N°d'ordre: 3814

50376
2006
86

THÈSE DE DOCTORAT DE L'UNIVERSITÉ DE LILLE 1

Spécialité :

Informatique

présentée par

VINCENT BÉNONY

pour obtenir le grade de DOCTEUR de L'UNIVERSITÉ de LILLE 1

Sujet de la thèse :

ÉTUDE ET CONCEPTION DE SYSTÈMES DE CHIFFREMENT À FLOT DANS LE CONTEXTE D'ARCHITECTURES MATÉRIELLES FORTEMENT CONTRAINTES

Soutenue le 29 juin 2006, devant le jury composé de :

M.	Jean-Paul	DELAHAYE	Président
Mlle.	Caroline	FONTAINE	Directeurs
M.	Jean-Marc	GEIB	
MM.	Thierry	BERGER	Rapporteurs
	Nicolas	SENDRIER	
MM.	Philippe	GUILLOT	Examinateurs
	Éric	WEGRZYNOWSKI	

Remerciements

Après trois années et quelques poussières à penser que la fin n'était pas pour tout de suite, me voici pourtant arrivé au terme de ma thèse : j'avoue ne pas l'avoir vu venir ! Tout le travail accompli n'aurait pu être mené à bien sans l'aide de mes proches.

Je souhaite remercier toutes les personnes qui m'ont permis d'y arriver, à commencer par Éric Wegrzynowski, qui est véritablement celui qui aura le plus influencé mon parcours universitaire. C'est d'ailleurs suite à un stage de maîtrise en sa compagnie et celle de Caroline, concernant la construction de fonctions booléennes, que j'ai eu l'envie de poursuivre en thèse. Je pense que je ne le remercierai jamais assez pour tout le temps qu'il m'a consacré.

Je remercie aussi vivement Caroline Fontaine, co-directrice de ma thèse, pour m'avoir permis de la mener à bien. C'est elle qui aura tout mis en œuvre pour que je puisse avoir ma chance. Je la remercie pour m'avoir toujours soutenu en toutes circonstances, pour sa patience, pour ses conseils éclairés, et pour son aide précieuse dans toutes mes démarches.

Merci à François Recher, qui m'a aussi beaucoup aidé. Sa maîtrise des outils mathématiques m'a bien souvent été d'un grand secours. Merci beaucoup pour les très nombreuses relectures auxquelles il a participé.

Merci à Jean-Marc Geib d'avoir accepté d'être le directeur de ma thèse, ainsi qu'à Thierry Berger et Nicolas Sendrier pour avoir accepté d'en être les rapporteurs. Je leur en suis très reconnaissant.

Merci à Philippe Guillot et à Éric Filiol ainsi que toutes les personnes du projet CODES de l'INRIA pour leur soutien, leurs encouragements, et pour l'intérêt qu'ils ont porté à mes travaux.

Je remercie Jean-Paul Delahaye d'avoir accepté d'être président de mon jury. C'est un honneur pour moi d'avoir pu compter sur sa présence.

Merci aussi aux membres de l'équipe POPS du LIFL pour m'avoir acceuilli dans le cadre de ma thèse, et aujourd'hui pour la confiance qu'ils me témoignent dans le cadre d'autres projets.

Un très grand merci à Sandrine pour avoir réussi à me supporter pendant toute la phase de rédaction puis pendant celle de mise au point de mes transparents. Je pense que je devais être particulièrement difficile à vivre, et pourtant, elle est toujours restée très compréhensive à mon égard.

Merci à mes amis, Amélia, Anne, Antoine, Cyril, Damien, François, Julien, Johann, Nicolas, Pierre et Stéphanie, et à tous ceux que j'aurais oublié, pour avoir toujours été là pour me changer les idées, et pour tous les bons moments que nous avons passés ensemble.

Un immense merci à mes parents, mes sœurs Gaëlle, Isabelle et Pascale, à mon frère Pierre-Yves. Je les remercie pour leur soutien inconditionnel, et pour les liens uniques et irremplaçables qui nous unis. Merci surtout à Gaëlle, qui m'aura toujours prodigué de très bon conseils, et qui m'aura ainsi permis de tenir bon pendant ces trois années. Merci aussi à Dominique, qui a toujours été d'un très grand soutien.

Je réserve tout de même une place particulière à ma maman, sans qui je ne serais jamais arrivé jusque là. C'est parce qu'elle a toujours cru en moi que j'ai fini par y croire moi aussi. Son courage face aux épreuves restera pour moi l'exemple à suivre. Merci d'avoir toujours été si patiente, si attentive et si prévenante. J'espère qu'elle restera toujours fière de moi.

Introduction

Depuis quelques années, on observe un besoin grandissant de *mobilité* dans le domaine de l'informatique. Aujourd'hui, de nombreux professionnels (mais aussi un très grand nombre de particuliers) utilisent dans leurs déplacements des *assistants personnels*, ou *PDA*. Ces assistants ont beaucoup évolué depuis leur apparition au début des années 1980, où ils n'étaient alors que des *agendas électroniques* relevant plus du gadget. On demande à ces assistants de pouvoir remplir des tâches aussi simples que la prise de notes, l'enregistrement de rendez-vous, mais aussi des tâches plus complexes comme la lecture de fichiers multimedia, ou encore la possibilité de se connecter à des *réseaux sans fil.*

Ce dernier point a conduit à de nombreux travaux dans le domaine des *réseaux sans fil spontannés sans infrastructure fixe*, aussi appelés réseaux *AdHoc* : il s'agit d'un modèle *coopératif* où des réseaux ont la capacité de se former entre de *Petits Objets Portables et Sécurisés* [61] (ou *POPS*), définissant un graphe de nœuds interconnectés. Les communications entre deux mobiles prennent alors la forme de paquets de données relayés au travers du graphe par les autres mobiles du réseau, par *sauts successifs* (*hops*).

Malheureusement, cette technologie soulève un grand nombre de problèmes. Le premier d'entre eux réside dans le fait que tout repose sur une confiance aveugle envers les autres participants. Que se passe-t-il alors si une personne mal intentionnée entre dans le réseau, et décide de ne plus transmettre correctement les paquets de données, voir même de modifier des paquets qui transitent par lui ? Ce type de comportement permet très simplement de mener des attaques de type *dénis de service*, amenant à l'effondrement total du réseau.

De plus, les échanges étant relayés par un grand nombre de mobiles, il devient beaucoup plus facile d'espionner les conversations en faisant tout simplement en sorte de se trouver sur le chemin des informations. Les solutions habituelles à ce type de problèmes ne peuvent s'appliquer ici, car la plupart d'entre elles s'appuient sur une architecture centralisée, incompatible avec les réseaux Ad-hoc. De nombreux travaux de sécurisation ont été initiés, et nous pouvons citer dans ce domaine, les travaux des projets *SERAC* [68], *SPLASH !*[73], *CESAM*[12] ou encore *KAA*[43].

Il est donc nécessaire de faire appel aux méthodes cryptographiques afin d'assurer l'*authentification* des intervenants, ainsi que la *sécurité des transactions.* Un des problèmes est la conception de *systèmes de chiffrement* s'adaptant aux contraintes techniques, généralement fortes, du monde de l'embarqué, car malgré la montée en puissance de ces appareils, ils restent néanmoins bien loin des ordinateurs de bureau classiques :

le processeur est beaucoup plus limité en terme de vitesse, leurs ressources énergétiques sont limitées, mais surtout, ils sont généralement équipés de très peu de mémoire.

On distingue deux grandes familles de systèmes de chiffrement : les systèmes de chiffrement à *clé publique*, et les systèmes de chiffrement *à clé secrète*. Cette dernière est elle-même divisée entre les *systèmes de chiffrement à flot* et les *systèmes de chiffrement par bloc*, les premiers visant à chiffrer les bits d'un message les uns après les autres, alors que la deuxième famille applique des permutations secrètes sur des blocs complets de bits. De part leur nature, les systèmes de chiffrement à flot sont plus souvent utilisés pour les communications réseau : ils sont en général beaucoup plus rapides, mais surtout, beaucoup plus tolérants aux erreurs de transmission. En effet, si une erreur survient sur l'un des caractères lors de la transmition d'un message chiffré à flot, seul ce caractère sera erroné lors du déchiffrement, alors que toute la partie suivant le caractère érroné sera perdue dans le cas d'une chiffrement par bloc.

La mise au point d'attaques très puissantes, comme les *attaques algébriques*, a fortement ébranlé le monde du chiffrement. Ces attaques, très génériques, tentent de représenter les systèmes de chiffrement par des systèmes d'équations plus ou moins faciles à résoudre. Loin de démotiver les *cryptographes*, ces nouvelles attaques n'ont fait qu'augmenter le besoin de nouveauté dans le domaine des chiffrement à flot, et l'on a vu apparaître des projets comme *eSTREAM*[22] dont le but est de stimuler la recherche dans ce domaine, en organisant un appel à contributions afin d'obtenir de nouveaux systèmes susceptibles de mieux résister à ces attaques. On compte aujourd'hui dans le cadre de ce projet, trente-cinq nouvelles propositions toujours en cours d'évaluation.

C'est dans cette effervescence que se place le travail de cette thèse, dans laquelle je présenterai l'étude que j'ai menée concernant les systèmes de chiffrement à flot. Je présenterai plusieurs algorithmes classiques de chiffrement, ainsi que la plupart des attaques connues sur cette famille de systèmes de chiffrement, mais aussi l'attaque que j'ai mise au point concernant les fonctions triangulaires ou *T-Fonctions*. Je proposerai ensuite un système de chiffrement utilisant une primitive particulièrement efficace pour construire des *suites pseudo-aléatoires* de bonne qualité : les *registres à décalage à rétroaction non linéaire*. Cette primitive n'a été que très peu utilisée [33, 13], très vraissemblablement à cause des difficultés qui existent pour exprimer les fonctions sous-jacentes à son fonctionnement de manière concise et efficace. Je proposerai en annexe une implémentation logicielle de ce système de chiffrement.

Je parlerai enfin d'un sujet d'étude qui m'a beaucoup intéressé : l'*introduction de biais* dans les générateurs pseudo-aléatoires des systèmes cryptographiques, et plus particulièrement, du risque qu'apporte l'utilisation de bibliothèques logicielles *toutes prêtes* d'outils cryptographiques.

Chapitre 1

Méthodes classiques pour la construction de systèmes de chiffrement à flot

1.1 Introduction

Ce chapitre présente la famille des systèmes de *chiffrement à flot* (ou *stream cipher*) ainsi que quelques exemples de tels systèmes.

Avant toute chose, je présenterai les primitives que l'on retrouve le plus souvent au cœur de ces systèmes, à savoir les *registres à décalage*, et plus particulièrement les *registres à décalage à rétroaction linéaire*, mais aussi une primitive beaucoup plus récente, appelée *T-fonction.*

1.2 Présentation générale du chiffrement à flot

Le *chiffrement à flot* (ou *stream cipher*) désigne une famille de cryptosystèmes dont le principe est de chiffrer les données caractère par caratère (généralement bit par bit) en utilisant une transformation qui varie dans le temps. Dans cette famille, on peut citer entre autres des systèmes comme RC4 [62], A5/1 [60] ou encore LILI128 [71].

Le principe du chiffrement à flot est très simple (figure 1.1) : il s'agit d'ajouter à

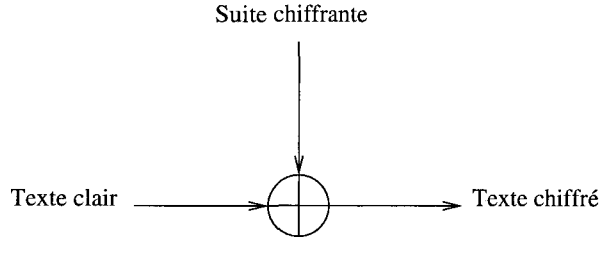


FIG. 1.1 – Principe du chiffrement à flot

un texte clair, composé de symboles appartenant à un alphabet $\mathcal{A}$, une suite chiffrante de symboles du même alphabet, afin d'obtenir la suite chiffrée (ou *cryptogramme*). Ce système est particulièrement efficace en terme de vitesse.

Parmi tous les systèmes de chiffrement à flot, l'un d'entre eux mérite un intérêt tout particulier : le *masque jetable* ou *One Time Pad* [36]. Le principe est d'utiliser une suite chiffrante de même longueur que le texte à chiffrer, composée de symboles tirés de manières *réellement aléatoire*, et de n'utiliser cette suite *qu'une seule fois*. Le système ainsi obtenu à été montré comme étant inconditionnellement sûr [69].

Malheureusement, les contraintes d'utilisation du masque jetable, qui lui permettent d'être un système sûr, empêchent une utilisation aisée de ce cryptosystème : l'utilisation d'une clé de chiffrement aussi longue que le texte à chiffrer est très contraignante. Ainsi, les cryptosystèmes de la famille du chiffrement par flot sont en général des approximations du masque jetable : la suite chiffrante aléatoire est remplacée par une suite chiffrante pseudo-aléatoire, dérivée d'une quantité secrète appellée *clé de chiffrement*, cette clé était de taille beaucoup plus petite que le texte.

La sécurité du One Time Pad repose aussi sur le fait qu'une même suite chiffrante ne doit pas être utilisée à plusieurs reprise. Ce critère est lui aussi très contraignant car l'échange de clé secrète peut se révéler être un problème complexe. Pour palier ce problème, on fait généralement appel à un *vecteur d'initialisation*. Il s'agit d'une *quantité aléatoire publique*, qui servira, avec la clé secrète, de paramètre à une *fonction d'initialisation* afin d'obtenir l'initialisation du générateur pseudo-aléatoire. De manière à ce que le destinataire du message puisse retrouver l'initialisation du générateur, le vecteur est envoyé en clair avec le cryptogramme.

C'est pourquoi il est nécessaire de disposer de bonnes méthodes de construction de suites pseudo-aléatoires, ayant de bonne propriétés statistiques et cryptographiques (non prédictibilité). Nous allons maintenant aborder différentes méthodes possibles de construction de suites pseudo-aléatoires.

1.3 Primitives utilisées pour le chiffrement à flot

1.3.1 Générateurs de pseudo-aléa

Les suites issues de générateurs pseudo-aléatoires sont en réalité des suites construites via un processus *entièrement déterministe*. Il est toujours possible de représenter un générateur pseudo-aléatoire comme un *automate à états finis*, composé d'un *état interne* $\mathrm{x} \in \mathcal{Q}$, d'une *fonction de transition* $f : \mathcal{Q} \mapsto \mathcal{Q}$, d'une *fonction de sortie* $F : \mathcal{Q} \mapsto \mathcal{A}^n$ (où $\mathcal{A}$ représente un alphabet quelconque) et d'une *horloge* servant à cadencer le processus de construction des suites pseudo-aléatoires (figure 1.2).

La construction s'opère alors ainsi : à chaque *top d'horloge*, le mot $F(\mathrm{x})$ est calculé à partir de l'état interne du générateur. Ce mot vient s'ajouter à la suite des symboles pseudo-aléatoires générés. L'état interne du générateur est ensuite mis à jour et remplacé par la valeur de $f(\mathrm{x})$.

De part le fait que la taille de l'état interne du générateur est finie et que le processus

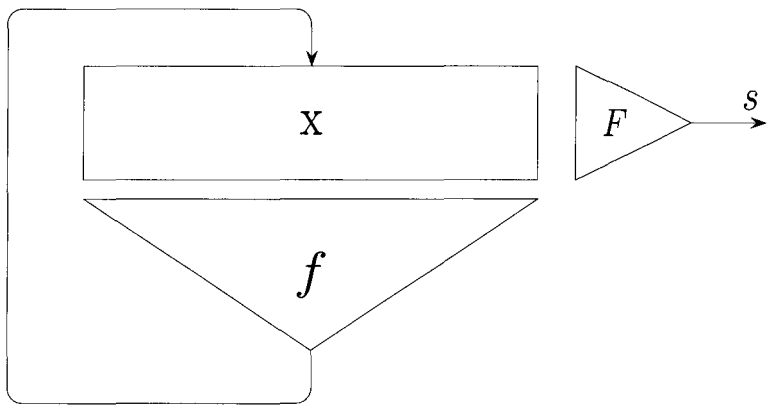


FIG. 1.2 – Principe d'un générateur pseudo-aléatoire

est déterministe, la suite s construite est *ultimement périodique*, et sa période est au plus $\text{card}(\mathcal{Q})$. On dit d'une suite $(s_n)_n$ qu'elle est *générée* (ou qu'elle est *engendrée*) par un générateur pseudo-aléatoire particulier si et seulement si ce générateur est capable, pour un état initial particulier, de construire une suite en tout point identique à $(s_n)_n$.

Il s'agit ici d'une description très générale d'un générateur de pseudo-aléa. Dans la suite du document, nous nous placerons dans le cas où $\mathcal{A} = \{0, 1\}$ (que l'on pourra noter aussi $\mathbb{F}_2$), et $\mathcal{Q} = \mathcal{A}^l$: il s'agira donc de considérer l'état interne du générateur, ainsi que sa sortie, comme des mots binaires.

1.3.2 Les registres à décalage

Les *registres à décalage*, ou *FSR* pour *Feedback Shift Register*, sont une instance particulière de l'automate décrit précédemment. Le grand intérêt de ce type d'objet réside dans son extrême simplicité. Si l'on reprend les notations précédentes, la fonction de transition et la fonction de sortie sont définies par :

$$\begin{aligned} f(x_{n-1}, x_{n-2}, \ldots, x_0) &= (r(x_{n-1}, x_{n-2}, \ldots, x_1, x_0), x_{n-1}, x_{n-2}, \ldots, x_1) \\ F(x_{n-1}, x_{n-2}, \ldots, x_0) &= x_0. \end{aligned}$$

où r est appellé *fonction de rétroaction*.

On s'apperçoit ici que le nom de *registre à décalage* provient du fait que l'état interne du générateur, une fois actualisé, est alors un *décalage vers la droite* de l'état précédent, où le caractère le plus à gauche est issu d'un calcul sur l'état interne précédent.

On représente généralement un registre à décalage comme sur la figure 1.3. Dans cet exemple, nous sommes face à un registre de taille 5. Selon les contextes, nous noterons indifféremment $r(\mathrm{x})$, ou $r(x_4, x_3, x_2, x_1, x_0)$, le symbole calculé par la fonction de rétroaction du registre dans l'état $\mathrm{x} = (x_4, x_3, x_2, x_1, x_0)$.

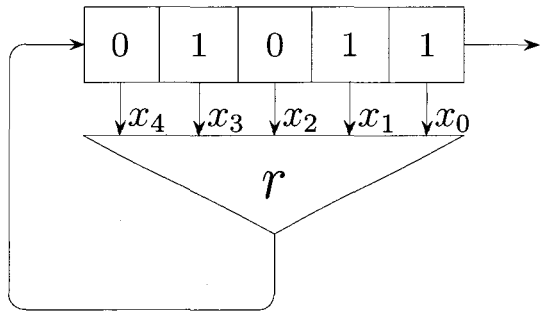


FIG. 1.3 – Représentation d'un registre à décalage utilisant l'alphabet $\{0, 1\}$

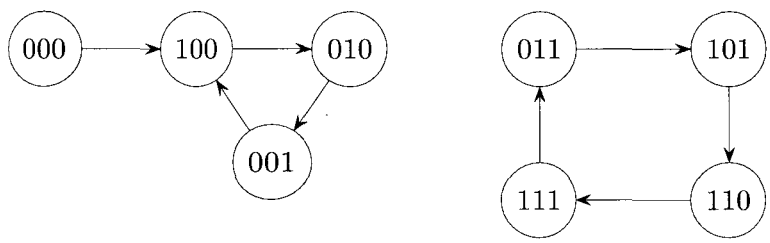


FIG. 1.4 – Graphe de transition d'un registre singulier

Nous nous intéresserons particulièrement aux registres travaillant sur l'alphabet binaire $\{0, 1\}$: dans ce contexte, un vecteur x de n bits pourra indifféremment être vu comme le vecteur $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0) \in \mathbb{F}_2^n$ ou bien comme l'entier modulo 2^n défini par $\sum_{i=0}^{n-1} 2^i x_i$.

1.3.3 Registre non singulier

Puisqu'il s'agit d'un automate à états finis, le registre à décalage peut être représenté sous la forme d'un graphe de transition : les nœuds du graphe représentent alors tous les états internes possibles du registre à décalage, et il existe une flèche allant d'un état x vers un état y si et seulement si l'état y est le successeur de l'état x pour le registre à décalage au top d'horloge suivant, soit en d'autres termes, si $f(\mathrm{x}) = \mathrm{y}$ (figure 1.4). À noter qu'il ne s'agit donc pas nécessairement d'un graphe connexe.

