(14)					
16	1,(14)	SK25,(6)	SK26,(15)						
17	1,(15)	SK27,(1)	SK28,(6)	SK29,(14)	SK30,(16)				
18	1,(16)	SK31,(13)	SK32,(17)						
19	1,(17)	SK33,(18)							
20	1,(18)	SK34,(5)	SK35,(6)	SK36,(19)					
21	1,(19)	SK37,(1)	SK38,(20)						
22	1,(20)	SK39,(17)	SK40,(21)						
23	1,(21)	SK41,(22)							
24	1,(22)	SK42,(23)							
25	1,(23)	SK43,(2)	SK44,(3)	SK45,(18)	SK46,(24)				
26	1,(24)	SK47,(11)	SK48,(25)						
27	1,(25)	SK49,(3)	SK50,(26)						
28	1,(26)	SK51,(7)	SK52,(27)						
29	1,(27)	SK53,(23)	SK54,(28)						
30	1,(28)	SK55,(23)	SK56,(29)						
31	1,(29)	SK57,(2)	SK58,(15)	SK59,(30)					
32	1,(30)	SK60,(14)	SK61,(28)	SK62,(31)					
33	1,(31)	SK63,(27)	SK64,(32)						
34	1,(32)	SK65,(14)	SK66,(16)	SK67,(21)	SK68,(33)				
35	1,(33)	SK69,(34)							
36	1,(34)	SK70,(35)							
37	1,(35)	SK71,(7)	SK72,(36)						
38	1,(36)	SK73,(20)	SK74,(26)	SK75,(37)					
39	1,(37)	SK76,(30)	SK77,(35)	SK78,(36)	SK79,(38)				

TABLE A.1 – Table sous clés

$$
\begin{aligned}
x_{t+1}[0] &= x_t[0] + x_t[1] + x_t[2] + x_t[3] + \\
&\quad x_t[4] + x_t[5] + x_t[6] + x_t[7] + x_t[9] + \\
&\quad x_t[10] + x_t[11] + x_t[12] + x_t[13] + x_t[14] + \\
&\quad x_t[15] + x_t[16] + x_t[17] + x_t[19] + x_t[20] + \\
&\quad x_t[21] + x_t[22] + x_t[23] + x_t[24] + x_t[25] + \\
&\quad x_t[26] + x_t[27] + x_t[28] + x_t[29] + x_t[30] + \\
&\quad x_t[31] + x_t[32] + x_t[33] + x_t[34] + x_t[35] + \\
&\quad x_t[36] + x_t[37] + x_t[38] + x_t[39] + \mathrm{S}(c_t + \mathrm{SK}_0) \cdot x_t[8] + \mathrm{S}(c_t + \mathrm{SK}_1) \cdot x_t[18] \\
x_{t+1}[1] &= x_t[0] + \mathrm{S}(c_t + \mathrm{SK}_2) \cdot x_t[7] + \mathrm{S}(c_t + \mathrm{SK}_3) \cdot x_t[38] + \mathrm{S}(c_t + \mathrm{SK}_4) \cdot x_t[39] \\
x_{t+1}[2] &= x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_5) \cdot x_t[2] \\
x_{t+1}[3] &= x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_6) \cdot x_t[2] \\
x_{t+1}[4] &= x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_7) \cdot x_t[3] \\
x_{t+1}[5] &= x_t[3] + \mathrm{S}(c_t + \mathrm{SK}_8) \cdot x_t[4] \\
x_{t+1}[6] &= x_t[4] + \mathrm{S}(c_t + \mathrm{SK}_9) \cdot x_t[5] \\
x_{t+1}[7] &= \mathrm{S}(c_t + \mathrm{SK}_{10}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{11}) \cdot x_t[5] + \mathrm{S}(c_t + \mathrm{SK}_{12}) \cdot x_t[6] \\
x_{t+1}[8] &= x_t[6] + \mathrm{S}(c_t + \mathrm{SK}_{13}) \cdot x_t[7] \\
x_{t+1}[9] &= x_t[7] + \mathrm{S}(c_t + \mathrm{SK}_{14}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{15}) \cdot x_t[8] \\
x_{t+1}[10] &= x_t[8] + \mathrm{S}(c_t + \mathrm{SK}_{16}) \cdot x_t[9] \\
x_{t+1}[11] &= x_t[9] + \mathrm{S}(c_t + \mathrm{SK}_{17}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{18}) \cdot x_t[10] \\
x_{t+1}[12] &= x_t[10] + \mathrm{S}(c_t + \mathrm{SK}_{19}) \cdot x_t[11] \\
x_{t+1}[13] &= x_t[11] + \mathrm{S}(c_t + \mathrm{SK}_{20}) \cdot x_t[12] \\
x_{t+1}[14] &= x_t[12] + \mathrm{S}(c_t + \mathrm{SK}_{21}) \cdot x_t[13] \\
x_{t+1}[15] &= x_t[13] + \mathrm{S}(c_t + \mathrm{SK}_{22}) \cdot x_t[5] + \mathrm{S}(c_t + \mathrm{SK}_{23}) \cdot x_t[7] + \mathrm{S}(c_t + \mathrm{SK}_{24}) \cdot x_t[14] \\
x_{t+1}[16] &= x_t[14] + \mathrm{S}(c_t + \mathrm{SK}_{25}) \cdot x_t[6] + \mathrm{S}(c_t + \mathrm{SK}_{26}) \cdot x_t[15] \\
x_{t+1}[17] &= x_t[15] + \mathrm{S}(c_t + \mathrm{SK}_{27}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{28}) \cdot x_t[6] + \mathrm{S}(c_t + \mathrm{SK}_{29}) \cdot x_t[14] + \mathrm{S}(c_t + \mathrm{SK}_{30}) \cdot x_t[16] \\
x_{t+1}[18] &= x_t[16] + \mathrm{S}(c_t + \mathrm{SK}_{31}) \cdot x_t[13] + \mathrm{S}(c_t + \mathrm{SK}_{32}) \cdot x_t[17] \\
x_{t+1}[19] &= x_t[17] + \mathrm{S}(c_t + \mathrm{SK}_{33}) \cdot x_t[18] \\
x_{t+1}[20] &= x_t[18] + \mathrm{S}(c_t + \mathrm{SK}_{34}) \cdot x_t[5] + \mathrm{S}(c_t + \mathrm{SK}_{35}) \cdot x_t[6] + \mathrm{S}(c_t + \mathrm{SK}_{36}) \cdot x_t[19]
\end{aligned}
$$

$$
\begin{aligned}
x_{t+1}[21] &= x_t[19] + \mathrm{S}(c_t + \mathrm{SK}_{37}) \cdot x_t[1] + \mathrm{S}(c_t + \mathrm{SK}_{38}) \cdot x_t[20] \\
x_{t+1}[22] &= x_t[20] + \mathrm{S}(c_t + \mathrm{SK}_{39}) \cdot x_t[17] + \mathrm{S}(c_t + \mathrm{SK}_{40}) \cdot x_t[21] \\
x_{t+1}[23] &= x_t[21] + \mathrm{S}(c_t + \mathrm{SK}_{41}) \cdot x_t[22] \\
x_{t+1}[24] &= x_t[22] + \mathrm{S}(c_t + \mathrm{SK}_{42}) \cdot x_t[23] \\
x_{t+1}[25] &= x_t[23] + \mathrm{S}(c_t + \mathrm{SK}_{43}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{44}) \cdot x_t[3] + \mathrm{S}(c_t + \mathrm{SK}_{45}) \cdot x_t[18] + \mathrm{S}(c_t + \mathrm{SK}_{46}) \cdot x_t[24] \\
x_{t+1}[26] &= x_t[24] + \mathrm{S}(c_t + \mathrm{SK}_{47}) \cdot x_t[11] + \mathrm{S}(c_t + \mathrm{SK}_{48}) \cdot x_t[25] \\
x_{t+1}[27] &= x_t[25] + \mathrm{S}(c_t + \mathrm{SK}_{49}) \cdot x_t[3] + \mathrm{S}(c_t + \mathrm{SK}_{50}) \cdot x_t[26] \\
x_{t+1}[28] &= x_t[26] + \mathrm{S}(c_t + \mathrm{SK}_{51}) \cdot x_t[7] + \mathrm{S}(c_t + \mathrm{SK}_{52}) \cdot x_t[27] \\
x_{t+1}[29] &= x_t[27] + \mathrm{S}(c_t + \mathrm{SK}_{53}) \cdot x_t[23] + \mathrm{S}(c_t + \mathrm{SK}_{54}) \cdot x_t[28] \\
x_{t+1}[30] &= x_t[28] + \mathrm{S}(c_t + \mathrm{SK}_{55}) \cdot x_t[23] + \mathrm{S}(c_t + \mathrm{SK}_{56}) \cdot x_t[29] \\
x_{t+1}[31] &= x_t[29] + \mathrm{S}(c_t + \mathrm{SK}_{57}) \cdot x_t[2] + \mathrm{S}(c_t + \mathrm{SK}_{58}) \cdot x_t[15] + \mathrm{S}(c_t + \mathrm{SK}_{59}) \cdot x_t[30] \\
x_{t+1}[32] &= x_t[30] + \mathrm{S}(c_t + \mathrm{SK}_{60}) \cdot x_t[14] + \mathrm{S}(c_t + \mathrm{SK}_{61}) \cdot x_t[28] + \mathrm{S}(c_t + \mathrm{SK}_{62}) \cdot x_t[31] \\
x_{t+1}[33] &= x_t[31] + \mathrm{S}(c_t + \mathrm{SK}_{63}) \cdot x_t[27] + \mathrm{S}(c_t + \mathrm{SK}_{64}) \cdot x_t[32] \\
x_{t+1}[34] &= x_t[32] + \mathrm{S}(c_t + \mathrm{SK}_{65}) \cdot x_t[14] + \mathrm{S}(c_t + \mathrm{SK}_{66}) \cdot x_t[16] + \mathrm{S}(c_t + \mathrm{SK}_{67}) \cdot x_t[21] + \mathrm{S}(c_t + \mathrm{SK}_{68}) \cdot x_t[33] \\
x_{t+1}[35] &= x_t[33] + \mathrm{S}(c_t + \mathrm{SK}_{69}) \cdot x_t[34] \\
x_{t+1}[36] &= x_t[34] + \mathrm{S}(c_t + \mathrm{SK}_{70}) \cdot x_t[35] \\
x_{t+1}[37] &= x_t[35] + \mathrm{S}(c_t + \mathrm{SK}_{71}) \cdot x_t[7] + \mathrm{S}(c_t + \mathrm{SK}_{72}) \cdot x_t[36] \\
x_{t+1}[38] &= x_t[36] + \mathrm{S}(c_t + \mathrm{SK}_{73}) \cdot x_t[20] + \mathrm{S}(c_t + \mathrm{SK}_{74}) \cdot x_t[26] + \mathrm{S}(c_t + \mathrm{SK}_{75}) \cdot x_t[37] \\
x_{t+1}[39] &= x_t[37] + \mathrm{S}(c_t + \mathrm{SK}_{76}) \cdot x_t[30] + \mathrm{S}(c_t + \mathrm{SK}_{77}) \cdot x_t[35] + \mathrm{S}(c_t + \mathrm{SK}_{78}) \cdot x_t[36] + \mathrm{S}(c_t + \mathrm{SK}_{79}) \cdot x_t[38]
\end{aligned}
$$