Lorsque l'on cherche à concevoir un registre à décalage, on souhaite souvent faire en sorte que son graphe de transition ne comporte que des cycles sans pré-période. Si l'on regarde la figure 1.4, on s'aperçoit que le graphe décrit deux cycles, dont un avec une pré-période : en effet, si l'on part de l'état 000, le successeur sera l'état 100, puis l'état 010, l'état 001 pour enfin revenir à l'état 100. Il ne sera donc plus possible de revenir à l'état 000. On dit alors que ce registre est *singulier*.

Ce type de phénomène apparaît lorsqu'un état comporte zéro ou deux flèches entrantes, c'est-à-dire, quand f n'est pas bijective.

Il existe un moyen simple de contrôler ce phénomène et de s'assurer de la *non singu-*

larité du registre. Considérons l'état x du registre :

$$\mathrm{x} = (x_{n-1}, \ldots, x_1, x_0)$$

De part la nature des registres à décalage, nous savons que les états précédents possibles sont :

$$(x_{n-1}, \quad \ldots \quad, x_1, 0),$$
$$(x_{n-1}, \quad \ldots \quad, x_1, 1).$$

Il nous faut donc faire en sorte que ces deux états n'aient pas le même successeur, et que par conséquent :

$$r((x_{n-1}, \ldots, x_1, 0)) \neq r((x_{n-1}, \ldots, x_1, 1)),$$

ce qui implique

$$r((x_{n-1}, \ldots, x_1, 0)) = r((x_{n-1}, \ldots, x_1, 1)) \oplus 1.$$

Nous pouvons résumer ceci grâce au théorème suivant :

Théorème 1 *[35] Soit un registre à décalage de taille n et sa fonction de rétroaction r associée. Ce registre est non-singulier si et seulement si r peut se mettre sous la forme :*

$$r(x_{n-1}, x_{n-2}, \ldots, x_0) = x_0 \oplus g(x_{n-1}, x_{n-2}, \ldots, x_1)$$

où g est une fonction quelconque sur les mots de longueur $n-1$.

1.3.4 Modification de la table de vérité de la fonction de rétroaction

Il est intéressant de se pencher sur ce qui se passe lorsque l'on modifie un bit de la table de vérité de la fonction de rétroaction d'un registre à décalage. Étant donné que seuls les registres sans pré-période nous intéressent, nous ne nous occuperons que des effets d'une modification de la table de vérité de la fonction g pour une fonction de rétroaction qui peut s'écrire sous la forme imposée par le théorème 1.

Modifier un bit de la table de vérité de g revient à modifier le successeur de deux états du graphe de transition du registre à décalage. En effet, modifier la valeur du vecteur $(x_{n-2}, x_{n-3}, \ldots, x_0)$ de g modifie la valeur des vecteurs $\mathrm{x}_0 = (x_{n-2}, x_{n-3}, \ldots, x_0, 0)$ et $\mathrm{x}_1 = (x_{n-2}, x_{n-3}, \ldots, x_0, 1)$ pris en entrée par f. Deux cas de figure peuvent alors se présenter :

- *les deux états sont sur le même cycle* : la modification entraîne que le nouveau successeur de l'état x_0 devient l'ancien successeur de l'état x_1, et réciproquement ; ceci à pour effet de *découper le cycle en deux cycles distincts* ;
- *les deux états sont sur des cycles différents* : on s'aperçoit alors qu'il apparaît le phénomène inverse au point exposé précédemment, et l'on obtient alors *la fusion des deux cycles.*

Nous pouvons donc résumer :

Théorème 2 *Modifier un bit de la table de vérité de g est équivalent à retirer ou ajouter un cycle dans le graphe de transition de la fonction f.*

1.3.5 Registres à décalage à rétroaction linéaire

Parmi les registres à décalage, il existe une sous-famille de registres particuliers : les *registres à décalage à rétroaction linéaire* ou *LFSR* (Linear Feedback Shift Register). Ces registres ont été beaucoup étudiés, et nous connaissons bien leurs caractéristiques. Comme leur nom l'indique, il s'agit ici de considérer uniquement des fonctions *linéaires* comme fonction de rétroaction. Ainsi, leur implémentation logicielle, ou matérielle, se trouve considérablement simplifiée. De plus, cette linéarité apporte une vitesse de calcul très intéressante.

La fonction de rétroaction d'un tel registre est de la forme (on travaille dans $\mathbb{F}_2$) :

$$r(x_{n-1}, \ldots, x_1, x_0) = \sum_{i=0}^{n-1} \alpha_i x_i.$$

Puisque nous nous intéressons particulièrement au cas binaire, les α_i valent ici 0 ou 1. La fonction de sortie est quant à elle la fonction

$$F(x_{n-1}, \ldots, x_1, x_0) = x_0.$$

La suite s générée par un registre à décalage à rétroaction linéaire est facile à décrire, puisqu'il s'agit d'une *suite à récurrence linéaire*. En effet, en posant $a_i = \alpha_{n-i}$, tous les termes de la suite de rang t peuvent s'écrire sous la forme :

$$s_t = \sum_{i=1}^{n} a_i s_{t-i}. \tag{1.1}$$

On associe souvent à cette suite périodique infinie, la série génératrice :

$$G(X) = \sum_{i=0}^{\infty} s_i X^i$$

En utilisant alors l'équation 1.1, la série G peut être réécrite sous la forme :

$$\begin{aligned}
G(X) &= \sum_{i=0}^{n-1} s_i X^i + \sum_{i=n}^{\infty} \sum_{j=1}^{n} a_j s_{i-j} X^i \\
&= \sum_{i=0}^{n-1} s_i X^i + \sum_{j=1}^{n} a_j X^j \sum_{i=n}^{\infty} s_{i-j} X^{i-j} \\
&= \sum_{i=0}^{n-1} s_i X^i + \sum_{j=1}^{n} a_j X^j \sum_{i=0}^{\infty} s_i X^i \\
&= \sum_{i=0}^{n-1} s_i X^i + G(X) \sum_{j=1}^{n} a_j X^j.
\end{aligned}$$

Ce qui nous amène à :

$$G(X) = \frac{\sum_{i=0}^{n-1} s_i X^i}{1 + \sum_{j=1}^{n} a_j X^j} = \frac{P(X)}{Q(X)}.$$

La série génératrice peut donc être vue comme le rapport d'un polynôme P de degré au plus $n-1$ par un polynôme Q de degré n. La suite est entièrement déterminée par les coefficients a_i ainsi que par les premières valeurs de $(s_n)_{n\geq 0}$. Le polynôme Q est généralement appelé *polynôme de rétroaction* de la suite, et le polynôme P, *polynôme d'initialisation*.

Ce résultat est très important : en effet, il signifie qu'il existe une correspondance entre les suites engrendrées par un LFSR de taille n et de fonction de rétroaction r, et les fractions polynomiales $P(X)/Q(X)$ avec Q le polynôme caractéristique associé à r, et P un polynôme de degré inférieur à n.

La première conséquence évidente issue de cette égalité est le fait qu'une suite engendrée par un LFSR de taille n peut aussi être engrendrée par un autre : il suffit de considérer par exemple un polynôme quelconque $S(X)$ non nul. Le rapport $\frac{P(X)S(X)}{Q(X)S(X)}$ est alors égal à $\frac{P(X)}{Q(X)}$. Le développement de la série étant identique, la suite le sera elle aussi, ce qui signifie qu'il existe un LFSR de taille $n + \deg(S)$ qui engendre la même suite que le LFSR de départ. À l'inverse, s'il existe un polynôme $T(x)$ qui divise $P(X)$ et $Q(X)$, alors cela signifie qu'il existe un LFSR de taille $n - \deg(T)$ qui engendre la même suite.

Cette représentation est très intéressante dans l'étude de la période d'un LFSR. Étant donné que le registre est de taille n, un maximum de 2^n états différents peuvent être atteints lors de la génération de la suite, ce qui implique une période inférieure ou égale à 2^n. Or, si l'on observe l'état où toutes les cases du registre sont égales à 0, de part la linéarité de la rétroaction, le registre ne pourra pas sortir de cet état. Ceci nous donne une borne supérieure théorique de la période d'une suite générée par un LFSR, soit $2^n - 1$.

Imaginons que la suite soit de période p : alors par définition $\forall i, s_i = s_{i+p}$, ce qui nous permet de réécrire $G(X)$ sous la forme :

$$\begin{aligned} G(X) &= \sum_{i=0}^{\infty} a_i X^i \\ &= (a_0 + a_1 X + \ldots a_{p-1} X^{p-1}) + X^p (a_0 + a_1 X + \ldots a_{p-1} X^{p-1}) \\ &\quad X^{2p} (a_0 + a_1 X + \ldots a_{p-1} X^{p-1}) + \ldots \\ &= (a_0 + a_1 X + \ldots a_{p-1} X^{p-1}) \sum_{i=0}^{\infty} X^{ip} \end{aligned}$$

En remarquant que $\frac{1}{1+X^p} = 1 + X^p + X^{2p} + \ldots = \sum_{i=0}^{\infty} X^{ip}$, nous pouvons réécrire la dernière égalité sous la forme

$$\frac{P(X)}{Q(X)} = \frac{a_0 + a_1 X + \ldots a_{p-1} X^{p-1}}{(1 + X^p)}$$

et donc

$$P(X)(1+X^p) = Q(X)(a_0 + a_1X + \dots a_{p-1}X^{p-1}). \tag{1.2}$$

La période dépend de l'état initial. Si l'on cherche à atteindre une période maximale, nous pouvons simplifier cette équation. En effet, quelque soit l'état initial que nous considérons alors, mis à part l'état $s_0 = s_1 = \dots = s_{n-1} = 0$, nous devons obtenir une période de $2^n - 1$ (tous les états font partie du cycle). Considérons alors le cas où $s_0 = 1$, $s_1 = s_2 = \dots s_{n-1} = 0$. Dans ce cas, $P(X) = 1$, et l'expression 1.2 s'écrit :

$$(1+X^{2^n-1}) = Q(X)(a_0 + a_1X + \dots a_{2^n-2}X^{2^n-2})$$

ce qui signifie en d'autre termes que $Q(X)$ divise $1 + X^{2^n-1}$.

Grâce à tous ces éléments, il est possible de déterminer comment construire la fonction de rétroaction d'un LFSR afin d'obtenir une période maximale.

Imaginons que nous ayons un LFSR de taille n de période maximale, et que pour un état différent de l'état entièrement à zéro, il existe un polynôme $S(X)$ de degré supérieur ou égal à 1 qui divise le polynôme d'initialisation et le polynôme de rétroaction. Alors, cela signifie qu'il existe un registre de taille plus petite engendrant la même suite que ce registre avec cette initialisation. Or, si ce registre est plus petit, il ne peut pas atteindre les $2^n - 1$ états différents imposés par la période de la suite. Ainsi cette contradiction nous indique que pour tout état différent de l'état nul, il n'existe pas de polynôme S qui divise à la fois P et Q, et le seul moyen d'arriver à ceci est de choisir un polynôme de rétroaction *irréductible.*

De plus, ce polynôme doit diviser $1 + X^{2^n-1}$. Un polynôme vérifiant ce critère est appellé *polynôme primitif.*

Les LFSRs permettent d'introduire la notion de *complexité linéaire* :

Définition 1 *La* complexité linéaire λ *d'une suite s est la taille du plus petit LFSR capable d'engendrer cette suite.*

Il s'agit d'une notion très importante dans le cadre de la cryptographie, puisqu'elle détermine la difficulté qu'il existe à reconstruire une suite pseudo-aléatoire à partir de la connaissance d'une partie de cette suite. L'attaque de Berlekamp-Massey [4, 50] (voir chapitre 2) permet par exemple de construire un registre capable de reconstruire l'intégralité d'une suite pseudo-aléatoire à partir uniquement de la connaissance de 2λ bits consécutifs de la suite étudiée.

Il est possible de voir la complexité linéaire sous un autre angle. Soit E l'opérateur de décalage, défini par

$$Es_n = s_{n+1}.$$

Il est évident que toute suite (s_n) de période p satisfait une récurrence linéaire de degré p, que nous pouvons écrire grâce à l'opérateur E sous la forme :

$$(E^p + 1)s_n = 0.$$

Il est possible que la suite vérifie aussi une récurrence linéaire de degré inférieur à p telle que $r(E)s_n = 0$. Soit r le polynôme de plus petit degré vérifiant cette récurrence, alors par définition :

$$\lambda(s) = \deg(r).$$

Nous avons un résultat remarquable dans le cas où la période est une puissance de 2. Grâce aux résultats précédents, nous savons que $f(X)|X^p + 1$. Si $p = 2^k$, alors $X^p + 1 = (X + 1)^p \mod 2$, et donc $r(X)|(X + 1)^p$. Comme le degré de r est $\lambda(s)$, alors :

$$r(X) = (X + 1)^{\lambda(s)},$$

d'où

$$r(E) = (E + 1)^{\lambda(s)}$$

où le symbole 1 représente, dans cette dernière équation, l'opérateur identité $\forall x, 1(x) = x$. Cette dernière équation est équivalente à dire que la complexité linéaire de (s_n) est le plus petit entier $\lambda(s)$ tel que $\forall i, (E+1)^{\lambda(s)}s_i = 0$, ou encore, ce qui est équivalent, le plus petit entier vérifiant :

$$\forall i, (E + 1)^{\lambda(s)-1}s_i = 1.$$

1.3.6 FCSR

Les *Feedback with Carry Shift Registers*, ou *FCSRs* (ou *registre à décalage avec retenue*), introduits dans [44], représentent une alternative très intéressante aux LFSRs : ils permettent de casser la linéarité des suites produites, en faisant appel à une simple opération d'addition avec retenue, ce qui permet de conserver une primitive simple et rapide.

On peut représenter un FCSR comme sur la figure 1.5, où le symbole $\odot$ représente le ET logique, et le symbole $\boxplus$ représente l'addition avec retenue. Cette addition est définie par :

$$\begin{aligned} \forall k > 0, a_k \boxplus b_k &= a_k \oplus b_k \oplus c_{k-1} \\ c_k &= a_k b_k \oplus c_{k-1}(a_k \oplus b_k) \end{aligned}$$

$$\begin{aligned} a_0 \boxplus b_0 &= a_0 \oplus b_0 \\ c_0 &= a_0 b_0. \end{aligned}$$

Ce type de registre permet d'effectuer des divisions suivant les puissances croissantes entre deux entiers 2-*adiques*.

Définition 2 *Un entier* 2-adique *est une serie infinie de la forme* $\sum_{i=0}^{\infty} s_i 2^i$ *où les coefficients* $s_i \in \{0, 1\}$.

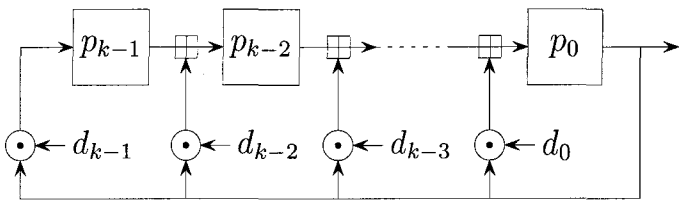


FIG. 1.5 – Représentation d'un FCSR

L'ensemble des entiers 2-adiques est noté $\mathbb{Z}_2$.

On peut voir un entier 2-adique comme une *décomposition binaire infinie* : il est alors possible de concevoir la notion d'addition ou de multiplication d'entiers 2-adiques, en notant que $2^n + 2^n = 2^{n+1}$.

Pour tout entier 2-adique $x = \sum_{i=0}^{\infty} s_i 2^i$, s'il existe un rang N à partir duquel $\forall n \geq N, s_n = 0$, alors on dit que x est *positif*. De la même manière, s'il existe un rang N à partir duquel $\forall n \geq N, s_n = 1$, alors x est *négatif*.

Remarquons que $1 + \sum_{i=0}^{\infty} 2^i = 0$, ce qui implique que $-1 = \sum_{i=0}^{\infty} 2^i$. Ainsi, pour tout entier 2-adique s non nul, il est possible de calculer son complément $-s$: soit j le rang du premier terme non nul de la série entière, c'est-à-dire que j vérifie $\forall i, 0 \leq i < j, s_i = 0$ et $s_j = 1$. Nous obtenons :

$$\begin{aligned} s &= 2^j + \sum_{i=j+1}^{\infty} s_i 2^i \\ -s &= 2^j + \sum_{i=j+1}^{\infty} (1 - s_i) 2^i \end{aligned}$$

De la même manière, il est possible d'exprimer l'inverse q^{-1} d'un entier q impair et négatif, sous la forme :

$$q^{-1} = \sum_{i=0}^{\infty} (1 - q)^i.$$

Les FCSRs nous permettent d'effectuer la division d'un entier p positif par un entier q négatif et impair, c'est-à-dire qu'il existe un entier d positif tel que $q = 1 - 2d$. Il suffit alors d'utiliser les termes de la serie de p pour remplir le registre, et ceux de la série de d pour définir les entrées des ET logiques.

On obtient en sortie du registre le entier 2-adique représentant la division p/q.

Si $p < -q$ et que la taille de p est inférieure à celle de q, alors la suite est périodique, sinon, elle est ultimement périodique.

De la même manière que pour les registres à décalage à rétroaction linéaire, on définit la *complexité 2-adique* Λ_2 d'une suite comme étant la taille du plus petit FCSR capable d'engendrer cette suite. Toujours de la même manière que pour les LFSRs, il est alors possible, grâce à un algorithme proche de celui de Berlekamp-Massey, de reconstruire le

plus petit registre engendrant la totalité de la suite à partir de la connaissance de $2\Lambda_2 + 1$ bits de la suite.

1.4 Les T-fonctions

Les *T-fonctions*, ou *fonctions triangulaires*, ont été introduites par Klimov et Shamir dans [45, 46, 47]. Le principe est de tirer partie de certaines fonctions présentes sur la plupart des microprocesseurs, appellées ici *fonctions primitives* comme les opérations arithmétiques et les opérations booléennes, afin de construire des *fontions inversibles*. Ces fontions sont alors utilisées comme fonction de transition conjointement à un registre de n bits (figure 1.6).

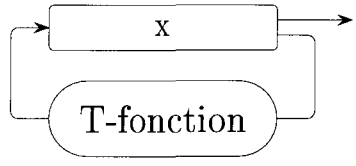


FIG. 1.6 – Automate du générateur

La méthode utilisée pour s'assurer de l'inversibilité de ces fonctions, ainsi que de l'unicité du cycle produit par l'automate est donné au chapitre 3.

1.5 BBS

Le générateur *BBS*, pour *Blum Blum Shub* [7], tient une place particulière parmi les générateurs de pseudo-aléa. En effet, il est le seul à avoir été montré comme étant cryptographiquement sûr, sous l'hypothèse que la factorisation d'un grand nombre soit calculatoirement impossible [42]. On se fixe un paramètre de sécurité τ, puis on choisit deux nombres premiers p et q dont la représentation binaire est d'au moins τ bits, et qui vérifient $p \mod 4 = q \mod 4 = 3$. Notons que les entiers vérifiant ces critères sont appellés *entiers de Blum.*

On calcule ensuite $N = pq$, et un entier $y \in \mathbb{Z}_n^\star$ est choisi comme état initial. On définit alors la suite $(x)_n$ des états internes successifs du générateur par :

$$\begin{aligned} x_0 &= y^2 \mod N \\ x_{i+1} &= x_i^2 \mod N \end{aligned}$$

La suite pseudo-aléatoire produite est alors la suite $(s_n)_{n\geq 0}$ définie par $s_i = x_i \mod 2^l$, avec $l < \log_2(N)$.

Bien que disposant d'un critère de sécurité non négligeable, ce générateur pseudo-aléatoire n'est jamais utilisé pour le chiffrement à flot du fait de la complexité calculatoire de l'algorithme le rendant particulièrement lent. En effet, l'élévation au carré d'un nombre de plusieurs centaines, voir milliers, de bits est très coûteuse, surtout s'il s'agit de ne produire au maximum qu'une dizaine de bits de pseudo-aléa à la fois.

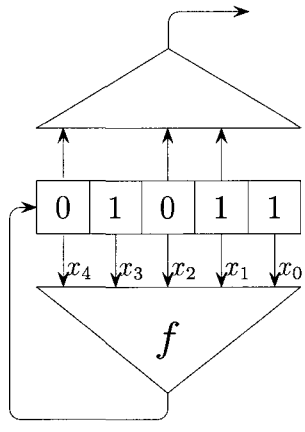


FIG. 1.7 – Exemple de registre filtré

1.6 Quelques exemples de systèmes de chiffrement à flot

Après avoir vu les primitives les plus couramment utilisées, je vais maintenant présenter quelques systèmes de chiffrement à flot dans lesquels elles sont mises en œuvre.

1.6.1 Shrinking Generator

Le *Shrinking Generator* à été introduit en 1993 dans [17]. Il est composé de deux LFSRs indépendants, notés A et S. La construction de la suite produite par ce générateur est la suivante : à chaque top d'horloge, les deux registres sont avancés. On note alors a et s les sorties respectives des registres A et S. Si s vaut 1, alors la sortie du générateur est a, sinon il n'y a pas de sortie pour ce top d'horloge. En résumé, le registre S vient *réduire* la sortie du registre A.

Ce générateur a été beaucoup étudié et sa sécurité à été mise en cause par [53, 78, 79].

1.6.2 Registre filtré

Le registre filtré constitue un des cryptosystèmes les plus simples à mettre en œuvre. Il s'agit ici d'utiliser un LFSR de période maximale unique, et une fonction de filtrage F dont les entrées correspondent à certaines cases du registre (figure 1.7). La suite en sortie de ce générateur correspond alors à la sortie de F. La fonction F doit être choisie avec soin, afin de ne pas révéler trop d'informations sur l'état interne du registre.

De nombreuses attaques ont été publiées contre ce type de registre. Nous pourrons retenir les articles [72, 39].

1.6.3 Registres combinés

Le principe des *registres combinés* est simple et très souvent utilisé. Il s'agit d'utiliser un ensemble de N registres à décalage à rétroaction linéaire, dont les sorties servent de paramètre à une fonction $F'(x_{N-1}, \ldots, x_1, x_0)$ de combinaison.

Le but de ce type de construction est d'améliorer les suites produites par les LFSRs, tout en conservant leur aspect simple et rapide. La principale caractéristique améliorée est la *complexité linéaire* des suites produites : en effet, grâce à l'algorithme de Berlekamp-Massey [4, 50], il est possible de construire un registre produisant la suite étudiée à partir de la connaissance d'uniquement 2λ de ses bits, où λ est la complexité linéaire.

Ainsi, la fonction F' doit être choisie de manière à augmenter la complexité linéaire de la suite produite en sortie. Il est possible de construire F' en maîtrisant la qualité de la suite que l'on obtient.

Par exemple, si l'on considère deux suites s_0 et s_1, de complexité linéaire respective λ_0 et λ_1, alors la suite s produite en effectuant le `OU-EXCLUSIF` entre les bits de s_0 et s_1 est de complexité linéaire $\lambda \leq \lambda_0 + \lambda_1$ [51]. Nous avons égalité dans le cas où les périodes de s_0 et s_1 sont premières entre elles.

De la même manière, la suite p produite en effectuant le `ET` logique entre les bits de s_0 et s_1 est de complexité linéaire $\lambda \leq \lambda_0\lambda_1$, avec égalité dans le cas où λ_0 et λ_1 sont premiers entre eux, et que les polynômes de rétroaction des LFSRs les plus courts engendrant s_0 et s_1 sont tous deux primitifs.

Augmenter la complexité linéaire n'est pas suffisant : comme nous le verrons dans le chapitre 2, il est nécessaire de faire attention à l'*immunité aux corrélations* de la fonction de combinaison.

La construction de F' est donc soumise à un certain nombre d'impératifs, permettant d'assurer la construction d'un système qui ne soit pas soumis aux attaques connues, parmi lesquelles [4, 50, 18, 20, 9, 70, 76].

1.6.4 A5/1

La famille des systèmes de chiffrement appellées A5 est constituée de deux systèmes : A5/1 et A5/2. Il s'agit de systèmes de chiffrement à flot utilisés dans les réseaux téléphoniques *GSM*, A5/1 étant utilisé en Europe, alors que A5/2 est utilisé aux Etats Unis. A5/2 a été cryptanalysé très tôt après sa publication, et est sujet à des attaques permettant, grâce à un matériel peu onéreux, un déchiffrement en temps réel. Il ne sera donc pas exposé ici.

A5/1 repose sur l'utilisation de trois LFSRs de longueurs respectives 19, 22, et 23 bits, dont les polynômes de rétroaction sont :

$$\begin{aligned} p_0(x) &= x^{19} + x^{18} + x^{17} + x^{14} + 1, \\ p_1(x) &= x^{22} + x^{21} + 1, \\ p_2(x) &= x^{23} + x^{22} + x^{21} + x^8 + 1. \end{aligned}$$

Les 64 bits d'état interne sont directement fixés grâce à la valeur d'une clé. Ensuite, 22 bits supplémentaires, constitués d'informations temporelles utilisées par le protocole, sont

introduits dans les registres.

Les sorties des trois générateurs sont alors simplement ajoutés les unes aux autres modulo 2.

La simplicité du système, et le non respect de certaines règles de conception ont aboutis à plusieurs attaques, dont certaines permettent de casser le système de chiffrement en quelques minutes sur une machine actuelle [6, 5].

1.6.5 F-FCSR

Ce générateur a récemment été soumis au projet *eSTREAM* [3, 22]. Il s'agit ici d'utiliser des *registres à décalage à retenue*, et d'utiliser une fonction linéaire de filtrage sur l'état du registre pour construire la suite pseudo-aléatoire. Ce générateur s'appuie sur le fait que la non-linéarité de la fonction de rétroaction du registre permet de s'affranchir d'une fonction de sortie complexe.

Plutôt que d'utiliser le bit de poids faible du registre comme bit sortant à chaque itération, les F-FCSRs utilisent une fonction linéaire pour filtrer l'état interne du registre et ainsi obtenir un bit de sortie. Il s'agit d'effectuer la somme modulo 2 entre au moins trois bits de l'état interne.

Deux variantes de F-FCSR ont été soumises au projet eSTREAM : F-FCSR-8 et F-FCSR-H [3]. Dans les deux cas, les auteurs proposent d'utiliser un ensemble de huit fonctions de filtrage linéaires différentes, afin d'obtenir un octet en sortie à chaque itération.

Dans la spécification de F-FCSR-8, les fonctions de filtrage sont dépendantes de la clé. Il est alors vérifié que ces fonctions sont belles et bien composées d'un minimum de trois monômes. Pour ce qui concerne F-FCSR-H, les 8 fonctions de filtrage sont fixées.

1.6.6 RC4

RC4 est un générateur pseudo-aléatoire proposé par R. Rivest [62]. Son principal intérêt est sa vitesse de génération d'aléa. Il est encore beaucoup utilisé de nos jours dans SSL, ou encore dans le chiffrement WEP du WiFi.

Le système ne repose pas, contrairement aux exemples précédents, sur l'utilisation de LFSRs. Il s'agit ici de créer une table de permutation, entièrement déterminée par la clé de chiffrement. Cette table est ensuite parcourue et modifiée (par inversion d'éléments) dans le même temps pour créer la suite aléatoire.

A ce jour, aucune attaque réelle n'a pu être menée contre RC4 : on pourra juste mentionner la propriété anecdotique de Finney qui établit que quand $i = a, j = a + 1$ et $S[a + 1] = 1$, la période du système est de $N(N - 1)$. Il existe tout de même un biais qui permet de construire un *distingueur*, c'est-à-dire un algorithme capable déterminer entre une suite purement aléatoire et une suite issue du processus de construction de RC4, laquelle est construite grâce au générateur pseudo-aléatoire [25].

L'algorithme est le suivant :

```
Entrée : K la clé de k octets

# Initialisation
    j = 0
    Pour i allant de 0 à 255
        S[i] = i
    Pour i allant de 0 à 255
        j = j + S[i] + K[i mod k]
        échanger S[i] et S[j]

# Génération de l'aléa
    i = j = 0
    tant que nécessaire :
        i = i + 1 mod 256
        j = j + S[i] mod 256
        échanger S[i] et S[j]
        sortir S[S[i] + S[j] mod 256] comme prochain octet
```

Il faut d'ailleurs noter que bien que le système de chiffrement WiFi ait subi des attaques, elles ne portent en aucun cas sur RC4 lui-même, mais plutôt sur le choix de la taille du vecteur d'initialisation, permettant d'obtenir très rapidement des collisions [24, 38].

1.6.7 E0

Le système de chiffrement *E0* a été mis au point dans le cadre du projet de communication sans fil *Bluetooth*. Il s'agit ici de combiner quatre registres à décalage à rétroaction linéaire grâce à un automate à états finis, ou *fonction de combinaison à mémoire*. Les quatre registres contiennent respectivement 25, 31, 33 et 39 bits. Leurs polynômes de rétroaction sont :

$$\begin{aligned} p_0(x) &= x^{25} + x^{20} + x^{12} + x^8 + 1, \\ p_1(x) &= x^{31} + x^{24} + x^{16} + x^{12} + 1, \\ p_2(x) &= x^{33} + x^{28} + x^{24} + x^4 + 1, \\ p_3(x) &= x^{39} + x^{36} + x^{28} + x^4 + 1. \end{aligned}$$

A chaque top d'horloge, les quatre registres sont avancés. Les quatre bits en sortie des registres, notés l_t^0, l_t^1, l_t^2 et l_t^3, sont alors considérés comme des entiers, et sont sommés pour obtenir une valeur y_t comprise entre 0 et 4.

La spécificité de E0 est d'utiliser deux états internes de 2 bits chacun. Au temps t, ces deux états sont notés $s_t = (s_t^0, s_t^1)$ et $c_t = (c_t^0, c_t^1)$. Au top d'horloge suivant, ces états sont mis à jour de la manière suivante :

$$\begin{aligned} s_{t+1} &= \left\lfloor \frac{y_t + c_t}{2} \right\rfloor \\ c_{t+1} &= s_{t+1} \oplus c_t \oplus T(c_{t-1}) \end{aligned}$$

où T est l'application de $\mathbb{F}_2^2 \mapsto \mathbb{F}_2^2$ définie par :

$$T : (x_1, x_0) \mapsto (x_0, x_1 \oplus x_0).$$

Le bit final généré à ce top d'horloge est alors : $z_t = l_t^0 \oplus l_t^1 \oplus l_t^2 \oplus l_t^3 \oplus c_t^0$.

L'état initial des registres est dérivé d'une clé, dont la taille, paire, est comprise entre 8 et 128 bits. Une fois l'état initial défini, le système fait avancer l'horloge deux cents fois, afin d'assurer la diffusion des bits de l'initialisation. Puis, les bits suivants sont utilisés pour produire la suite aléatoire.

Plusieurs tentatives d'attaques ont été menées contre E0. On pourra retenir l'attaque de Serge Vaudenay et Yi Lu [77], qui permet de retrouver l'initialisation des registres dans le cadre de l'utilisation de E0 dans le protocole de communication sécurisé Bluetooth : à partir de la connaissance des 24 premiers bits de 2^{35} trames consécutives (une trame faisant 2745 bits dans le protocole Bluetooth, on considère donc approximativement 2^{46} bits) il est possible de retrouver l'initialisation en environ 2^{39} opérations, après un calcul préalable de 2^{37} opérations. Cette attaque réduit considérablement la sécurité initiale de E0, dont la recherche exhaustive de clé est de 2^{128} opérations.

1.6.8 Achterbahn

Le cryptosystème Achterbahn [32, 30] a été proposé récemment dans le cadre du projet *eSTREAM* [22]. Ce cryptosystème reprend le principe de registres combinés, mais en faisant usage de *registre à décalage à rétroaction non linéaire* comme primitives de base. Il s'agit ici de combiner huit registres de ce type, grâce à une fonction de combinaison $R(x_7, \ldots, x_0)$ définie par :

$$R(x_7, x_6, \ldots, x_0) = x_7 \oplus x_6 \oplus x_5 \oplus x_4 \oplus x_1x_3 \oplus x_1x_2 \oplus x_0x_2 \oplus x_1x_2x_3 \oplus x_0x_1x_2.$$

Cette fonction est donc de degré 3, de non-linéarité 64 et est 4-résiliente (voir chapitre 2). Elle a principalement été choisie pour sa forme normale disjonctive très concise permettant une implémentation aisée en *hardware* :

$$R(x_7, x_6, \ldots, x_0) = x_7 + x_6 + x_5 + x_4 \vee x_1x_3 \vee x_1x_2 \vee x_0x_2.$$

La sortie de chaque registre est définie par une fonction de filtrage linéaire sur l'état du registre [29]. Les fonctions de rétroaction des registres ont été choisies de telle sorte que

pour un registre de taille n, sa période soit de longueur $2^n - 1$. Pour cela, les auteurs ont retiré l'état nul du cycle du graphe de transition de registres de période maximale. Cette manipulation permet donc d'obtenir une période en sortie de la fonction de combinaison beaucoup plus grande.

La principale faiblesse ayant été exploitée contre ce système concerne la taille des registres utilisés. En effet, les auteurs d'Achterbahn ont choisi de prendre des registres de tailles respectives 22, 23, 25, 26, 27, 28, 29 et 31. De ce fait, les cycles très courts des registres ont ammené à l'attaque décrite dans [40].

Des modifications ont été apportées au système par les auteurs dans [28] permettant de corriger certains défauts de la fonction de combinaison. Un résumé de l'état actuel de Achterbahn est disponible dans [31].

Chapitre 2

Attaques sur les systèmes de chiffrement à flot

2.1 Introduction

Nous allons nous intéresser dans ce chapitre aux différentes attaques connues contre les systèmes de chiffrement par flot. Nous verrons quelles sont les méthodes habituellement utilisées pour tenter de retrouver une partie du secret, ainsi que les précautions nécessaires pour s'en prémunir.

2.2 Attaque de Berlekamp-Massey

L'attaque de Berlekamp-Massey [4, 50] fut une des premières attaques à être menée sur les registres à décalage à rétroaction linéaire (voir chapitre 1). Son but est de construire, à partir d'une suite, le registre le plus petit capable de reconstruire la même suite. Cette attaque est à l'origine de la notion de *complexité linéaire* d'une suite.

Définition 3 *: La* complexité linéaire d'une suite *ultimement périodique* $(s_n)_{n\geq 0}$*, notée* $\lambda(s)$*, est la taille du plus petit registre à décalage à rétroaction linéaire capable d'engendrer cette suite.*

Définition 4 *: La* complexité linéaire d'un mot $w = w_0, w_1, \ldots, w_{k-1}$ *de* k *bits* $\lambda(w)$ *est définie comme étant la longueur du plus petit LSFR qui produit une séquence qui débute par ce mot.*

La complexité linéaire vérifie quelques propriétés évidentes :
- la complexité linéaire $\lambda(w)$ d'un mot w de k bits vérifie $\lambda(w) \leq k$,
- soit w un mot de k bits. Alors $\lambda(w) = k \Longleftrightarrow w = 0, 0, 0, \ldots, 0, 1$,
- soit $(s_n)_{n\geq 0}$ la suite définie par $\forall i, s_i = 0$, alors $\lambda(s) = 0$,
- soit $(s_n)_{n\geq 0}$ une suite de période k, alors $\lambda(s) \leq k$,
- soient $(s_n)_{n\geq 0}$ et $(t)_n$ deux suites, alors $\lambda(s \oplus t) \leq \lambda(s) + \lambda(t)$ et $\lambda(s \odot t) \leq \lambda(s) \times \lambda(t)$.

Pour toute suite $(s_n)_{n\geq 0}$, l'attaque permet, à partir de la connaissance de $2\lambda(s)$ bits consécutifs, de reconstruire le plus petit LFSR de taille $\lambda(s)$ engendrant la totalité de la suite. Il est donc très important, pour tout générateur pseudo-aléatoire, de faire en sorte que la suite produite soit de complexité linéaire la plus grande possible.

L'algorithme de l'attaque repose sur la contruction itérative d'un registre à décalage.

Définition 5 *[4, 50] Soient w un mot de longueur k, et w' le mot composé des $k-1$ premiers termes de w. On suppose connu un LFSR de taille $l = \lambda(w')$ engendrant une suite dont les $k-1$ premiers termes sont ceux de w', et dont la fonction de rétroaction est $r(x_{k-2}, x_{k-3}, \ldots, x_0)$. On appelle* anomalie suivante *(notée d_k) la quantité :*

$$d_k = w_{k-1} \oplus r(w_{k-2}, w_{k-3}, \ldots, w_{k-l-1})$$

L'attaque de Berlekamp-Massey va alors se dérouler ainsi : on considère que l'on a réussi à construire un LFSR de taille l et de fonction de rétroaction $r(x_{l-1}, \ldots, x_0)$ engendrant les $k-1$ premiers bits de $(s_n)_{n\geq 0}$. On calcule alors d_k :

- si $d_k = 0$, alors le LFSR est capable d'engendrer les k premiers termes de $(s_n)_{n\geq 0}$. Le plus petit LFSR engendrant la sous-suite des k premiers termes ne pouvant être de taille inférieure à celui engendrant la sous-suite des $k-1$ premiers termes, on conserve ce LFSR.
- si $d_k = 1$, alors le LFSR n'est pas capable d'engendrer les k premiers termes de $(s_n)_{n\geq 0}$. Soit $m < k$ le plus grand entier tel que $\lambda(s_0, s_1, \ldots, s_{m-1}) < l$, c'est-à-dire le rang dans la suite où l'on a effectué le dernier changement de complexité linéaire, et soit $b(x_{m-1}, \ldots x_0)$ la fonction de rétroaction engendrant les m premiers termes de $(s_n)_{n\geq 0}$. La création du LFSR de taille l' et de fonction de rétroaction r' capable d'engendrer les k premiers termes est alors :

$$\begin{aligned} l' &= \begin{cases} l \text{ si } l > k/2 \\ k+1-l \text{ sinon} \end{cases} \\ r'(x_{l'-1}, \ldots, x_0) &= r(x_{l-1}, x_{l-2}, \ldots, x_0) \oplus b(x_{k-1}, x_{k-2}, \ldots, x_{k-m}). \end{aligned}$$

La construction du plus petit LSFR est linéaire en fonction du nombre d'éléments lus dans la suite, ce qui rend cette attaque très efficace.

Il existe une extension à la notion de complexité linéaire, que l'on appelle *spectre de complexité linéaire.* Le *spectre de complexité linéaire* $\Lambda(s)$ correspond à l'ensemble des complexités linéaires des mots représentant les préfixes de $(s_n)_{n\geq 0}$, c'est-à-dire l'ensemble $\lambda(s_0)$, $\lambda(s_0, s_1)$, $\lambda(s_0, s_1, s_2)$, ... On pourra remarquer, de part sa méthode itérative, que l'algorithme de Berlekamp-Massey est capable de calculer le spectre d'une suite.

2.3 Attaque par corrélation

L'*attaque par corrélation* vise essentiellement les générateurs pseudo-aléatoires fondés sur la combinaison de suites pseudo-aléatoires, engendrées par des automates à états finis.

Considérons un générateur composé de n automates, dont les tailles respectives de leurs états internes sont L_i, combinés grâce à une fonction $F'(x_{n-1}, x_{n-2}, \ldots x_0)$. Ce générateur est donc équivalent à un automate à états finis dont l'état interne serait de $\sum L_i$ bits.

La méthode la plus naïve pour un attaquant souhaitant retrouver l'initialisation consiste à essayer toutes les valeurs possibles. L'ampleur de la tâche peut être colossale si les tailles L_i sont bien choisies, car il existe $\prod_{i=0}^{i=n-1} 2^{L_i}$ valeurs d'initialisation différentes.

Le but de l'*attaque par corrélation* [70] est d'essayer d'exclure un certain nombre d'automates, puis de retrouver l'initialisation des automates restant. Si l'on note E l'ensemble des automates exclus, le nombre de recherche des initialisations du système global devient alors $\prod_{i \in E} 2^{L_i} + \sum_{i \notin E} 2^{L_i}$.

L'attaque consiste à exploiter une éventuelle corrélation entre la sortie d'un registre et la sortie de la fonction F'. En effet, si la fonction F' n'a aucune corrélation avec une de ses entrées, alors pour une entrée x_i quelconque, $P[x_i = F'(x_0, x_1, \ldots, x_{n-1})] = 1/2$, ce qui en d'autres termes revient à dire que si l'on fixe arbitrairement l'entrée i de la fonction F' à une valeur constante, alors la sortie de la fonction reste imprédictible.

Dans le cas contraire, on dit que la fonction F' est corrélée à son entrée i, et il est alors possible de calculer un facteur de corrélation sur N bits entre la séquence en sortie de la fonction notée $(s_t)_{t \geq 0}$, et la séquence des bits produits par le registre i notée $(u_t)_{t \geq 0}$. Ce facteur est calculé de la manière suivante :

$$\alpha = \sum_{i=0}^{N-1} (-1)^{s_t + u_t}.$$

Il représente la *quantité de ressemblance* entre s_t et u_t : plus ce coefficient est grand (en valeur absolue), plus s_t et u_t sont semblables (à une complémentation près). Un α nul implique par contre une absence totale de corrélation entre s_t et u_t.