Équation déchiffreur :

$$
\begin{aligned}
\widehat{m}_t \;=\;& \mathrm{S}(c_{t-1}+\mathrm{SK}_5)\cdot\widehat{x}_t[0]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{10})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)+\mathrm{S}(c_{t-2}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[1]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[2]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{11})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)\Big)\cdot\widehat{x}_t[5]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{12})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)\Big)\cdot\widehat{x}_t[6]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_2)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[7]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{73})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[20]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{74})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[26]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{76})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[30]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{77})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[35]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{78})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)+\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[36]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{75})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)+\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[37]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{79})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)+\mathrm{S}(c_{t-3}+\mathrm{SK}_3)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[38]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_4)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[39]\\
&+c_t+\\
&\widehat{x}_t[0]+\widehat{x}_t[1]+\widehat{x}_t[2]+\widehat{x}_t[3]+\\
&\widehat{x}_t[4]+\widehat{x}_t[5]+\widehat{x}_t[6]+\widehat{x}_t[7]+\widehat{x}_t[9]+\\
&\widehat{x}_t[10]+\widehat{x}_t[11]+\widehat{x}_t[12]+\widehat{x}_t[13]+\widehat{x}_t[14]+\\
&\widehat{x}_t[15]+\widehat{x}_t[16]+\widehat{x}_t[17]+\widehat{x}_t[19]+\widehat{x}_t[20]+\\
&\widehat{x}_t[21]+\widehat{x}_t[22]+\widehat{x}_t[23]+\widehat{x}_t[24]+\widehat{x}_t[25]+\\
&\widehat{x}_t[26]+\widehat{x}_t[27]+\widehat{x}_t[28]+\widehat{x}_t[29]+\widehat{x}_t[30]+\\
&\widehat{x}_t[31]+\widehat{x}_t[32]+\widehat{x}_t[33]+\widehat{x}_t[34]+\widehat{x}_t[35]+\\
&\widehat{x}_t[36]+\widehat{x}_t[37]+\widehat{x}_t[38]+\widehat{x}_t[39]+\mathrm{S}(c_t+\mathrm{SK}_0)\cdot\widehat{x}_t[8]+\mathrm{S}(c_t+\mathrm{SK}_1)\cdot\widehat{x}_t[18]
\end{aligned}
$$