On peut considérer $X = \frac{1}{2}(N + \alpha)$ comme étant une variable aléatoire suivant une loi binomiale. On peut aisément calculer la valeur moyenne de cette variable aléatoire en posant p_i la probabilité que $F'(x_0, x_1, \ldots, x_{n-1})$ soit différent de x_i. Pour chaque élément de s_t et u_t, la valeur attendue est alors $(-1)p_i + 1(1 - p_i) = 1 - 2p_i$, ce qui nous amène à :

$$E(\alpha) = N(1 - 2p_i).$$

L'attaque consiste alors à tester toutes les initialisations possibles du registre i, et à calculer le coefficient de corrélation entre chacune de ces suites et la suite chiffrante : la bonne initialisation est alors celle dont le coefficient de corrélation se rapproche le plus de la valeur moyenne théorique (l'attaque suppose que la probabilité de *fausse alarme* est négligeable). L'état initial de ce registre est donc trouvé après 2^{L_i} essais, et l'attaque continue alors avec les autres registres.

2.3.1 Attaque par corrélation rapide

Dans le cas où les automates sont des LFSRs, il existe une variante de l'attaque par corrélation donnant de très bons résultats. L'*attaque par corrélation rapide* [16, 10] prend

en compte le fait que les entrées de la fonction de combinaison sont des suites à récurrence linéaire, et c'est cette linéarité qui est exploitée.

Cette attaque s'appuie sur la théorie des codes correcteurs.

Lorsqu'il est nécessaire de transmettre des données sur un canal potentiellement bruité, on a recours en général aux codes correcteurs : le principe est d'ajouter à la donnée de n bits à transmettre, une quantité de m bits de telle sorte que les $n+m$ bits ainsi transférés soient plus résistants aux erreurs que les n bits originaux. Ceci est possible grâce à l'introduction d'une redondance d'information dans les m bits ajoutés. Cet ajout est appelé *codage*.

Lors de la transmission de ce vecteur v de $n+m$ bits, il est possible qu'une erreur ε de poids w vienne s'ajouter au message. Le message reçu est alors $v \oplus \varepsilon$. Si, pour tout v, le destinataire du message est capable de se rendre compte que le message $v \oplus \varepsilon$ est erroné, on dit alors que le codage est w-détecteur. Si de plus, il est capable de corriger cette erreur en calculant ε, on dit alors que le codage est w-correcteur.

On utilise fréquemment des codes linéaires pour transmettre les messages à protéger du bruit. Le principe est alors de calculer le produit du vecteur à transmettre par une matrice G. Le vecteur ainsi obtenu est le vecteur transmis. Le destinataire connaissant lui aussi la matrice effectue le calcul, et est alors en mesure de vérifier que la transmition s'est correctement déroulée. En fonction des propriétés de la matrice G, il lui sera possible de corriger certaines erreurs.

Dans notre cas, nous pouvons voir les LFSRs comme étant des codeurs linéaires ; il s'agit d'ailleurs du moyen technique généralement mis en œuvre pour construire un codeur fondé sur le principe évoqué ci-dessus.

L'attaque par corrélation rapide va donc considérer la fonction de combinaison comme un canal fortement bruité, et le registre attaqué comme la source de la transmition d'un message v. Ce message est en fait l'initialisation du registre. Retrouver l'état initial du registre revient alors à décoder le message reçu en sortie de la fonction de combinaison.

Il est possible de déterminer la quantité de bits nécessaires en sortie de f pour s'assurer de la reconstruction de l'état initial du registre attaqué : la fonction est considérée comme un canal binaire symétrique, c'est-à-dire que la probabilité qu'un bit soit erroné lors de son passage dans le canal (0 se transformant en 1, ou 1 se transformant en 0) est de p. Cette probabilité est à mettre en relation avec la probabilité de corrélation entre l'entrée i de la fonction et sa sortie. La capacité de ce canal est alors donnée par :

$$C(p) = 1 + p\log_2 p + (1-p)\log_2(1-p).$$

Quand p est proche de $1/2$, c'est-à-dire que $p = 1/2 - \varepsilon$ avec ε petit, alors $C(p)$ peut être approchée par $C(p) \approx 2\varepsilon^2/ln(2)$. Sachant que le taux de transmission du code (le rapport entre la quantité de données à transmettre et la quantité réellement transmise) ne doit pas dépasser la capacité du canal, nous obtenons :

$$\frac{L_i}{N} \leq C(p)$$

soit, puisque tout est positif,

$$N \geq \frac{L_i}{N} \approx \frac{ln(2)L_i}{2\varepsilon^2}.$$

Cette borne est appelée *borne de Shannon.* L'attaque s'appuie alors sur des techniques classiques de décodage de codes linéaires rapides.

2.4 Attaques algébriques

Les *attaques algébriques* ont été introduites par Courtois et Meier dans [18, 20].

Le principe est de trouver un ensemble d'équations de petit degré décrivant le cryptosystème attaqué, et de le résoudre.

Prenons le cas de registres combinés (voir chapitre 1). Soient n LFSRs de tailles respectives $L_0, L_1, \ldots L_{n-1}$, de fonctions de rétroaction $f_0(x_{L_0-1}, \ldots x_0)$, $f_1(x_{L_1-1}, \ldots x_0)$, $\ldots$, $f_{n-1}(x_{L_{n-1}-1}, \ldots x_0)$, et soit la fonction de combinaison $F'(x_{n-1}, \ldots x_0)$.

Les fonctions f_i étant linéaires, on peut considérer que le système est équivalent à un automate dont l'état interne serait un mot de $k = \sum_{i=0}^{n-1} L_i$ bits, noté $K = (K_{k-1}, K_{k-2}, \ldots K_0)$, mis à jour par une fonction linéaire $L(x_{k-1}, \ldots, x_0)$. Les bits de sortie $s_0, s_1, \ldots$ sont produits par une fonction $F(x_{k-1}, \ldots, x_0)$. Il est alors possible de décrire le générateur sous la forme d'un *système d'équations surdéterminé* :

$$\begin{cases} s_0 = F(K_{k-1}, K_{k-2}, \ldots, K_0) \\ s_1 = F(L(K_{k-1}, K_{k-2}, \ldots, K_0)) \\ s_2 = F(L \circ L(K_{k-1}, K_{k-2}, \ldots, K_0)) \\ s_3 = F(L \circ L \circ L(K_{k-1}, K_{k-2}, \ldots, K_0)) \\ \vdots \end{cases}$$

Le principal intérêt du fait que L soit linéaire, est qu'alors $L \circ L \circ \ldots \circ L$ est aussi linéaire. Le seul problème à la résolution du système reste le degré de F, généralement trop élevé. Le but est alors de diminuer ce degré : nous allons voir différentes méthodes pour y arriver.

2.4.1 Attaques algébriques par approximation

Une méthode fréquemment employée consiste à approcher F par une fonction F' de petit degré. L'attaque n'est alors possible que si l'approximation de la fonction est bonne, donc $P(F(\mathrm{x}) = F'(\mathrm{x})) \approx 1$. Ensuite, ces équations sont résolues grâce à des méthodes connues, telles que le calcul de *bases de Gröbner* [1] ou l'*algorithme XL* [19].

2.4.2 Attaques algébriques et multiples de petit degré

Le principe est cette fois de trouver un polynôme non nul $a(\mathrm{x})$ de petit degré d tel que $\forall \mathrm{x}, a(\mathrm{x})F(\mathrm{x}) = 0$. Ce polynôme est appelé *polynôme annulateur* de $F(\mathrm{x})$. Il suffit de remarquer que pour toute sortie s_i du générateur telle que $s_i = 1$, nous obtenons :

$$a(L^i(\mathrm{x}))F(L^i(\mathrm{x})) = 0 \Longleftrightarrow a(L^i(\mathrm{x})) = 0$$

Pour chacune des sorties du générateur telle que $s_i = 1$, nous obtenons donc une équation de degré d.

S'il n'est pas possible de trouver un polynôme annulateur de $F(\mathrm{x})$ de petit degré, il est possible de procéder dans l'autre sens, en cherchant cette fois-ci un polynôme annulateur de $1 + F(\mathrm{x})$; les auteurs de [18] et [20] affirment qu'il est toujours possible de se ramener à un de ces deux cas, même si ce n'est pas trivial. Nous nous intéressons cette fois-ci non plus aux sorties s_i telles que $s_i = 1$, mais aux sorties $s_i = 0$ pour obtenir nos équations.

De la même manière que pour la méthode par approximation, on utilise les algorithmes de résolution de systèmes d'équations connus pour déterminer les K_i.

Cette attaque permet d'introduire la notion d'*immunité algébrique* d'un polynôme.

Définition 6 *[20] Soit $\mathcal{AN}(F)$ l'ensemble des polynômes annulateurs du polynôme F.* L'immunité algébrique *d'un polynôme $F(\mathrm{x})$ est le degré minimal des polynômes éléments de $\mathcal{AN}(F) \cup \mathcal{AN}(1 \oplus F)$.*

2.4.3 Attaques algébriques augmentées

Il s'agit d'une variante des attaques algébriques beaucoup plus générale : nous n'allons plus considérer un bit de sortie à la fois pour construire notre système d'équations, mais plutôt un ensemble de m bits de sortie. On construit donc un polynôme $H(\mathrm{x})$ de $\mathbb{F}_2^k \mapsto \mathbb{F}_2^m$ de la forme :

$$\begin{aligned} \mathrm{y} &= (s_t, s_{t+1}, \ldots s_{t+m-1}) \\ H(K) &= (h(L^t(K)), h(L^{t-1}(K)), \ldots h(L^{t+m-1}(K))) \\ H(K) &= \mathrm{y} \end{aligned}$$

On recherche alors des polynômes $g(\mathrm{y}, K)$ de bas degré tels que $g(\mathrm{y}, K) = 0$. En général, il n'est pas trivial de trouver de telles équations, mais une fois déterminées, elles peuvent mener à des attaques particulièrement efficaces.

2.5 Attaque par compromis temps-mémoire

Il s'agit ici de l'attaque la plus générique. Cette attaque a été présenté contre les systèmes de chiffrement par bloc par Hellman dans [37], puis généralisée pour les systèmes de chiffrement à flot par Babbage [2] et Golic [34].

Définition 7 *Soit deux ensembles finis $\mathcal{A}$ et $\mathcal{B}$, et une fonction $f : \mathcal{A} \mapsto \mathcal{B}$. f est appellée* fonction à sens unique *si et seulement si pour tout $x \in \mathcal{A}$ il est facile de calculer $f(x)$, alors que pour tout $y \in \mathcal{B}$, il est difficile de déterminer son antécédant par f.*

Le but premier de cette attaque est de tenter de retrouver l'antécédant d'un élément quelconque de $\mathcal{B}$.

Soit $y \in \mathcal{B}$, élément dont on cherche l'antécédant.

La méthode la plus évidente serait de calculer, pour tous les éléments $x \in \mathcal{A}$, la valeur $f(x)$ et de la comparer à y. Bien évidemment, cette méthode est impraticable si le cardinal

de $\mathcal{A}$ est important. De plus, si à l'avenir on cherche à déterminer l'antécédant d'un autre élément de $\mathcal{B}$ alors tous les calculs sont à recommencer.

Une deuxième méthode serait de calculer une fois pour toute la valeur de $f(x)$ pour tous les éléments $x \in \mathcal{A}$, et d'enregistrer les couples $(x, f(x))$ dans une table, triée par rapport aux $f(x)$. Lors de l'attaque, il suffit alors de rechercher dans cette table, le couple dont le deuxième élément est celui que l'on souhaite inverser. Cette méthode est extrêmement rapide, mais malheureusement, la phase de pré-calcul de la table est très longue et très coûteuse en termes de mémoire.

Il s'agit alors de trouver un compromis entre le temps de calcul, et la quantité de mémoire consommée, en créant une table contenant uniquement un sous-ensemble de ces couples.

Pour cela, nous allons nous intéresser au cas où les ensembles $\mathcal{A}$ et $\mathcal{B}$ sont respectivement de la forme $\mathbb{F}_2^n$ et $\mathbb{F}_2^l$.

Dans le cas où $n = l$, nous pouvons utiliser une astuce permettant de stocker dans la table un plus grand nombre de valeurs. En effet, dans ce cas précis, il est possible de composer f avec elle même. Si l'on choisit un élément quelconque $x \in \mathbb{F}_2^n$, on peut alors calculer l'ensemble s de t éléments :

$$s = x, f(x), f \circ f(x), \ldots, f^{t-1}(x).$$

Conservons uniquement le dernier élément, $f^{t-1}(x)$. Imaginons que l'on cherche à calculer l'antécédant d'un élément y. Il suffit alors de calculer l'ensemble $y, f(y), f \circ f(y), \ldots, f^t(y)$, et si, pendant le calcul, un des éléments est égal à $f^t(x)$, alors cela signifie que y est l'un des éléments de s.

Dans le cas où $n \neq l$, il suffit de choisir un ensemble de r fonctions quelconques, rapides à calculer de $\mathbb{F}_2^l \mapsto F^n$, notées h_0, h_1, ..., h_{r-1}. Ainsi, les fonctions $h_0 \circ f$, $h_1 \circ f$, ..., $h_{r-1} \circ f$ sont toutes des fonctions de $\mathbb{F}_2^n \mapsto \mathbb{F}_2^n$, et il est de nouveau possible d'appliquer la méthode précédente pour ces fonctions.

Cette attaque est donc paramétrée par t, m le nombre d'éléments que l'on souhaite conserver dans la table, et r le nombre de fonctions choisies. Le *paradoxe des anniversaires* nous dit que dans un ensemble de N éléments, si l'on tire aléatoirement $\sqrt{N}$ éléments x_0, x_1, ... de cet ensemble, alors la probabilité qu'il existe i et j, $i \neq j, i < \sqrt{N}, j < \sqrt{N}$ tels que $x_i = x_j$ est de $0,5$. Ainsi, pour s'assurer que les points stockés dans la table aient une grande probabilité d'être tous différents, il faut s'assurer que $mt^2 < N$.

Pour ce qui concerne le chiffrement à flot, on considère que la fonction à sens unique que l'on cherche à inverser est la fonction qui associe aux n bits de l'état interne du générateur les n premiers bits du flux de sortie. À partir de la connaissance de $D \geq n$ bits consécutifs de la suite chiffrante, on construit $D - n$ valeurs de $\mathbb{F}_2^n$, que l'on tente d'inverser avec cette méthode. Si pour l'une de ces valeurs on est capable de retrouver son antécédant, alors l'initialisation du générateur est retrouvée, et il est alors possible de calculer l'intégralité de la suite chiffrante.

2.6 Synthèse des attaques

Tout au long de ce chapitre, nous avons pu voir les différentes attaques connues contre les systèmes de chiffrement à flot. Ces attaques ont mené à divers critères de sécurité devant être respectés lors de la conception d'un système de chiffrement.

Résistance à l'attaque de Berlekamp-Massey

Cette attaque s'applique dans le cas où la complexité linéaire de la suite produite est faible : il faut donc, pour s'en prémunir, augmenter la complexité linéaire des suites produites.

Dans le cas de registres à décalage à rétroaction linéaire combinés par une fonction F, il faut choisir *une fonction de combinaison de haut degré.* Ainsi, la complexité de la suite produite sera soumise aux règles de sommation et de multiplication de suites binaires énoncées dans la section sur l'attaque de Berlekamp-Massey.

Par exemple, considérons l'utilisation de trois registres à décalage de période maximale, deux à deux premières entre elles. On note $\lambda_0, \lambda_1, \lambda_2$ les complexités linéaires respectives de ces trois registres. Soit $F(x_2, x_1, x_0) = x_2x_1x_0 \oplus x_2x_0 \oplus x_1x_0 + x_0$ une fonction de combinaison. Alors, le générateur formé de ces trois registres et de cette fonction de combinaison produira des suites dont la complexité linéaire λ vérifie :

$$\lambda = \lambda_2 \times \lambda_1 \times \lambda_0 + \lambda_2 \times \lambda_0 + \lambda_1 \times \lambda_0 + \lambda_0.$$

Résistance à l'attaque par corrélation

Il n'est malheureusement pas possible de pouvoir complètement contrer cette attaque portant sur la fonction de combinaison. En effet, la seule fonction qui permet de se prémunir totalement d'une attaque par corrélation est la fonction $F(x_{n-1}, x_{n-2}, \ldots, x_0) = x_{n-1} \oplus x_{n-2} \oplus x_0$. Malheureusement, cette fonction ne permet par d'augmenter correctement la complexité linéaire de la suite produite : il faudra donc opter pour un compromis.

Cette attaque a amené à définir le critère d'*immunité aux corrélations* :

Définition 8 *[70] Une fonction F définie sur $\mathbb{F}_2^n \mapsto \mathbb{F}_2$ est* immune aux corrélations d'ordre t *si et seulement si lorsque l'on fixe entre $1 \leq k \leq t$ variables à une valeur arbitraire, la distribution des valeurs en sortie de F reste équilibrée. De plus, si la fonction est équilibrée, alors on dit que la fonction est* t-résiliente.

Concernant l'attaque par corrélation rapide, celle-ci ne s'applique que dans le cas où la fonction de combinaison ne possède pas un degré de résilience assez haut, et dans le cas de l'utilisation de registres à décalage à rétroaction linéaire en entrée de la fonction de combinaison.

Il existe de nombreuses méthodes de construction de fonctions ayant un haut degré d'immunité aux corrélations [26, 65, 66, 59].

Il est aussi possible de tenter une approximation de F par une fonction affine, puis de mener l'attaque par corrélation sur ce nouveau cryptosystème. Il faut donc prendre garde,

en plus du critère d'immunité aux corrélations, à considérer F la plus éloigné possible des fonctions affines. On dit d'une fonction F qu'elle est de *non-linéarité* k si et seulement si la plus petite distance, en terme de *distance de Hamming* séparant sa table de vérité de celles des fonctions affines, est d'au moins k.

Une méthode de calcul de l'ordre de résilience, ainsi que de la non-linéarité d'une fonction est donnée annexe A.

Attaque par compromis temps-mémoire

Tout comme pour l'attaque par corrélation, il n'est pas possible de se protéger totalement de l'attaque par compromis temps-mémoire. On préconise en général que l'état interne du générateur soit au moins deux fois plus grand que la taille de la clé de chiffrement. Ainsi, la recherche de l'état interne du générateur est au moins aussi longue que la recherche de la clé.

Attaques algébriques

Les attaques algébriques sont des attaques particulièrement efficaces. Il faut prendre de grandes précautions si l'on souhaite ne pas les subir.

Ces attaques s'appuient essentiellement sur le fait que l'actualisation de l'état interne décrivant l'automate du générateur pseudo-aléatoire est linéaire, ou bien assez proche d'une fonction linéaire. Dans ce cas, il est alors possible de construire un système d'équations qui permettra de retrouver la clé du cryptosystème.

Afin d'éviter ces attaques, il est possible d'opter pour une mise à jour non linéaire de l'état interne du générateur. Ainsi, le degré algébrique des équations grandit très vite au fur et à mesure des compositions de la fonction d'actualisation, ce qui rend l'attaque impraticable.

Chapitre 3

Attaque d'un cas particulier de générateur reposant sur les T-fonctions

3.1 Introduction

Nous allons nous intéresser ici à la reconstruction de l'état interne du générateur pseudo-aléatoire sommairement exposé dans la section 3.2. Cet article a été présenté à la conférence *Sequences And Their Applications* [8] en 2004. Pour cela, nous nous plaçons du point de vue d'un attaquant cherchant à retrouver cet état interne à partir de l'unique connaissance du plus petit nombre possible de sorties consécutives.

3.2 Les T-Fonctions

Les *T-Fonctions*, ou *fonctions triangulaires*, ont été introduites par Alexander Klimov et Adi Shamir [45, 46]. Le principal objectif de ces fonctions est de permettre l'utilisation de primitives mathématiques simples que l'on trouve classiquement sur la plupart des microprocesseurs actuels, pour construire des fonctions complexes, non-linéaires, rapides à calculer, et utilisables, par exemple, lors de la création de suites pseudo-aléatoires.

La principale originalité de ce type de construction est, sans conteste, la possibilité de mélanger des fonctions arithmétiques classiques, comme l'addition ou la multiplication, avec des fonctions booléennes comme par exemple le `OU` logique. Cette particularité permet, entre autres, de considérablement compliquer les attaques purement arithmétiques sur ce type d'objets.

3.2.1 Définitions et notations

Tout au long de la description des T-fonctions, nous noterons indifféremment à l'aide du symbole x le vecteur de n bits $(x_{n-1}, x_{n-2}, \ldots, x_1, x_0) \in \mathbb{F}_2^n$ et l'entier modulo 2^n, $\sum_{i=0}^{n-1} 2^i x_i$. La notation $[\mathrm{x}]_i$ désigne le $i+1^{\mathrm{e}}$bit de x, avec par convention, $[\mathrm{x}]_0$ le bit le plus à droite.

Définition 9 *[45] Une fonction* $\phi : \mathbb{F}_2^{k \times n} \longrightarrow \mathbb{F}_2^n$ *est dite* primitive *si*

- $k = 1$ *et* ϕ *est l'identité* $\phi(x) = x$*, l'opération de négation* $\phi(\mathrm{x}) = -\mathrm{x}$*, ou l'opération de complémentation* $[\phi(\mathrm{x})]_i = \overline{[\mathrm{x}]_i}$*, pour tout* i*,*
- $k = 2$ *et* ϕ *est l'opération d'addition* $\phi(\mathrm{x}, \mathrm{y}) = \mathrm{x} + \mathrm{y} \mod 2^n$*, ou l'opération de soustraction* $\phi(\mathrm{x}, \mathrm{y}) = \mathrm{x} - \mathrm{y} \mod 2^n$*, ou l'opération de multiplication* $\phi(\mathrm{x}, \mathrm{y}) = \mathrm{xy} \mod 2^n$*, ou encore toute opération booléenne classique, telle que le* ***ET*** *logique* $[\phi(\mathrm{x}, \mathrm{y})]_i = [\mathrm{x}]_i \wedge [\mathrm{y}]_i$*, le* ***OU*** *logique* $[\phi(\mathrm{x}, \mathrm{y})]_i = [\mathrm{x}]_i \vee [\mathrm{y}]_i$*, ou encore le* ***OU-EXCLUSIF*** $[\phi(\mathrm{x}, \mathrm{y})]_i = [\mathrm{x}]_i \oplus [\mathrm{y}]_i$*.*

Il est à noter que le décalage à gauche est autorisé, car équivalent à une multiplication par une puissance de 2, mais pas le décalage à droite, ni la division ou les rotations.

Toutes ces fonctions primitives sont généralement présentes sur les microprocesseurs actuels, et implémentées en une seule instruction. De plus, elles sont en général (à part la multiplication) capables de s'exécuter en un seul cycle CPU : ainsi, en utilisant une combinaison de ces fonctions primitives, nous sommes assurés d'obtenir une fonction rapide à calculer.

Nous nous intéressons maintenant à la construction de fonctions inversibles, composées uniquement de fonctions primitives. Notons au passage que nous cherchons à construire des fonctions inversibles, mais que nous ne ferons que prouver l'existence de la fonction inverse sans nécessairement pouvoir la donner explicitement.

Par convention, dans la suite de ce document, nous considérerons que les lignes et les colonnes des matrices sont numérotées à partir de 0. De plus, pour une matrice M de taille $m \times n$ (m lignes et n colonnes), nous noterons $M_{i-1,\star}$ sa i^{e} ligne, et $M_{\star,j-1}$ sa j^{e} colonne.

De plus, l'entrée d'une fonction ayant m vecteurs de n bits sera représentée par une matrice $m \times n$, où chaque ligne de la matrice représente une entrée de la fonction. Il en sera de même pour sa sortie.

Définition 10 *[45] Une fonction* $\phi : \mathbb{F}_2^{m \times n} \longrightarrow \mathbb{F}_2^{l \times n}$ *est appelée* fonction triangulaire *ou* T-fonction*, si le calcul de la* k^{e} *colonne de la sortie* $[\phi(\mathrm{x})]_{\star,k-1}$ *ne dépend que des* k *premières colonnes de l'entrée* $[\mathrm{x}]_{\star,0}, [\mathrm{x}]_{\star,1}, \ldots, [\mathrm{x}]_{\star,k-1}$*.*

Le principe d'une fonction triangulaire est donc de calculer les colonnes de sa sortie au fur et à mesure de la connaissance des colonnes de son entrée : pour calculer la première colonne de la sortie, seule la connaissance de la première colonne est nécessaire, puis pour calculer la valeur de la deuxième colonne, seule la connaissance de la première et de la deuxième colonne est nécessaire, et ainsi de suite.

Il est alors facile de vérifier que toutes les fonctions primitives décrites dans la définition 9 sont triangulaires. Prenons l'exemple de l'addition de x et de y, deux mots de n bits : pour calculer la valeur du i^{e} bit de sortie, nous effectuons la somme, modulo 2, du i^{e} bit de x et du i^{e} bit de y, puis nous ajoutons à cette somme la retenue engendrée par les calculs précédents, calculs qui ne dépendent que des bits précédents le i^{e} bit de x et le i^{e} bit de y. Ainsi, pour calculer le i^{e} bit du résultat de la somme de deux mots ne nécessite que la connaissance des i premiers bits de ces deux mots, l'addition est donc bien une fonction triangulaire.

Nous allons maintenant décrire une méthode simple de construction de *fonctions inversibles* à partir de fonctions primitives.

Théorème 3 *[45] Soit f une composition quelconque de fonctions primitives. La fonction $\phi : \mathrm{x} \mapsto \mathrm{x} + 2f(\mathrm{x}) \mod 2^n$ est inversible.*

Il est à noter que cette classe de fonctions inclut la fonction $f(\mathrm{x}) = \mathrm{x} + 2\mathrm{x}^2 \mod 2^n$, utilisée dans *RC6*, un des candidats au concours *AES* [56] dont le but était de remplacer le vieillissant standard de chiffrement Américain *DES*, par un algorithme plus sûr.

Preuve : Afin de montrer que la fonction ϕ est inversible, nous allons tenter de retrouver x à partir de la connaissance de $\phi(\mathrm{x})$.

Considérons que nous avons réussi à retrouver les k premiers bits de x. Nous cherchons donc maintenant à retrouver la valeur de $[\mathrm{x}]_k$ (nous rappelons que les bits sont numérotés à partir de 0). Puisque la multiplication par 2 correspond à un décalage vers la gauche, le $k + 1^{\mathrm{e}}$ bit de $\phi(\mathrm{x})$ est la somme, modulo 2, du $k + 1^{\mathrm{e}}$ bit de x et du k^{e} bit de $f(\mathrm{x})$.

Or, si nous avons réussi à déterminer la valeur de $[\mathrm{x}]_0, \ldots [\mathrm{x}]_{k-1}$, il est facile de calculer la valeur de $[f(\mathrm{x})]_{k-1}$. En effet, f étant composée de fonctions triangulaires, elle est elle-même triangulaire et, par conséquent, le calcul de $[f(\mathrm{x})]_{k-1}$ ne dépend que de $[\mathrm{x}]_0, \ldots, [\mathrm{x}]_{k-1}$. Connaissant la valeur de $[\phi(\mathrm{x})]_k$, nous obtenons :

$$[\mathrm{x}]_k \oplus [f(\mathrm{x})]_{k-1} = [\phi(\mathrm{x})]_k,$$

ce qui donne

$$[\mathrm{x}]_k = [f(\mathrm{x})]_{k-1} \oplus [\phi(\mathrm{x})]_k.$$

Il nous reste à déterminer s'il est possible de calculer $[\mathrm{x}]_0$ à partir de $\phi(\mathrm{x})$. Si l'on regarde de plus près, nous avons $\phi(\mathrm{x}) \mod 2 = (\mathrm{x} + 2f(\mathrm{x})) \mod 2 = \mathrm{x} \mod 2$, ce qui signifie que

$$[\mathrm{x}]_0 = [\phi(\mathrm{x})]_0.$$

◇

Nous allons maintenant introduire la notion de *réseau de Feistel triangulaire*. Rappelons tout d'abord ce qu'est le schéma de Feistel [23] : il s'agit d'une méthode permettant de construire des fonctions inversibles à partir d'une fonction quelconque. Le principe est assez simple : soit une fonction $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^n$, alors la fonction suivante est inversible :

$$\begin{array}{rccc} F : & \mathbb{F}_2^{2\times n} & \longrightarrow & \mathbb{F}_2^{2\times n} \\ & (h, l) & \longmapsto & (l, h \oplus f(l)) \end{array}$$

En effet, il suffit de considérer l'application $H(h, l) \mapsto (l \oplus f(h), h)$

$$\begin{array}{rl} & H(F(h,l)) \\ = & H(l, h \oplus f(l)) \\ = & (h \oplus f(l) \oplus f(l), l) \\ = & (h, l) \end{array} \tag{3.1}$$

Ce principe est d'ailleurs appliqué par exemple, dans le chiffrement *DES*, où il apparaît successivement 16 fois par bloc à chiffrer : on parle alors de *réseau de Feistel.*

Introduisons une nouvelle définition :

Définition 11 *[45] Une* fonction paramétrique *est une fonction que l'on peut écrire sous la forme* $g(x_1, \ldots, x_a; \alpha_1, \ldots, \alpha_b)$*, où les arguments sont séparés par un point-virgule en deux entités que l'on nomme les entrées (les* x_i*), et les paramètres (les* α_i*). Une fonction paramétrique inversible, ou* PIF *(Parametric Invertible Function), est une fonction paramétrique où la relation entrée/sortie est inversible pour tout paramètre fixé :*

$$\forall \alpha, \forall \mathrm{x}, \mathrm{y} : g(\mathrm{x}, \alpha) = g(\mathrm{y}, \alpha) \Longleftrightarrow \mathrm{x} = \mathrm{y}$$

Grâce à cette définition, nous pouvons maintenant formaliser la notion de *réseau de Feistel triangulaire* :

Définition 12 *[45] Un* Réseau de Feistel Triangulaire*, noté TFN (pour Triangular Feistel Network), est une application g définie par*

$$\begin{aligned} g : \mathbb{F}_2^{m \times n} &\longrightarrow \mathbb{F}_2^{m' \times n} \\ p &\longmapsto (g_0(p_{\star,0}), g_1(p_{\star,1}; p_{\star,0}), \\ &\qquad g_2(p_{\star,2}; p_{\star,0}, p_{\star,1}), \ldots, g_{n-1}(p_{\star,n-1}; p_{\star,0}, \ldots, p_{\star,n-2})) \end{aligned}$$

où les g_i*,* $i \geq 1$ *sont des fonctions paramétriques inversibles, et* g_0 *une fonction inversible.*

Il est facile de prouver que toute *fonction triangulaire inversible* est un *réseau de Feistel triangulaire.*

3.2.2 Inversibilité d'une fonction paramétrique

Nous allons maintenant décrire une méthode permettant de construire une fonction inversible à partir de primitives, ainsi qu'une méthode permettant de déterminer si une fonction triangulaire quelconque est inversible [45]. Le problème complet étant difficile, nous ne nous intéresserons ici qu'au cas des fonctions univariées et bivariées.

Tester l'inversibilité d'une composition de fonctions primitives revient à déterminer s'il s'agit d'un réseau de Feistel triangulaire : nous devons déterminer si le calcul du premier bit de la sortie est inversible, puis si le calcul de chacun des bits suivants est bien déterminé par une fonction paramétrique inversible.

Pour ce faire, la méthode la plus simple serait de tester toutes les entrées possibles pour chacune de ces fonctions paramétriques, puis de vérifier que nous sommes bien face à une fonction inversible. Malheureusement, il n'est pas possible de procéder ainsi, car la complexité du calcul grandit exponentiellement en fonction du nombre de bits des entrées. Afin de simplifier tout ceci, pour chacune des fonctions primitives, nous allons généraliser le calcul d'un bit de sortie en fonction de son rang en ignorant les paramètres tels que les retenues. Ainsi, il sera facile de déterminer la forme générale d'une composition de ces fonctions.

Considérons le cas le plus simple, à savoir celui des fonctions booléennes. Soit $\circ$ une fonction booléenne (OU, OU – EXCLUSIF, ET, ...), et soient a et b deux mots de $\mathbb{F}_2^n$. Nous obtenons :

$$\forall i, [a \circ b]_i = [a]_i \circ [b]_i.$$

En effet, le calcul des fonctions booléennes s'effectue *bit à bit.*

Considérons maintenant le cas de l'addition modulo 2^n. Ce cas est légèrement plus complexe, puisque le calcul du i^{e} bit de la sortie ne dépend pas uniquement du i^{e} bit des entrées. En fait, le calcul de l'addition se déroule comme ceci :

$$\begin{aligned} [a+b]_0 &= [a]_0 \oplus [b]_0 \\ \forall i > 0, [a+b]_i &= [a]_i \oplus [b]_i \oplus \alpha \end{aligned}$$

où α représente la retenue ne dépendant que des bits de 0 à $i-1$ de a et b.

D'une manière générale, les fonctions primitives peuvent être mises sous les formes suivantes [45] :

Primitive	Premier bit	Bits suivants ($i > 0$)
$\overline{a}$	$[\overline{a}]_0 = \overline{[a]_0}$	$[\overline{a}]_i = \overline{[a]_i}$
$\circ =$ OU, OU – EXCLUSIF, ET	$[a \circ b]_0 = [a]_0 \circ [b]_0$	$[a \circ b]_i = [a]_i \circ [b]_i$
$a + b$	$[a+b]_0 = [a]_0 \oplus [b]_0$	$[a+b]_i = [a]_i \oplus [b]_i \oplus \alpha$
$a - b$	$[a-b]_0 = [a]_0 \oplus [b]_0$	$[a-b]_i = [a]_i \oplus [b]_i \oplus \alpha$
$a \times b$	$[a \times b]_0 = [a]_0 \wedge [b]_0$	$[a \times b]_i = ([a]_i \wedge [b]_0) \oplus ([a]_0 \wedge [b]_i) \oplus \alpha$
$\forall k \in \mathbb{N}, a << k$	$[a << k]_0 = 0$	$\forall i < k, [a << k]_i = 0$ $\forall i \geq k, [a << k]_i = [a]_{i-k}$
$\forall k \in \mathbb{N}^\star, [a^k]$	$[a^k]_0 = 0$	si k est pair, $[a^k]_i = \alpha$ sinon, $[a^k]_i = [a]_i \wedge [a]_0 \oplus \alpha$

où les α sont des retenues ne dépendant que des bits précédents.

Grâce à ces décompositions, nous allons maintenant pouvoir déterminer si une fonction résultant de la composition de fonctions triangulaires peut être mise sous la forme d'un réseau de Feistel Triangulaire. Prenons un exemple :

Considérons l'application $\mathbb{F}_2^{2\times n} \longrightarrow \mathbb{F}_2^{2\times n}$ définie par :

$$f(\mathrm{x}, \mathrm{y}) = (\mathrm{x} + 2(\mathrm{x} \wedge \mathrm{y}), (\mathrm{y} + 3\mathrm{x}^3) \oplus \mathrm{x}).$$

Une analyse purement algébrique de cette fonction ne peut pas permettre de déterminer s'il s'agit d'une fonction inversible. En effet, le mélange d'opérateurs tels que l'addition

classique ou la multiplication, avec des opérateurs booléens vient compliquer l'analyse du problème.

Malgré tout, il ne s'agit ici que de fonctions primitives. Nous allons donc tenter d'écrire cette fonction en utilisant le tableau donné précédemment :

$$\begin{cases} x' &= x + 2(x \wedge y) \\ y' &= (y + 3x^3) \oplus x \end{cases}$$

$$\begin{aligned} [x']_i &= [x]_i \oplus [2(x \wedge y)]_i] \oplus \alpha \\ &= [x]_i \oplus [x \wedge y]_{i-1} \oplus \alpha \\ &= [x]_i \oplus ([x]_{i-1} \wedge [y]_{i-1}) \oplus \alpha \\ &= [x]_i \oplus \beta \end{aligned}$$

avec β une quantité dont la valeur ne dépend que des bits de x et y précédant le rang actuel i.

$$\begin{aligned} [y']_i &= [y + 3x^3]_i \oplus [x]_i \\ &= [y]_i \oplus [3x^3]_i \oplus [x]_i \oplus \gamma \\ &= [y]_i \oplus ([3]_i \wedge [x^3]_0) \oplus ([3]_0 \wedge [x^3]_i) \oplus \gamma \oplus \delta \\ &= [y]_i \oplus [x^3]_i \oplus \gamma \oplus \delta \\ &= [y]_i \oplus [x]_i \wedge [x]_0 \oplus \varepsilon \end{aligned}$$

On voit facilement que ces deux réécritures aboutissent à des fonctions inversibles pour $i > 0$. Concernant les bits de poids faible, nous obtenons :

$$[x']_0 = [x]_0 \oplus [2(x \wedge y)]_0 = [x]_0$$

et

$$[y']_0 = [x]_0 \oplus [(y + 3x^3)]_0 = [x]_0 \oplus [y]_0 \oplus ([3]_0 \wedge [x^3]_0) = [x]_0 \oplus [y]_0.$$

Ces deux fonctions sont, elles aussi, inversibles. La fonction f peut donc être écrite sous la forme d'un réseau de Feistel triangulaire, et nous pouvons donc affirmer qu'elle est inversible.

Il est possible de généraliser un peu plus ces résultats. Dans le cas univarié, il n'existe que quatre fonctions booléennes de $\mathbb{F}_2 \to \mathbb{F}_2$. Ces quatre fonctions sont $x \mapsto 0, x \mapsto 1, x \mapsto x$ et $x \mapsto x \oplus 1$. Parmi ces quatre fonctions, seules les deux dernières sont inversibles. Si l'on s'intéresse au cas des fonctions triangulaires, et du calcul d'un bit en particulier, on peut dire que la fonction est inversible si et seulement si le calcul de chacun de ses bits est de la forme $[x]_i \oplus \alpha$, ou α est calculé en fonction des bits précédents.

Théorème 4 *Une fonction triangulaire f est inversible si et seulement si le calcul de $[f(x)]_0$ est inversible, et $\forall i > 0$, f peut être mise sous la forme $[f(x)]_i = [x]_i \oplus \alpha$, avec α une quantité ne dépendant que de $[x]_{i-1}[x]_{i-2} \ldots [x]_0$.*

3.2.3 Construction de suites pseudo-aléatoires

Ce que nous recherchons en général lorsque l'on construit un générateur pseudo-aléatoire, c'est qu'il soit rapide, et de très grande période. Concernant le premier point, l'utilisation de fonctions triangulaires dans le cadre d'une implémentation en logiciel semble toute indiquée. Nous allons donc étudier une méthode de construction de fonction de transition pour un générateur pseudo-aléatoire proposée dans [45], qui résulte de la composition d'un petit nombre de fonctions triangulaires, et qui nous garantit une grande période.

Analyse exhaustive de petites compositions

La première chose réalisable est une analyse exhaustive de toutes les fonctions triangulaires inversibles que l'on peut construire à partir d'un nombre réduit de compositions des fonctions primitives. Cette analyse est ici limitée à la composition de deux fonctions primitives au maximum. Elle révèle l'existence de huit familles de fonctions triangulaires inversibles :

1. $\mathrm{x}C'$
2. $\mathrm{x} \oplus C$
3. $\mathrm{x}C' + C$
4. $\mathrm{x}C'' \oplus \mathrm{x}$
5. $\mathrm{x} + C$
6. $(\mathrm{x} + C_1) \oplus C_2$
7. $\mathrm{x}C' \oplus C$
8. $(\mathrm{x} \oplus C)C'$

où C, C_1 et C_2 sont des constantes arbitraires, C' un entier impair quelconque, et C'' un entier pair quelconque. Malheureusement ces fonctions sont d'un intérêt fort limité : en effet, soit, comme $\mathrm{x} + C$, elles sont trop linéaires pour être utilisées comme fonction de transition d'un générateur pseudo-aléatoire, soit, comme $\mathrm{x} \oplus C$ elles n'atteignent qu'une période très faible puisqu'en l'occurrence, ici, le générateur oscillerait entre deux valeurs uniquement, ce qui ne correspond pas du tout à ce que l'on recherche pour une fonction de transition d'un générateur pseudo-aléatoire ; au contraire, nous souhaiterions que pour un générateur travaillant sur des mots de n bits, que la période soit de 2^n. Nous sommes donc contraints de nous intéresser à des fonctions plus complexes, résultant de la composition d'au moins trois fonctions primitives.