et

$$
\begin{aligned}
\widehat{x}_{t+1}[0] \;=\;& \mathrm{S}(c_{t-1}+\mathrm{SK}_5)\cdot\widehat{x}_t[0]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{10})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)+\mathrm{S}(c_{t-2}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[1]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_5)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[2]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{11})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)\Big)\cdot\widehat{x}_t[5]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{12})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_2)\Big)\cdot\widehat{x}_t[6]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_2)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[7]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{73})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[20]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{74})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[26]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{76})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[30]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{77})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[35]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{78})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)+\mathrm{S}(c_{t-2}+\mathrm{SK}_3)\Big)\cdot\widehat{x}_t[36]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{75})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_3)+\mathrm{S}(c_{t-2}+\mathrm{SK}_4)\Big)\cdot\widehat{x}_t[37]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_{79})\cdot\mathrm{S}(c_{t-2}+\mathrm{SK}_4)+\mathrm{S}(c_{t-3}+\mathrm{SK}_3)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[38]\\
&+\Big(\mathrm{S}(c_{t-3}+\mathrm{SK}_4)\cdot\mathrm{S}(c_{t-1}+\mathrm{SK}_5)\Big)\cdot\widehat{x}_t[39]\\
&+c_t
\end{aligned}
$$

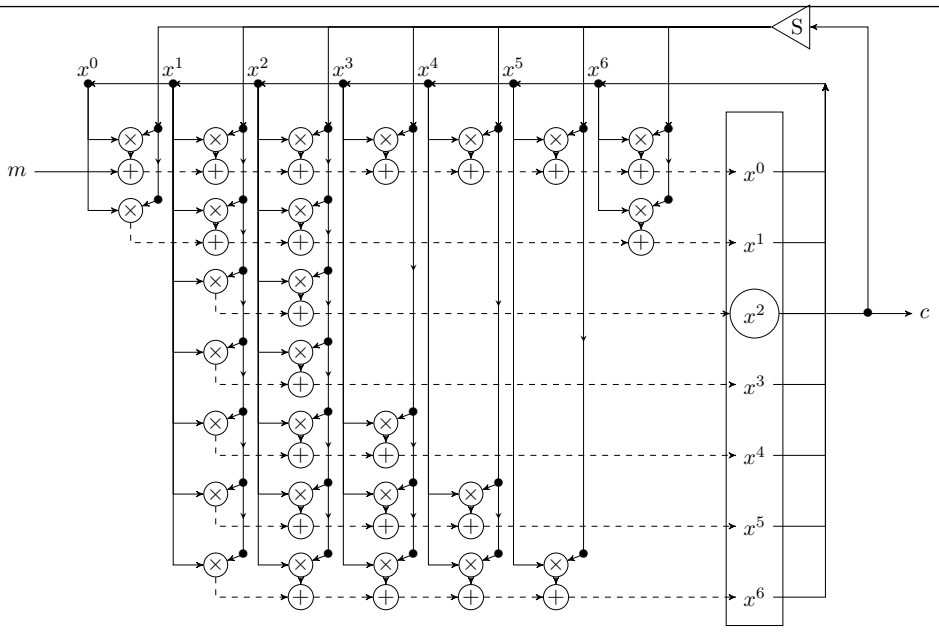


FIGURE A.1 – Schéma descriptif du chiffrement

A.2 Construction du graphe

On se donne un triplet (n, r, n_a) où n est la dimension du système dynamique LPV, r son degré relatif et n_a le nombre d'arcs dans le graphe orienté $\mathcal{G}(\Sigma_\rho)$. La construction du graphe se fait en plusieurs étapes.