Construction d'une fonction de période maximale

Il est difficile de déterminer en général l'allure du graphe de transition d'un générateur pseudo-aléatoire disposant d'une fonction d'actualisation quelconque. Dans le cas de la composition de fonctions triangulaires, les cycles sont régis par le lemme suivant :

Lemme 1 *Soit* $f_n(\mathrm{x}) = f(\mathrm{x}) \mod 2^n$ *une fonction triangulaire inversible. Pour chaque cycle de longueur* l *dans* f_{n-1}*, il y a soit deux cycles de longueur* l*, soit un cycle de longueur* $2l$ *dans* f_n*. De plus, si* f *possède un point fixe, alors* $\forall i > 0$*, il y a au moins* 2^{i+1} *points qui appartiennent à un cycle de longueur au plus* 2^i*.*

Ce résultat découle directement du fait que f_n est une fonction triangulaire, et que, par conséquent, $\forall \mathrm{x}, f_n(\mathrm{x}) \mod 2^{n-1} = f_{n-1}(\mathrm{x})$. Il s'ensuit une propriété remarquable :

étant donné que les quatre fonctions possibles pour f_0 sont $f_0(\mathrm{x}) = 0, f_0(\mathrm{x}) = 1, f_0(\mathrm{x}) = \mathrm{x}$ et $f_0(\mathrm{x}) = \mathrm{x} \oplus 1$, et que ces fonctions ont des cycles de taille 1 ou 2, alors *tout cycle d'une fonction triangulaire inversible est de longueur une puissance de 2.*

Fonctions à un seul cycle

Nous allons maintenant nous intéresser à la construction de fonctions triangulaires inversibles n'ayant qu'*un seul cycle*. Nous avons précédemment donné un moyen de déterminer si une fonction triangulaire quelconque est inversible (théorème 4).

Le lemme 1 nous dit que toute fonction triangulaire inversible possède des cycles dont la longueur est nécessairement une puissance de 2. Par conséquent, si l'on note x_i la suite des éléments d'un cycle, rechercher si une fonction triangulaire inversible travaillant sur des mots de taille n est une fonction n'ayant qu'un seul cycle, revient à déterminer si pour un x et un i quelconque, $\mathrm{x}_i \neq \mathrm{x}_{i+2^{n-1}}$. Si cette condition est vérifiée, il est alors possible de dire que le cycle n'est pas de taille 2^{n-1}, ni même de taille inférieure et que donc, la fonction n'a qu'un seul cycle de taille 2^n. Étant donné que pour toute fonction triangulaire, $\mathrm{x}_i = \mathrm{x}_{i+2^{n-1}} \mod 2^{n-1}$, nous obtenons qu'une fonction triangulaire inversible possède un cycle unique si et seulement si $[\mathrm{x}_i]_{n-1} \neq [\mathrm{x}_{i+2^{n-1}}]_{n-1}$. Or, comme la fonction est inversible,

$$[\mathrm{x}_{i+1}]_{n-1} = [f(\mathrm{x}_i)]_{n-1} = [\mathrm{x}_i]_{n-1} \oplus \alpha(\mathrm{x}_i)$$

et donc,

$$[\mathrm{x}_{i+2^{n-1}}]_{n-1} = [\mathrm{x}_i]_{n-1} \oplus \alpha(\mathrm{x}_i) \oplus \alpha(\mathrm{x}_{i+1}) \oplus \alpha(\mathrm{x}_{i+2}) \ldots \oplus \alpha(\mathrm{x}_{i+2^{n-1}-1})$$

ce qui nous amène au théorème suivant :

Théorème 5 *Une T-Fonction décrit un cycle unique modulo 2^n si et seulement si*

$$[f(\mathrm{x})]_{n-1} = [\mathrm{x}]_{n-1} \oplus \alpha([\mathrm{x}]_0, [\mathrm{x}]_1, \ldots, [\mathrm{x}]_{n-2}) \text{ et } \bigoplus_{\mathrm{x}=0}^{2^{n-1}-1} \alpha(\mathrm{x}) = 1.$$

Malheureusement, ce théorème ne nous permet toujours pas de construire facilement une fonction triangulaire inversible n'ayant qu'un seul cycle, mises à part des fonctions du type $x \mapsto x + C$, avec C une constante impaire.

Nous allons donc nous intéresser à un cas particulier.

Définition 13 *[45] Un* paramètre *est une fonction triangulaire dont le calcul du i^e bit du résultat ne dépend pas des i^e bits des variables en entrée.*

Un paramètre est donc une restriction d'une fonction triangulaire. Nous pouvons remarquer qu'une fonction triangulaire r est un paramètre si $\forall \mathrm{x}, r(\mathrm{x}) = r(\mathrm{x} + 2^{n-1}) \mod 2^n$. Il s'ensuit que si l'on considère l'expression modulo 2^{n+1}, les quantités $r(\mathrm{x})$ et $r(\mathrm{x} + 2^{n-1})$ ne vont différer que d'un bit : nous notons $b(\mathrm{x})$ ce bit, défini par $r(\mathrm{x}) = r(\mathrm{x} + 2^{n-1}) + 2^n b(\mathrm{x}) \mod 2^{n+1}$.

Considérons l'expression suivante :

$$B[r,n] = \frac{1}{2^n} \sum_{i=0}^{2^{n-1}-1} (r(i+2^{n-1}) - r(i)) \mod 2 = \bigoplus_{i=0}^{2^{n-1}-1} b(i)$$

Définition 14 *[45] Soit r un paramètre. On dit que ce paramètre est* pair pour n, *si et seulement si $B[r,n] = 0$. De manière analogue, ce paramètre est dit* impair pour n *si et seulement si $B[r,n] = 1$.*

Définition 15 *[45] Soit r un paramètre. Ce paramètre est dit* pair *s'il est pair pour tout $n > 0$, ou* impair *s'il est impair pour tout $n > 0$.*

À titre d'exemple, prenons le paramètre défini par $r(\mathrm{x}) = C$: ce paramètre est clairement pair, car pour tout n, $b(i)$ est constant et, par conséquent, $B[r,n]$ sera pair. Dans le cas d'un paramètre tel que $r(\mathrm{x}) = 2\mathrm{x}$, il est possible de montrer que pour tout $n \geq 2$, r est pair.

Cette notion de paramètre pair ou impair est très intéressante, car elle nous amène au résultat suivant :

Théorème 6 *Soit r un paramètre, et N_0 tels que l'application $\mathrm{x} \mapsto \mathrm{x} + r(\mathrm{x}) \mod 2^{N_0}$ n'a qu'un seul cycle. Si $\forall n > N_0$, le paramètre r est pair, alors l'application $\mathrm{x} \mapsto \mathrm{x} + r(\mathrm{x}) \mod 2^n$ ne possède qu'un seul cycle.*

Preuve : Considérons ce résultat jusqu'à un rang $n \geq N_0$, c'est-à-dire, $\forall i \leq n$, l'application $\mathrm{x} \mapsto \mathrm{x} + r(\mathrm{x}) \mod 2^i$ ne possède qu'un seul cycle. Nous allons maintenant regarder ce qui se passe pour le rang $n+1$. Considérons la suite $\{\mathrm{x}_i\}_{i=0}^{2^{n+1}-1}$ des termes définis par $\mathrm{x}_0 = 0$ et $\mathrm{x}_{i+1} = \mathrm{x}_i + r(\mathrm{x}_i) \mod 2^{n+1}$. Analysons maintenant cette suite :

$$\mathrm{x}_{2^n} \mod 2^{n+1} = \mathrm{x}_{2^n-1} + r(\mathrm{x}_{2^n-1}) \mod 2^{n+1} = \sum_{i=0}^{2^n-1} r(\mathrm{x}_i) \mod 2^{n+1}.$$

Nous allons ici utiliser le fait que, modulo 2^n, la suite $(\mathrm{x}_i)_{i\geq 0}$ est de période 2^n. De ce fait, nous savons que les x_i sont tous différents modulo 2^n et, par conséquent, les ensembles $\{\mathrm{x}_i\}_{i=0}^{2^n-1}$ et $\{i\}_{i=0}^{2^n-1}$ sont les mêmes. De plus, comme r est un paramètre, nous savons que $r(\mathrm{x}) = r(\mathrm{x} + 2^n) \mod 2^{n+1}$. Si nous découpons la somme que nous avons introduite lors du développement de $\mathrm{x}_{2^n} \mod 2^{n+1}$, nous obtenons :

$$\begin{aligned}
\mathrm{x}_{2^n} \mod 2^{n+1} &= \sum_{i=0}^{2^n-1} r(\mathrm{x}_i) \mod 2^{n+1} \\
&= \sum_{i=0}^{2^n-1} r(\mathrm{x}_i \mod 2^n) \mod 2^{n+1} \\
&= \sum_{i=0}^{2^n-1} r(i) \mod 2^{n+1} \\
&= \sum_{i=0}^{2^{n-1}-1} r(i) + \sum_{i=0}^{2^{n-1}-1} r(i+2^{n-1}) \mod 2^{n+1} \qquad (3.2)
\end{aligned}$$

Essayons d'écrire cette somme autrement : nous savons que la suite des $x_i \mod 2^n$ est de période 2^n, ce qui implique $x_0 \neq x_{2^{n-1}}$. Or $x_0 = x_{2^{n-1}} \mod 2^{n-1}$, et comme $x_0 = 0$, nous obtenons $x_{2^{n-1}} = 2^{n-1} \mod 2^n$. La première partie de l'équation 3.2 peut donc s'écrire :

$$\sum_{i=0}^{2^{n-1}-1} r(i) = x_{2^{n-1}} = 2^{n-1} \mod 2^n$$

Ce qui implique que cette somme est de la forme $k2^n + 2^{n-1} \mod 2^{n+1}$, avec k un entier. Comme r est un paramètre pair, il s'ensuit que, modulo 2^{n+1}, la première et la deuxième partie de l'équation 3.2 sont égales, et donc :

$$x_{2^n} = 2(k2^n + 2^{n-1}) = 2^n \mod 2^{n+1}$$

ce qui entraîne que $x_0 \neq x_{2^n} \mod 2^{n+1}$, et par conséquent l'application $x \mapsto x + r(x) \mod 2^{n+1}$ n'a qu'un seul cycle. ◇

Cas de la fonction $x \rightarrow x + (x^2 \vee C)$

Nous allons maintenant nous pencher sur le cas des fonctions triangulaires s'écrivant sous la forme $x \rightarrow x+(x^2 \vee C)$, où C est une constante quelconque. Ce type de fonctions est de complexité légèrement supérieure à celles décrites dans l'analyse exhaustive, puisque résultant de la composition de trois fonctions primitives. Par contre, elle présente l'énorme avantage de ne pas être linéaire, et surtout, de mélanger des opérations arithmétiques, comme l'addition ou l'élévation au carré, avec une opération booléenne. Ce mélange nous assure une certaine complexité lors de l'analyse de la fonction par des méthodes purement algébriques. Malheureusement, cette fonction n'est pas inversible pour tout C, mais il existe une caractérisation simple des constantes C rendant la fonction inversible :

Théorème 7 *Pour tout $n \in \mathbb{N}^*$, la fonction $f(x) = x + (x^2 \vee C) \mod 2^n$ est inversible sur les mots de n bits, si et seulement le bit de poids le plus faible de C est égal à 1.*

Preuve : Nous avons vu précédemment que pour montrer qu'une fonction triangulaire est inversible, il suffit de pouvoir l'exprimer sous la forme d'un réseau de Feistel triangulaire.

Regardons ce que cela nous donne pour le bit de poids faible : $f(x) \mod 2 = x + (x^2 \vee C) \mod 2$. Or nous savons que, pour tout x, $x^2 \mod 2 = x \mod 2$. ce qui nous donne : $[f(x)]_0 = [x]_0 \oplus ([x]_0 \vee [C]_0)$. Dans ce cas, si $[C]_0 = 0$, alors nous obtenons :

$$[f(x)]_0 = [x]_0 \oplus [x]_0 = 0,$$

ce qui n'est pas une fonction inversible. Or si $[C]_0 = 1$, nous obtenons :

$$[f(x)]_0 = [x]_0 + ([x]_0 \vee [1]_0) = [x]_0 \oplus 1,$$

qui est inversible.

Pour ce qui est des bits suivants, la fonction f s'écrit alors sous la forme :

$$\begin{aligned}
[f(\mathrm{x})]_i &= [\mathrm{x}]_i \oplus [\mathrm{x}^2 \vee C]_i \oplus \alpha \\
&= [\mathrm{x}]_i \oplus ([\mathrm{x}^2]_i \vee [C]_i) \oplus \alpha \\
&= [\mathrm{x}]_i \oplus (\beta \vee [c]_i) \oplus \alpha \\
&= [\mathrm{x}]_i \oplus \alpha \oplus \gamma \\
&= [\mathrm{x}]_i \oplus \delta
\end{aligned}$$

Où, α, β, γ et δ sont des fonctions qui ne dépendent que des bits précédant le bit i. Dans ce cas, la fonction $[f(\mathrm{x})]_i$ est inversible, et la fonction $f(\mathrm{x})$ peut être mise sous la forme d'un réseau de Feistel triangulaire, ce qui conclut la preuve. ◇

Il est possible d'obtenir un résultat encore plus intéressant si l'on s'intéresse aux valeurs de C permettant d'obtenir une fonction à un seul cycle. Il apparaît le résultat suivant :

Théorème 8 *La fonction* $f(\mathrm{x}) = \mathrm{x} + (\mathrm{x}^2 \vee C) \mod 2^n$ *sur les mots de n bits n'a qu'un seul cycle de longueur* 2^n*, si et seulement si le bit de poids le plus faible et le troisième bit de poids le plus faible de C sont tous deux égaux à* 1.

Preuve : Concernant l'inversibilité de la fonction, le résultat a déjà été démontré précédemment. Pour ce qui est du cycle unique, il est bien évidemment nécessaire que la fonction soit inversible. Nous allons donc utiliser le théorème 6 pour prouver l'existence d'un cycle unique. Considérons le cas $n = 3$. Dans ce cas, les deux seules constantes vérifiant la condition sont $C = 5$ et $C = 7$ qui s'écrivent respectivement, en base 2, $C = 101$ et $C = 111$. Il est facile de vérifier que $f_5(\mathrm{x}) = \mathrm{x} + (\mathrm{x}^2 \vee 5) \mod 8$ et $f_7(\mathrm{x}) = \mathrm{x} + (\mathrm{x}^2 \vee 7) \mod 8$ sont des fonctions de période 8. Nous pouvons aussi nous en convaincre en nous rappellant que $\forall \mathrm{x}, \mathrm{x}^2 \mod 4 = \mathrm{x} \mod 2$, ce qui signifie que le deuxième bit de poids le plus faible de x^2 est toujours 0 et, par conséquent, $\forall \mathrm{x}$, $(\mathrm{x}^2 \vee 3) \mod 8$ et $(\mathrm{x}^2 \vee 7) \mod 8$ sont des constantes impaires, ce qui implique de manière évidente que f_5 et f_8 sont à cycle unique.

Dans ce cas, il est alors facilement démontrable que $\forall n > 3, \mathrm{x}^2 \vee C \mod 2^n$ est un paramètre pair, et par conséquent, $\forall n \geq 3$, l'application $f(\mathrm{x}) = \mathrm{x} + (\mathrm{x}^2 \vee C)$ est à cycle unique. ◇

Ce type de fonction est particulièrement intéressant : il s'agit de fonctions faciles à calculer en logiciel, et dont les propriétés statistiques, mesurées par les auteurs, sont très bonnes. Les auteurs proposent d'ailleurs l'utilisation de la fonction $f(\mathrm{x}) = \mathrm{x} + (\mathrm{x}^2 \vee C)$ conjointement à la *fonction de filtrage* $F(\mathrm{x}) = \mathrm{x} >> \frac{n}{2}$ comme générateur pseudo-aléatoire [45]. Ainsi, le principe de fonctionnement est le suivant :

- on suppose que l'état interne du générateur, noté x, est un mot de n bits.
- à chaque top d'horloge, le générateur construit un mot de $n/2$ bits concaténé à la suite pseudo-aléatoire, en calculant $g(\mathrm{x})$.
- puis, le contenu de l'état interne est mis à jour : il est remplacé par $f(\mathrm{x}) \mod 2^n$.

Les auteurs préconisent l'emploi de ce générateur en remplacement des LFSRs.

Plusieurs attaques ont été menées contre ce type de générateur. Parmi celles-ci, nous retiendrons la découverte de relations linéaires [54], ou encore la cryptanalyse exposée dans [52]. De nouvelles fonctions de filtrages ont alors été proposées par les auteurs dans [47] : il s'agit de considérer en parallèle, un ensemble de quatre mots mis à jour par quatre fonctions triangulaires, puis de les combiner par une dernière fonction triangulaire afin de former la suite pseudo-aléatoire. Une attaque a été publiée concernant ce générateur [74].

Je vais maintenant présenter l'attaque que j'ai mis au point concernant ce générateur, plus particulièrement contre la fonction de filtrage initialement proposée.

3.3 Contexte de l'attaque, définitions et faits

Notre but est donc de retrouver l'état interne du générateur dans le cas particulier où la constante C de la fonction de transition f vérifie $\log_2 C \leq n/4$, en observant en moyenne $O(2^{n/4})$ sorties consécutives du générateur. Dans leurs articles, les auteurs proposent de prendre $C = 5$, qui vérifie cette condition pour tout $n \geq 12$.

Nous allons nous intéresser à des états particuliers susceptibles de nous donner de l'information : regardons ce qu'il se passe lorsque la moitié des bits de poids fort de la moitié des bits de poids faible de l'état interne sont à zéro, en d'autres termes, lorsque :

$$[x_t]_{(\frac{n}{2}-1)\ldots\frac{n}{4}} = 0 \text{ ou encore } x_t \mod 2^{n/2} < 2^{n/4}. \tag{3.3}$$

Nous allons introduire deux notations permettant de séparer clairement les deux morceaux de l'état interne du générateur : nous notons

$$u_t = [x_t]_{(n-1)\ldots\frac{n}{2}}, \text{ et } v_t = [x_t]_{(\frac{n}{2}-1)\ldots 0}$$

Ainsi, la suite $(u_t)_{t\geq 0}$ correspond à la suite des sorties du générateur, suite connue de l'attaquant. La suite $(v_t)_{t\geq 0}$, par contre, lui est inconnue.

Nous pouvons introduire quelques lemmes simples :

Lemme 2 $\forall x, y \in \mathbb{N}$, *nous avons la relation* $\max(x, y) \leq x \vee y \leq x + y$.

Preuve : La première partie, $\max(x, y) \leq x \vee y$ est triviale, puisque nécessairement $x \leq x \vee y$ et $y \leq x \vee y$. Pour ce qui concerne la deuxième inégalité, nous allons la prouver en effectuant un raisonnement par récurrence sur la taille $L \in \mathbb{N}$ de la plus courte représentation binaire de x et y.

Si $L = 0$, cela signifie que x et y sont tous deux égaux à 0, et le résultat est immédiatement vérifié.

Supposons maintenant que l'inégalité soit vérifiée jusqu'à une taille k. Supposons que la taille de la plus courte représentation binaire de x et de y soit $L+1$. Deux cas peuvent se produire en fonction de l'apparition d'une retenue lors du calcul de $x + y$.

1. aucune retenue n'est engendrée par le calcul de $x + y$, alors $x \vee y = x + y$.

2. une retenue apparaît au cours du calcul ; supposons que cette retenue apparaisse lors du calcul du bit de rang i du résultat, avec $0 \leq i \leq L$. Par conséquent, x et y peuvent s'écrire sous la forme :

$$\begin{aligned} x &= 2^i x_h + 2^{i-1} + x_l \\ y &= 2^i y_h + 2^{i-1} + y_l \end{aligned}$$

avec $x_l, y_l < 2^{i-1}$. Ceci qui signifie

$$\begin{aligned} x + y &= 2^{i-1}(2x_h + 2y_h + 1) + (x_l + y_l) \\ &\geq 2^{i-1}(2(x_h + y_h) + 1) + (x_l \vee y_l) \\ &\geq 2^i(x_h \vee y_h) + (x_l \vee y_l) + 2^{i-1} \\ &\geq (x_h \vee y_h)2^i + (x_l \vee y_l) \\ &\geq x \vee y \end{aligned}$$

ce qui conclut la preuve.