Étape 1 : Le graphe $\mathcal{G}(\Sigma_\rho)$ correspond au système Σ_ρ de dimension n qui admet $n+1$ sommets. L'entrée est associée au sommet $\mathbf{v^0}$. Les n autres sommets sont notés $\mathbf{v^1}, \ldots, \mathbf{v^n}$.

Étape 2 : (sommets essentiels) ajout des sommets $\mathbf{v^i}, \ldots, \mathbf{v^{i+1}}$ avec $i = 0, \ldots, r-1$. Le degré relatif étant r, la sortie plate correspond au sommet $\mathbf{v^r}$ et il y a r sommets qui lient $\mathbf{v^0}$ à $\mathbf{v^r}$.

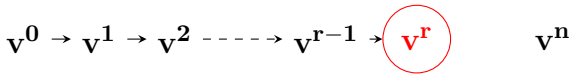


FIGURE A.2 – Graphe obtenu après les étapes 1-2. Le sommet $\mathbf{v^r}$ correspond à la sortie plate.

Après l'étape 2, les conditions **C0-C2** sont remplies et le graphe résulant est décrit dans la Figure A.2. Toutefois, la dynamique du système est triviale. Les étapes suivantes fournissent un moyen d'ajouter les arcs $(\mathbf{v^i}, \mathbf{v^j})$ qui garantissent que les Conditions **C0-C2** sont remplies.

Étape 3 : (sommets non-essentiels) ajout des arcs $(\mathbf{v^i}, \mathbf{v^{i+1}})$ pour $i = r, \ldots, n-1$ pour éviter des sommets $\mathbf{v^j}$, $j = r+1, \ldots, n$ sans prédecesseur. Autrement, la dynamique correspondant au sommet $\mathbf{v^j}$ réduirait la valeur de x^j_{k+1} à 0.

Étape 4 : (sommets essentiels) Considérons les sommets $\mathbf{v^i}$ $i = 1, \ldots, r-2$, comme sommets de début. Pour chaque sommet $\mathbf{v^i}$, $i = 1, \ldots, r-2$, ajouter les arcs $(\mathbf{v^i}, \mathbf{v^j})$ avec $j = 1, \ldots, i$. Ces sommets introduisent

des cycles, y compris des cycles d'ordre 1. Néanmoins ces cycles satisfont la Condition **C2** puisque les sommets $\mathbf{v^i}$, $i = 0, \ldots, r-1$ appartiennent à $V_{ess}(\mathbf{U}, \{\mathbf{v^r}\})$.

Etape 5 : (sortie plate et sommet après la sortie plate) on considère les sommets $\mathbf{v^i}$, $i = r-1, r$. On ajoute des arcs qui relient le sommet $\mathbf{v^i}$ aux sommets $\mathbf{v^j}$: $(\mathbf{v^i}, \mathbf{v^j})$, $j = 1, \ldots, i, i+2, \ldots, n$.

Le graphe obtenu après les étapes 1-5 est illustré par la Figure A.3.

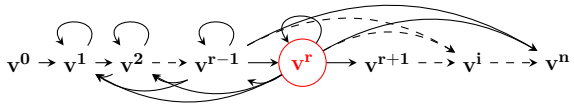


FIGURE A.3 – Graphe obtenu après les étapes 1-5.

Etape 6 :(sommets non-essentiels) on considère les sommets $\mathbf{v_i}$, $i = r+1, \ldots, n$. Pour chaque sommet $\mathbf{v_i}$, $i = r+1, \ldots, n$, on ajoute les arcs $(\mathbf{v_i}, \mathbf{v_j})$ pour $j = 1, \ldots, r-1$ et $j = i+2, \ldots, n$.

Le graphe final obtenu après réalisation des Etapes 1-6 est illustré par la Figure A.4.