◇

Lemme 3 *Soit* $x = u2^{n/2} + v$, *où* $u < 2^{n/2}$ *et* $v < 2^{n/4}$. *Soit* $x' = f(x) \mod 2^n = u'2^{n/2} + v'$ *où* $u' < 2^{n/2}$ *et* $v' < 2^{n/2}$. *Alors nous avons*

$$u' = u(2v + 1) \mod 2^{n/2}. \tag{3.4}$$

Preuve : Grâce au lemme 2 et au fait que $v < 2^{n/4}$, nous pouvons écrire :

$$v + (v^2 \vee C) \leq 2^{n/2} - 2^{n/4} + C$$

et comme $\log_2 C \leq n/4$, nous obtenons :

$$v + (v^2 \vee C) < 2^{n/2}.$$

Ainsi, nous savons que si $v < 2^{n/4}$, alors f(x) s'écrit avec au plus $n/2$ bits, ce que nous pouvons synthétiser grâce au schéma suivant :

x =	u	0	v
$x^2 \vee C$ =	$2uv$	$v^2 \vee C$	
$x + (x^2 \vee C)$ =	$u(2v+1)$	$v + (v^2 \vee C)$	

et nous obtenons bien l'égalité recherchée. ◇

Lemme 4 *Soient* x *et* x' *deux entiers définis comme dans le lemme 3 ; alors* u *et* u' *sont divisibles par la même puissance de 2.*

Preuve : Soit k le nombre de zéros à la fin de la représentation binaire de u. Alors $u = 2^k\tilde{u}$, où $\tilde{u}$ est un entier impair. Comme $u' = 2^k\tilde{u}(2v + 1) \mod 2^{n/2}$ et que $2v + 1$ est lui aussi un entier impair, le produit $\tilde{u}(2v + 1)$ est impair, et u' a donc exactement k zéros à la fin de sa représentation binaire. ◇

3.4 L'attaque

Nous allons maintenant nous pencher sur la réalisation effective de l'attaque. Nous avons vu, grâce au lemme 3 que si l'état interne vérifiait certains critères, il était alors possible de le retrouver en intégralité. Le lemme 4 nous donne quant à lui une condition nécessaire sur les sorties du générateur afin de déterminer à partir de deux sorties consécutives, si l'état interne est susceptible de vérifier les conditions du lemme 3.

L'attaque se déroulera donc ainsi : nous tenterons de résoudre l'équation 3.4 en v pour tous les couples de sorties consécutives (avec $u = u_t$ et $u' = u_{t+1}$) du générateur divisibles par la même puissance de 2. Si une solution existe, elle peut être trouvée grâce à l'*algorithme d'Euclide étendu*. Nous pouvons d'ores et déjà remarquer ici que si le résultat calculé par l'algorithme d'Euclide est supérieur à $2^{n/4}$, alors cette incohérence nous permet de savoir que nous n'étions pas dans les conditions requises par le lemme 3. Par contre, si ce résultat est inférieur à $2^{n/4}$, nous ne pourrons pas affirmer que nous avons réussi à retrouver l'intégralité de l'état interne du registre : il faudra encore le vérifier.

Il s'agit donc de trouver un couple (a, b) qui vérifie :

$$2^{n/2}a + u_t b = 2^s, \text{ où } 2^s = \gcd(2^{n/2}, u_t).$$

Une fois ce couple calculé, nous avons :

$$v_t = \frac{bu_{t+1} - 2^s}{2^{s+1}} \mod 2^{\frac{n}{2}-s}.$$

Il est malheureusement possible d'avoir plusieurs solutions à la première équation quand $s \geq n/4$: dans ce cas, nous ne considérerons que la plus petite valeur positive afin que la consommation mémoire de notre algorithme reste linéaire.

À ce stade, le principal problème de l'attaquant est de savoir si la quantité qu'il vient de calculer pour v correspond à la réalité de l'état interne. Afin de le vérifier, il va construire un nouveau générateur en tout point semblable au générateur attaqué, et il va placer la quantité $2^{n/2}u_t + v$ en guise d'état initial. Il s'agit donc maintenant de vérifier qu'il n'y a pas de divergence entre les sorties de ce nouveau générateur et celles du générateur attaqué.

Notons c_t l'état calculé par l'attaquant à partir de la connaissance de u_t et de u_{t+1}. Nous avons alors $c_t = x_t + \varepsilon$, où ε représente l'erreur de l'attaquant.

Si $\varepsilon = 0$, alors l'attaque est réussie ; dans le cas contraire, nous allons estimer la divergence que ce biais va introduire dans la génération du pseudo-aléa du nouveau générateur.

Regardons de plus près l'évolution des états internes des deux générateurs. Après un top d'horloge, l'état interne du générateur attaqué sera :

$$x_{t+1} = x_t + (x_t^2 \vee C)$$

alors que celui du nouveau générateur sera :

$$c_{t+1} = c_t + (c_t^2 \vee C) = x_t + \varepsilon + [(x_t^2 + \varepsilon^2 + 2\varepsilon x_t) \vee C]$$

ce qui est approximativement équivalent à :

$$c_{t+1} \approx x_t + (x_t^2 \vee C) + E(\varepsilon) = x_{t+1} + E(\varepsilon)$$

où $E(\varepsilon)$ est un polynôme de degré 2 en ε.

L'erreur évolue donc de manière quadratique, ce qui signifie que sa taille double à chaque top d'horloge : ainsi, les effets de cette erreur seront perceptibles au niveau de la sortie du nouveau générateur dès qu'elle viendra empiéter sur la moitié des bits de poids fort de l'état interne, ce qui se produit après au plus $log_2(n/2)$ itérations. Ainsi, une méthode efficace permettant de déterminer pour l'attaquant s'il a obtenu le bon état interne est de comparer $log_2(n/2)$ sorties du générateur attaqué avec les sorties de son propre générateur.

Il est intéressant de savoir à partir de combien de sorties nous pouvons atteindre un état qui correspond aux critères recherchés. Grâce aux propriétés des T-Fonctions, nous savons que la distance entre un x_t quelconque et un état vérifiant les conditions recherchées est en moyenne de $2^{n/4}$. De plus, nous savons qu'il existe $2^{3n/4}$ états vérifiant ces conditions.

3.5 Un exemple d'attaque

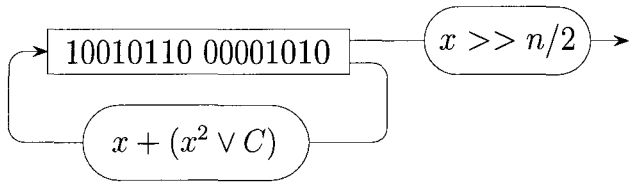
Afin de clarifier les choses, nous allons voir le déroulement d'une attaque sur un générateur de petite taille.

Considérons un générateur de taille $n = 16$ bits : la première étape consiste à trouver une paire de sorties vérifiant la condition de divisibilité exposée plus haut, ce qui signifie que nous allons observer les sorties du générateur pseudo-aléatoire jusqu'à ce que deux sorties consécutives aient le même nombre de zéros à la fin de leur représentations binaire.

Partie haute (sorties)	Partie basse (inconnues)
01011101	01010000
10010110	01010101
01001110	10010010
10011001	11010111
01001100	01101100
10011010	00000001
11001110	00000110
⋮	⋮

FIG. 3.1 – Première attaque

Nous avons exposé dans la colonne de gauche de la figure 3.1 les sorties consécutives du générateur. Le premier couple ayant le même nombre de zéros à la fin de leur représentations binaire sont les mots 10010110 et 01001110. L'attaquant va alors supposer



Sorties réelles : 01001110, 10011001, 01001100, ...
Générateur simulé : 01001110, 00100010, 11001000, ...

FIG. 3.2 – Attaque ratée - vérification

qu'au moment où la sortie du générateur était 10010110, l'état interne vérifiait l'équation 3.3. L'exemple à été volontairement choisi de telle sorte que cette hypothèse soit fausse ; l'attaquant va alors résoudre le système suivant :

$$\begin{aligned} u' &= u(2v+1) \mod 2^{n/2} \\ 78 &= 150(2v+1) \mod 2^8 \\ v &= 10 \end{aligned}$$

Le résultat calculé vérifie bien $v \leq 2^{n/4}$, ce qui signifie qu'il s'agit peut être d'un attaque réussie : pour s'en convaincre, l'attaquant doit alors construire un générateur dont l'état interne sera $2^8 \times 150 + 10 = 38410$, soit en binaire 1001011000001010, et vérifier si les sorties de ce générateur concordent avec celles du véritable générateur. Puisque nous savons que si les états internes ne sont pas identiques, les sorties vont diverger à partir d'au plus $log_2(n/2) = 3$ sorties, nous allons comparer trois sorties consécutives des deux générateurs ; nous pouvons ici observer une divergence dès la deuxième sortie, ce qui nous indique clairement que l'etat calculé n'est pas le bon. Nous pouvons d'ailleurs préciser au passage que, malgré le fait que la théorie prévoit une divergence au bout d'au plus $log_2(n/2)$ sorties, la divergence à lieu en pratique très rapidement, au bout de deux à trois sorties en pratique.

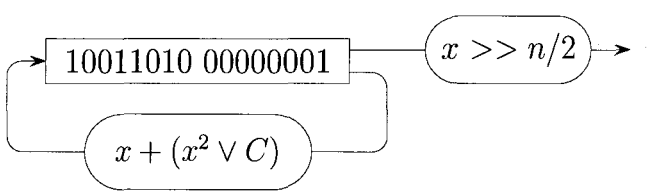
L'attaquant va alors maintenant rechercher un nouveau couple vérifiant le critère de divisibilité, et recommencer les étapes précédentes. Le couple suivant est celui de la figure 3.3. Le nouveau système à résoudre est donc :

$$\begin{aligned} u' &= u(2v+1) \mod 2^{n/2} \\ 206 &= 154(2v+1) \mod 2^8 \\ v &= 1 \end{aligned}$$

A nouveau, comme la valeur de v calculée est inférieure à $2^{n/4}$, l'attaquant procède à une vérification en construisant un registre dont l'état interne est 100110100000001 et en comparant trois sorties consécutives du générateur attaqué avec trois sorties de ce générateur.

Partie haute (sorties)	Partie basse (inconnues)
01011101	01010000
10010110	01010101
01001110	10010010
10011001	11010111
01001100	01101100
10011010	00000001
11001110	00000110
⋮	⋮

FIG. 3.3 – Deuxième attaque



Sorties réelles : 11001110, 01110110, 00100001, ...
Générateur simulé : 11001110, 01110110, 00100001, ...

FIG. 3.4 – Deuxième attaque - vérification

Cette fois-ci, comme nous pouvons le voir sur la figure 3.4, les deux suites sont identiques sur une longueur d'au moins $\log_2(n/2) = 3$ sorties consécutives : nous sommes donc certains d'avoir calculé la bonne valeur de l'état interne.

3.6 L'algorithme

Il est possible de synthétiser cette attaque en écrivant un algorithme en pseudo-code tel que celui-ci :

```
crack(u_t, u_{t+1}) :=
  s := log_2(gcd(u_t, 2^{n/2}))
  s' := log_2(gcd(u_{t+1}, 2^{n/2}))
  if u_t ≠ 0 and s = s' then
    trouver (a, b) ∈ ℕ × ℕ*, une solution à 2^{n/2}a + u_t b = 2^s
    v_t := (b u_{t+1} − 2^s) / 2^{s+1}  mod 2^{n/2 − s}
    x_t := v_t + 2^{n/2} u_t
    si v_t < 2^{n/4} alors renvoyer x_t sinon renvoyer faux
  sinon
    renvoyer faux
```

```
attack(u) :=
  n :=la longueur de l'état interne
  l := log_2 n
  t := 0
  seeds   := {faux, faux, ..., faux}  (une liste de l éléments)
  success := {0, 0, ..., 0}          (une liste de l éléments)
  faire
    seeds_{0...(l−2)}   := seeds_{1...(l−1)}
    success_{0...(l−2)} := success_{1...(l−1)}
    seeds_{l−1}         := crack(u_t, u_{t+1})
    success_{l−1}       := 0
   pour tous les éléments seeds_j de seeds faire
      si seeds_j ≠ faux alors seeds_j := f(seeds_j)
      si seeds_j = u_{t+1} alors success_j := success_j + 1
    si success_0 = l alors l'attaque est un succes !
    t := t + 1
  boucler jusqu'au succes
```

3.7 Améliorations possibles

Nous avons vu précédemment que cette attaque avait une complexité en temps de $O(2^{n/4})$. Cette complexité mesure le nombre moyen de sorties nécessaire à la reconstruction de l'état interne. En fait, il serait intéressant de déterminer comment reconstruire cet état à partir du plus petit nombre de sorties possible. De ce point de vue, il est possible d'améliorer notre attaque en considérant les sorties dont la forme est la suivante, ce qui n'impose plus la condition sur la divisibilité :

$\leftarrow s \rightarrow$

$$u_t = \boxed{\ldots 1 \mid 0}$$
$$u_{t+1} = \boxed{\ldots 1 \mid \ldots}$$

avec l'hypothèse suivante sur l'état interne :

$$[x_t]_{(\frac{n}{2}-1)\ldots\frac{n}{4}+\frac{s}{2}} = 0 \Longleftrightarrow x_t \mod 2^{n/2} < 2^{n/4+s/2}$$

où $s = \log_2(\gcd(u_t, 2^{n/2}))$ est supposé être pair.

Ceci constitue une extension des conditions imposées par l'attaque : il s'agit en fait de considérer les mots comme étant une partie d'un mot de $n+2s$ bits, où les $2s$ bits de poids le plus fort seraient ignorés. Nous allons tenter d'appliquer notre attaque dans ce contexte en considérant maintenant que les sorties du générateur sont des sorties tronquées d'un générateur de taille $n + 2s$. Nous obtenons :

$$\widetilde{u}_t = \frac{u_t}{2^s}, \qquad \widetilde{u}_{t+1} = \frac{u_{t+1} - (u_{t+1} \mod 2^s)}{2^s}$$

ce qui nous amène à résoudre l'équation :

$$\widetilde{u}_t(2\widetilde{v}_t + 1) = \widetilde{u}_{t+1} \mod 2^{\frac{n}{2}-s}$$

afin de trouver une solution en $\widetilde{v}_t$ telle que $x_t \mod 2^{n/2} < 2^{n/4+s/2}$, avec $x_t = 2^{n/2+s}\widetilde{u}_t + \widetilde{v}_t$. Il peut alors y avoir plusieurs solutions qui vérifient l'équation.

Il est clair que cette attaque nécessite plus de calculs puisque la moitié des couples $(\widetilde{u}_t, \widetilde{u}_{t+1})$ vérifie les conditions de départ. Jetons un œil à la complexité en temps de cette nouvelle attaque : le nombre d'états internes qui sont susceptibles d'être utilisés est, pour un s fixé :

$$s = 0 : \boxed{\ldots \mid \ldots \mid 0 \mid \ldots}$$
$$s = 2 : \boxed{\ldots \mid \ldots \mid 0 \mid 0 \mid 0 \mid 1 \mid \ldots}$$
$$s = 4 : \boxed{\ldots \mid \ldots \mid 0 \mid 0 \mid 0 \mid 0 \mid 0 \mid 1 \mid . \mid \ldots}$$
$$\vdots \qquad\qquad \vdots$$
$$s = n/2 : \boxed{0 \mid 0 \mid 1 \mid \ldots \mid \ldots}$$

donc, le nombre total d'états est :

$$2^{\frac{3n}{4}} + \sum_{\tilde{s}=1}^{n/4} 2^{\frac{3n}{4}-(\tilde{s}+1)} = 2^{\frac{3n}{4}} + 2^{\frac{3n}{4}-1} - 2^{\frac{n}{2}-1}.$$

donc, asymptotiquement, le nombre d'état internes qui peuvent être utilisés dans cette nouvelle version est de 50% supérieur à l'attaque précédente.

3.8 Conclusion

Ces deux attaques ne permettent en aucun cas de briser complétement l'intérêt des T-fonctions, mais montrent plutôt que le choix de la fonction de sortie du générateur doit être effectué avec soin. En effet, l'argument principal exposé par les auteurs de ce générateur est la simplicité d'implémentation en logiciel, sa grande vitesse et sa grande non linéarité, dues à la combinaison de fonctions arithmétiques et booléennes. Cette vitesse est due au fait que dans le cas de la fonction $f(x) = x + (x^2 \vee 5) \mod 2^n$, toutes les primitives correspondent à une instruction processeur et que, par conséquent, il est possible d'appliquer cette fonction sur des mots machine en obtenant de très grandes performances.

Or, si l'on cherche à implémanter ce générateur sur des mots machine des processeurs actuels, nous sommes contraints de prendre $n = 32$, voire au mieux $n = 64$ sur certaines architectures. Dans ce cas, la première version de notre attaque n'aura besoin en moyenne que de $2^{n/4} = 256$ sorties consécutives sur une machine 32 bits, ou $2^{16} = 65536$ sorties consécutives sur une machine 64 bits. De plus comme l'attaque ne comporte aucun calcul complexe, elle est particulièrement intéressante dans ce cas de figure! Nous pouvons réussir à retrouver l'état interne d'un registre de 128 bits de ce type sur un processeur actuel en quelques minutes.

En conclusion, malgré le fait que la motivation principale de ce générateur soit la vitesse de production de l'aléa, je préconise l'emploi d'une fonction de sortie plus complexe que celle proposée par les auteurs. On peut estimer que la complexité de l'attaque est multipliée par deux à chaque fois que l'on retire un bit supplémentaire à la fonction de sortie. Ainsi, dans le cas extrême, la sécurité la plus importante serait obtenue avec une fonction de sortie $g(x) = \frac{x}{2^{n-1}}$ qui ne sort donc que le bit de poids fort de l'état interne à chaque itération ; il est par contre bien évident que la vitesse du générateur est alors divisée par $2^{n/2}$.

Chapitre 4

Les suites de De Bruijn : présentation et constructions

4.1 Introduction

Les *suites de De Bruijn* sont de grand intérêt en cryptographie, car elle possèdent un grand nombre des caractéristiques requises pour une utilisation cryptographique. Nous allons nous intéresser dans ce chapitre à ces caractéristiques, ainsi qu'aux différentes méthodes de construction possibles.

4.2 Définitions

On définit les *suites de De Bruijn* de la manière suivante :

Définition 16 *Une* suite de De Bruijn *d'ordre $n \in \mathbb{N}$ sur un alphabet $\mathcal{A}$ est une suite de période $card(\mathcal{A})^n$ telle que tous les motifs de longueur n n'apparaissent qu'une seule fois.*

Ce type de suites présente des caractéristiques très intéressantes d'un point de vue statistique ; il est facile de voir que si tous les motifs de longueur n n'apparaissent qu'une seule fois, alors les motifs d'une longueur inférieure apparaissent tous avec la même fréquence. D'une manière générale, tous les motifs de longueur $k \leq n$ apparaissent exactement 2^{n-k} fois.

Nous nous intéresserons plus particulièrement dans notre cas aux suites définies sur l'alphabet binaire $\mathcal{A} = \{0, 1\} = \mathbb{F}_2$.

La construction de ce type de suite n'est pas particulièrement aisée ; il s'agit d'un problème combinatoire non trivial, car construire une suite de De Bruijn est équivalent à rechercher un chemin *Hamiltonien* dans un graphe particulier, appelé *graphe de De Bruijn.*

Définition 17 *Un* graphe de De Bruijn d'ordre n *sur un alphabet $\mathcal{A}$ est un graphe composé de $card(\mathcal{A})^n$ sommets, étiquetés par les mots de n caractères que l'on peut former à partir des lettres de $\mathcal{A}$, et de $card(\mathcal{A})^{n+1}$ flèches. Il existe une flèche allant d'un nœud x*

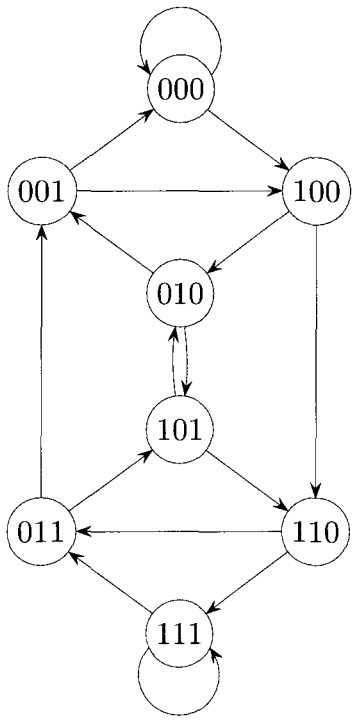


FIG. 4.1 – Graphe de De Bruijn d'ordre 3 sur l'alphabet $\{0, 1\}$.

vers un nœud y si et seulement si les $n-1$ premiers caractères de x sont identiques aux $n-1$ derniers caractères de y.

A titre d'exemple, la figure 4.1 représente un graphe de De Bruijn d'ordre 3 pour l'alphabet $\{0, 1\}$: ce graphe comporte donc huit nœuds et seize flèches. Ceci implique qu'il y a donc toujours $card(\mathcal{A})$ flèches sortant d'un nœud quelconque.

Grâce à cette relation qui lie deux nœuds, on peut voir le parallèle qui existe entre un graphe de De Bruijn et le graphe de transition d'un registre à décalage (voir chapitre 1) : il existe une flèche dans un graphe de De Bruijn, allant d'un nœud x vers un nœud y, si et seulement si il est possible de construire un registre à décalage qui fait succéder l'état y à l'état x.

Si l'on regarde à nouveau le graphe représenté à la figure 4.1, chaque nœud est étiqueté par un mot de trois bits. En considérant par exemple l'état 110, deux flèches sortent de cet état, pour arriver aux états 011 et 111 qui sont deux successeurs possibles de l'état 110 dans le graphe de transition d'un registre à décalage. De même, les deux flèches entrantes de l'état 110 proviennent des états 101 et 100 qui sont bien les successeurs possibles de l'état 110 dans le graphe de transition d'un registre à décalage.