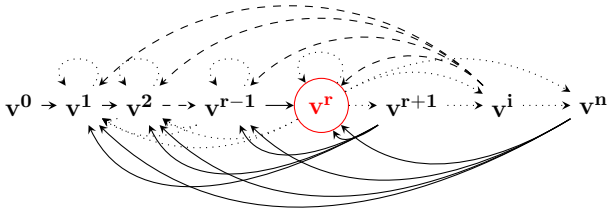


FIGURE A.4 – Graphe obtenu après réalisation des étapes 1-6.

A.3 Opérations dans GF(16)

Toutes les variables du schéma (entrée m_i, état interne x_i, cryptogramme c_i) sont considérées comme des éléments sur le corps fini GF(16) et sont représentés sur 4 bits.

Le corps fini GF(16) est défini comme extension du corps $\mathbb{F}_2 = \mathrm{GF}(2)$ à l'aide du polynôme irréductible $x^4 + x + 1$. L'addition et la multiplication sont donc définies comme addition et multiplication sur des polynômes à coefficients dans $\mathbb{F}_2$ modulo $x^4 + x + 1$.

A.3.1 Addition

Pour $a \in \mathrm{GF}(16)$, on peut représenter a sous forme polynômiale, $a = a_3x^3 + a_2x^2 + a_1x + a_0$, $a_i \in \{0, 1\}$, et on notera tout simplement $a = (a_3, a_2, a_1, a_0)$. L'addition est définie comme addition bit-à-bit modulo 2 : $a \oplus b = a$ XOR b.

A.3.2 Multiplication

Le produit de a par un élément b est entièrement déterminé une fois qu'on connait les produits de a par les monômes $x^i,\ i = 1, \ldots, 3$.

Produit $a \bullet x$:

- Si $a_3 = 0$, le produit de a par x donne $a \bullet x = (a_2, a_1, a_0, 0)$ que l'on peut voir comme un simple décalage à gauche de la valeur initiale de a : $a \bullet x = a << 1$.
- Si $a_3 = 1$ alors $a \bullet x = a_3x^4 + a_2x^3 + a_1x^2 + a_0x \bmod (x^4 + x + 1) = a_2x^3 + a_1x^2 + (a_0 \oplus 1)x + 1$. Ainsi en notation binaire $a \bullet x = (a << 1)$ XOR 0x03.

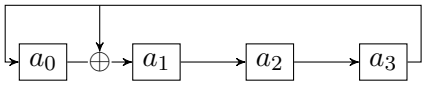


FIGURE A.5 – Produit $a \bullet x$

Produit $a \bullet b$:

On a donc

$$a \bullet b = a \bullet b_3x^3 \oplus a \bullet b_2x^2 \oplus a \bullet b_1x \oplus a \bullet b_0 = b_3 \cdot a \bullet x^3 \oplus b_2 \cdot a \bullet x^2 \oplus b_1 \cdot a \bullet x \oplus a \cdot b_0$$

avec $a \bullet x^3 = (a \bullet x^2) \bullet x = ((a \bullet x) \bullet x) \bullet x$.

Input: a
Output: $a \bullet x$
  if $a_3 == 0$ **then**
    return $a << 1$
  end if
  return $(a << 1)$ XOR 0x03

Input: a, b
Output: $r := a \bullet b$
  $r := a \times b_0$
  for i=1 ;i< 4 ;i++ **do**
    $a := a \bullet x$
    if $b_i \neq 0$ **then**
      $r := r$ XOR a
    end if
  end for
  return r

On a en notation hexadécimale $x = 0x2,\ x^2 = 0x4, ; x^3 = 0x8$.

En exemple, pour $a = 0x7 = 0111,\ b = 0xd = 1101$, on a $a \bullet b = 0x7 \bullet (0x8 \oplus 0x4 \oplus 0x1)$

$$\begin{aligned} 0x7 \bullet 0x1 &= 0x7 \\ 0x7 \bullet 0x2 &= 0xe \\ 0x7 \bullet 0x4 = 0xe \bullet 0x2 &= 0xf \\ 0x7 \bullet 0x8 = 0xf \bullet 0x2 &= 0xd \end{aligned}$$

Au final, $a \bullet b = 0x7 \bullet 0xd = 0xd \oplus 0xf \oplus 0x7 = 0x5$.