Nous noterons dans la suite G_n le graphe de De Bruijn d'ordre n sur l'alphabet $\{0, 1\}$.

Dans toute la suite, nous nous concentrerons essentiellement sur la construction de suites de De Bruijn pour l'alphabet $\{0, 1\}$. Il a été montré dans [49], puis dans [21] qu'il était possible de dénombrer le nombre de suites de De Bruijn :

Théorème 9 *Il existe exactement* $2^{2^{n-1}-n}$ *suite de De Bruijn d'ordre* n.

4.3 Prefer one

Une méthode simple de construction de suites de De Bruijn est la méthode appelée *prefer one*. Le principe de cette méthode est très simple, et nous allons l'illustrer par un exemple.

Considérons que nous voulons construire une suite (s_n) d'ordre $n = 5$. La construction de la suite est itérative : nous allons construire un mot w caractère après caractère, correspondant à une période de la suite (s_n). La première étape consiste à créer un motif constitué du caractère 0 répété n fois.

$$w = 00000.$$

La construction consiste maintenant à concaténer à w le caractère 0 ou le caractère 1 en considérant le fait que lorsque l'on ajoute un caractère, il faut que le motif formé des n derniers bits du mot formé ne soit jamais apparu auparavant dans le mot. Quand le choix est possible, c'est-à-dire, quand le motif constitué des $n - 1$ derniers caractères de la suite concaténés au caractère 0 ou au caractère 1 n'est jamais apparu, on choisit toujours le caractère 1.

Ainsi, dans notre exemple, nous n'avons pas le choix. Le mot étant actuellement 00000, nous ne pouvons pas choisir le caractère 0 puisque le motif des 5 derniers bits du mot qui serait ainsi formée apparaîtrait déjà précédemment dans la suite. Nous ajoutons donc le caractère 1, et le mot devient :

$$w = 000001\ldots$$

Cette fois ci, nous pouvons remarquer que si nous concaténons le caractère 0 ou le caractère 1, le nouveau motif (00010 ou 00011) n'apparaît nul part ailleurs : puisque nous avons le choix, nous choisissons le caractère 1.

$$w = 0000011\ldots$$

et ainsi de suite :

$$w = 00000111110111001101011000101001$$

Nous nous arrêtons dès que nous avons $2^5 = 32$ caractères. Une période de la suite (s_n) est alors constitué des bits du mot w.

$$(s_n) = w, w, w, \ldots$$

Cette méthode est d'autant plus intéressante qu'elle n'est pas réduite à la construction d'une seule suite de De Bruijn d'ordre n : en effet, maintenant que nous avons construit une suite, il est possible de construire une autre suite de même ordre, en revenant au dernier endroit où l'on a eu un choix possible entre 0 et 1, et en optant pour l'autre choix. Ainsi il est possible de construire l'intégralité des suites de De Bruijn d'ordre n grâce à cette méthode simple. Il est par contre évident qu'en pratique cette méthode est limitée à des valeurs de n très petites.

4.4 Construction à partir d'un LFSR

Il est connu que la période maximale atteignable par un LFSR de taille n est de $2^n - 1$, période que l'on atteint en utilisant une fonction de rétroaction primitive (voir chapitre 1). L'idée est la suivante : puisque dans la suite produite par un LFSR de période maximale de longueur n, tous les mots de longueur n apparaissent une seule fois hormis le mot $0000\ldots0$, il suffirait d'ajouter ce mot manquant, en fusionnant les deux cycles du graphe de transition (un cycle de longueur 1 et un autre de longueur $2^n - 1$) pour en obtenir un unique de longueur 2^n.

Pour ce faire, nous allons modifier la table de vérité de la fonction de rétroaction f du LFSR afin d'ajouter l'état $(0, 0, \ldots 0)$ au cycle de longueur $2^n - 1$: le principe sera donc de modifier le successeur et le prédécesseur de cet état afin d'atteindre la période de 2^n.

Pour cela, nous modifions la table de vérité de f pour que le successeur de $(0, 0, \ldots, 0)$ devienne $(1, 0, \ldots, 0)$ et que le successeur de $(0, 0, \ldots 0, 1)$ devienne $(0, 0, \ldots 0)$.

Soit f la fonction de rétroaction d'un LFSR de période maximale de taille n, et soit f' la fonction définie par :

$$f'(x_{n-1}, x_{n-2}, \ldots, x_0) = f(x_{n-1}, x_{n-2}, \ldots, x_0) \oplus \prod_{i=1}^{n-1} \overline{x_i}$$

Alors le FSR de taille n ayant pour fonction de rétroaction la fonction f' est un FSR de période maximale 2^n générant une suite de De Bruijn.

Le principal intérêt de cette méthode est la simplicité de la construction de la fonction de rétroaction, et dans le nombre relativement important de suites d'ordre n qu'il est possible de construire. En effet, nous savons qu'il existe $\phi(2^n - 1)/n$ polynômes primitifs distincts [27], et donc autant de LFSR différents de période maximale et de suites de De Bruijn que l'on peut construire ainsi. Malheureusement, cette suite est en grosse majorité une suite linéaire, puisque identique à la suite produite par un registre à décalage à rétroaction linéaire. Par conséquent ce registre sera très vulnérable à l'attaque de Berlekamp-Massey [4, 50] (voir chapitre 2). Nous perdons ici l'intérêt principal des suites de De Bruijn en général.

4.5 Construction à partir de PCR

4.5.1 Définition d'un PCR

Un autre type de construction intéressante repose sur la fusion de cycles d'objets simple. Nous allons nous pencher sur le cas des *PCR*, ou *Pure Circulating Register.*

Le principe de ces registres est très simple : il s'agit d'un registre ayant pour fonction de rétroaction la fonction :

$$f(x_{n-1}, x_{n-2}, \ldots, x_0) = x_0$$

Ce registre va uniquement faire *tourner* l'état interne, c'est-à-dire que le bit sortant du registre à chaque top d'horloge est réinjecté en tant que bit entrant dans l'état interne, d'où le nom de PCR. Il est donc évident que ce type de registre comporte un grand nombre de petits cycles, dont la taille maximale est n.

Pour être plus précis, il est possible de déterminer le nombre exact de cycles que contient le graphe de transition d'un tel registre [35].

Son dénombrement repose sur le théorème suivant (théorème de Burnside-Frobenius) :

Théorème 10 *Soit S un ensemble, et G un groupe opérant sur S. Le nombre d'orbites dans lequel G décompose S est donné par :*

$$\frac{1}{\gamma} \sum_{\sigma \in G} I(\sigma)$$

où γ est l'ordre de G, et $I(\sigma)$ le nombre de points fixes, c'est-à-dire d'éléments $x \in S$ tel que $\sigma(x) = x$.

Considérons S l'ensemble des mots binaires de longueur n, et $G = \{\sigma, \sigma^2, \ldots \sigma^n\}$ le groupe engendré par σ, où σ est l'opérateur de rotation d'un mot binaire de longueur n, défini par :

$$\begin{array}{rcl} \sigma : \mathbb{F}_2^n & \rightarrow & \mathbb{F}_2^n \\ (x_{n-1}, x_{n-2}, \ldots x_1, x_0) & \mapsto & (x_{n-2}, x_{n-3}, \ldots, x_1, x_0, x_{n-1}) \end{array}$$

Il nous faut déterminer le nombre de points fixes de S pour chacune des puissances de σ. Pour cela, il suffit de voir que si x est un point fixe, alors :

$$\begin{array}{rcl} \mathrm{x} & = & \sigma^i(\mathrm{x}) \\ (x_{n-1}, x_{n-2}, \ldots, x_1, x_0) & = & (x_{i-1}, x_{i-2}, \ldots, x_1, x_0, x_{n-1}, \ldots x_{i+1}, x_i) \end{array}$$

Remarquons alors que si i ne divise pas n, alors x n'est pas un point fixe. Il nous faut donc considérer les vecteurs qui sont eux-mêmes un motif répété un nombre entier de fois, jusqu'à tomber sur une longueur qui soit un diviseur de n. Cet entier est donc par conséquent le plus grand diviseur commun de n et de i.

Nous obtenons donc d'une manière générale que le nombre de points fixes de S pour σ_i est donné par :

$$I(\sigma_i) = 2^{(n,i)} \tag{4.1}$$

où (n, i) représente le plus grand diviseur commun de n et de i.

Ainsi, le nombre de décompositions de S par G est donné par :

$$Z(n) = \frac{1}{n} \sum_{i=1}^{n} 2^{(n,i)}$$

Nous pouvons réécrire cette formule de manière plus élégante, en regroupant les termes différemment :

$$\begin{aligned} Z(n) &= \frac{1}{n} \sum_{d|n} \sum_{(n,i)=d, i \leq n} 2^d \\ &= \frac{1}{n} \sum_{d|n} 2^d \sum_{(n,i)=d, i \leq n} 1 \\ &= \frac{1}{n} \sum_{d|n} \phi(\frac{n}{d}) 2^d \\ &= \frac{1}{n} \sum_{d|n} \phi(d) 2^{n/d} \end{aligned}$$

où $\phi(d)$ représente la *fonction d'Euler* définie comme étant le nombre d'entiers dans l'intervalle $[1, d-1]$ premiers avec d.

Le nombre total de cycles $Z(n)$ d'un PCR de taille n est donc donné par

$$Z(n) = \frac{1}{n} \sum_{d|n} \phi(d) 2^{n/d}.$$

4.5.2 Construction de suites de De Bruijn

Pour obtenir une suite de De Bruijn à partir d'un PCR, nous allons fusionner les $Z(n)$ cycles afin d'en obtenir un unique. Dans ce but, nous allons utiliser le théorème 2, et modifier $Z(n)$ entrées dans la table de vérité de g. Pour chacun des cycles du PCR, nous choisissons l'état comportant le plus grand nombre de zéros à droite de sa représentation vectorielle. Si l'on note $(x_{n-1}, x_{n-2}, \ldots, x_2, x_1, x_0)$ un tel état, nous modifions dans la table de vérité de g l'entrée $(x_{n-1}, x_{n-2}, \ldots, x_2, x_1)$.

À titre d'exemple, voyons ce qui se passe pour $n = 4$. Un PCR ayant cette taille a un graphe de transition comportant $Z(4) = 6$ cycles (figure 4.2).

La première fusion évidente que nous allons opérer est la fusion du cycle (0000) et du cycle (1000, 0100, 0010, 0001). Nous considérons que le FSR est non singulier, et que sa fonction de rétroaction est donc de la forme $f(x_{n-1}, \ldots, x_0) = x_0 \oplus g(x_{n-1}, \ldots, x_1)$,

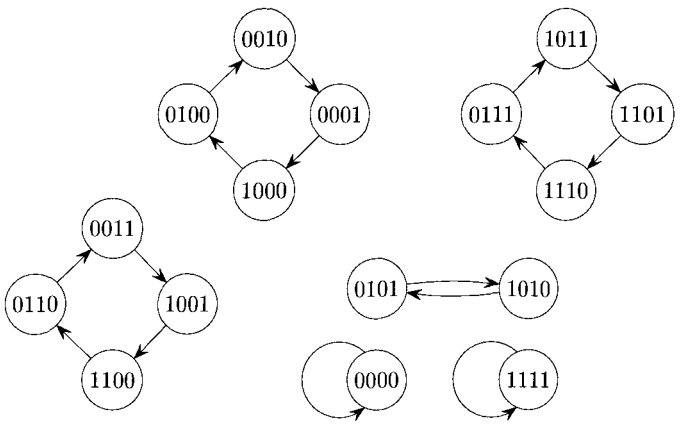


FIG. 4.2 – Cycles d'un PCR de taille 4

ce qui est une condition nécessaire à l'obtention d'un graphe de transition où les cycles n'ont pas de pré-période (voir section 1.3.3). Nous allons modifier la table de vérité de g pour l'entrée $(0,0,0)$, ce qui aura pour conséquence de modifier les successeurs des états $(0,0,0,0)$ et $(1,0,0,0)$, pour obtenir le cycle :

$$(0000, 1000, 0100, 0010, 0001).$$

Pour ce qui concerne le cycle $(1100, 0110, 0011, 1001)$, l'état ayant le plus de zéros à droite est l'état 1100. Ainsi, nous modifions la table de vérité de g pour l'entrée $(1,1,0)$, ce qui modifie les successeurs des états $(1,1,0,0)$ et $(1,1,0,1)$, et nous obtenons le cycle :

$$(0000, 1000, 1100, 0110, 0011, 1001, 0100, 0010, 0001)$$

et ainsi de suite ...

Cette méthode de construction nous amène à un résultat intéressant, puisqu'elle nous permet de construire une fonction de rétroaction de poids minimal pour la construction d'une suite de De Bruijn. En effet, en partant d'une table de vérité pour g complètement vide, et en fusionnant les cycles obtenus, nous obtenons une table de vérité de poids $Z(n)$. Pour chaque ordre, cette méthode n'est capable de construire qu'une unique suite.

4.6 Construction à partir de CCR

Le *CCR*, ou *Complementary Circulating Register*, est très similaire au PCR. Nous considérons cette fois-ci un registre à décalage dont la fonction de rétroaction est :

$$f(x_0, x_1, \ldots, x_{n-1}) = x_0 \oplus 1.$$

De la même manière que pour les PCR, il est possible de déterminer le nombre $Z^*(n)$ de cycles que comporte le graphe de transition d'un tel registre [27] :

$$Z^*(n) = \frac{Z(n)}{2} - \frac{1}{2n} \sum_{d|n} \phi(2d) 2^{n/2d}.$$

À partir de ces $Z^*(n)$ cycles, il est possible d'obtenir une suite de De Bruijn de la même manière que pour les PCR. Nous obtenons ainsi une suite de De Bruijn pour une fonction de rétroaction de poids maximal.

4.7 Le D-homomorphisme

Nous allons maintenant aborder une méthode de construction de suite de De Bruijn un peu plus complexe, reposant sur l'homomorphisme de Lempel [48].

Commençons par introduire quelques définitions qui vont nous permettre de mieux appréhender la suite.

4.7.1 Définitions

Nous avons vu dans la section 1.3.3 une condition nécessaire et suffisante pour assurer la non singularité d'une fonction de rétroaction (théorème 1), et nous ne nous intéresserons qu'à ce type de fonctions.

Définition 18 *[48] Soient* $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ *et* $y = (y_{n-1}, y_{n-2}, \ldots, y_0)$ *deux vecteurs de* $\mathbb{F}_2^n$*, et soit la relation* $\longrightarrow$ *définie par :*

$$x \longrightarrow y \text{ si } (x_{n-1}, x_{n-2}, \ldots, x_2, x_1) = (y_{n-2}, y_{n-3}, \ldots, y_1, y_0).$$

Cette définition est à mettre en relation avec les flèches des états d'un graphe de De Bruijn : en effet, $x \longrightarrow y$ signifie qu'il existe une flèche entre de l'état x vers l'état y dans le graphe de De Bruijn d'ordre n.

Nous noterons un cycle dans G_n sous la forme $C = [\alpha_0 \alpha_1 \ldots \alpha_{k-1}]$, où α_i représente le bit de poids faible du i^{e} état traversé pendant le parcours du cycle. Un cycle de longueur k est appelé *k-cycle*.

Définition 19 *[48] Soit* $x = (x_{n-1}, x_{n-2}, \ldots, x_0)$ *un état de* G_n*. On appelle* état conjugué de x*, l'état* $\widehat{x}$ *défini par :*

$$\widehat{x} = (x_{n-1}, x_{n-2}, \ldots, x_1, \overline{x_0})$$

De même, on appelle état dual de l'état x*, l'état* $\overline{x}$ *définie par :*

$$\overline{x} = (\overline{x_{n-1}}, \overline{x_{n-2}}, \ldots, \overline{x_0}).$$

Définition 20 *[48] Un cycle $C = [\alpha_0\alpha_1 \dots \alpha_{k-1}]$ dans G_n est dit* primitif *si pour tout état x du cycle, l'état $\widehat{x}$ n'appartient pas à C. À l'inverse, un cycle qui n'est pas primitif est appelé cycle* réductible.

Il est facile de voir qu'un cycle est dit primitif s'il ne peut pas être découpé en deux cycles plus petits par modification du successeur d'un de ses états.

De la même manière que pour les états, il est possible de parler de cycles *duaux*.

Définition 21 *[48] Deux k-cycles C et $\overline{C}$, définis par $C = [\alpha_0, \alpha_1, \dots \alpha_{k-1}]$ et $\overline{C} = [\overline{\alpha_0}, \overline{\alpha_1}, \dots \overline{\alpha_{k-1}}]$ sont dit* duaux *l'un de l'autre. De plus, si $\overline{C}$ n'est autre qu'un décalage de C, alors C est dit* auto-dual.

Il apparaît alors le résultat suivant :

Lemme 5 *[48] Un k-cycle $C = [\alpha_0, \alpha_1, \dots, \alpha_{k-1}]$ est auto-dual si et seulement si $k = 2p$ et $\forall i \in [0, \dots p-1], \alpha_{i+p} = \overline{\alpha_i}$.*

Preuve : Il est évident qu'un cycle ne peut être auto-dual que s'il est de longueur paire. De plus, si un cycle C est auto-dual et qu'un état x appartient à ce cycle, alors l'état $\overline{x}$ appartient aussi à C. D'après la définition 21 nous savons que $\overline{C}$ est un décalage de C ; or, si y est le successeur de x dans C, alors $\overline{y}$ est le successeur de $\overline{x}$, ceci n'étant possible pour tous les états que si la distance séparant deux état duaux est la même dans tout le cycle. Cette distance est p. ◇

Définition 22 *[48] Le* poids d'un cycle *est défini comme étant le nombre d'états de ce cycle dont le bit de poids faible est non nul : le poids d'un k-cycle C est donc obtenu par :*

$$W(C) = \sum_{i=0}^{k-1} \alpha_i$$

Nous pouvons voir la définition du poids d'un cycle comme étant le *poids de Hamming* du mot binaire représentant ce cycle selon la représentation des cycles donnée précédemment.

Nous définissons enfin la projection D (appelée D-homomorphisme) de la manière suivante :

Définition 23 *[48] Soit $x = (x_{n-1}, \dots, x_0) \in \mathbb{F}_2^n$. La fonction $D_n : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{n-1}$ est définie par*[1]

$$D_n(x) = (x_{n-1} \oplus x_{n-2}, x_{n-2} \oplus x_{n-3}, \dots, x_2 \oplus x_1, x_1 \oplus x_0).$$

[1] dans son article, Lempel note la fonction $D_n(x) = y$ sous la forme $xD = y$.

4.7.2 Propriétés de D

Nous allons maintenant dégager les principales propriétés de D lorsqu'il est appliqué sur des cycles de G_n.

Tout d'abord, on remarque que D est une application linéaire de $\mathbb{F}_2^n$ dans $\mathbb{F}_2^{n-1}$ surjective et dont le noyau est $\{(0,0,\ldots,0),(1,1,\ldots,1)\}$. Par conséquent, tout élément de $\mathbb{F}_2^{n-1}$ est l'image par D d'un élément de $\mathbb{F}_2^n$: en effet, si l'on définit le vecteur X par :

$$X = (\bigoplus_{i=0}^{n-1} x_i, \bigoplus_{i=0}^{n-2} x_i, \ldots, x_1 \oplus x_0, \ x_0, \ 0)$$

alors,

$$\begin{aligned} \forall k, X_k &= \bigoplus_{i=0}^{k} x_i \\ D_k(X) &= \bigoplus_{i=0}^{k} x_i \oplus \bigoplus_{i=0}^{k-1} x_i = x_k \\ D_k(X) &= x \end{aligned}$$

De plus, pour x et y dans $\mathbb{F}_2^n$, il est facile de démontrer que $D_n(x) = D_n(y)$ si et seulement si $x = y$ ou $x = \overline{y}$, ce qui signifie que tout élément de $\mathbb{F}_2^{n-1}$ est l'image par D d'une paire d'éléments duaux de $\mathbb{F}_2$ (D n'est pas injective).

Un propriété intéressante de D est qu'il préserve la relation de décalage. Soient x et y dans $\mathbb{F}_2^n$ et tels que $x \rightarrow y$:

$$x = (x_{n-1}, \ldots x_0), \ y = (y_{n-1}, x_{n-1}, \ldots, x_1)$$

$$D_n(x) = (x_{n-1} \oplus x_{n-2}, \ldots, x_2 \oplus x_1, x_1 \oplus x_0)$$

$$D_n(y) = (y_{n-1} \oplus x_{n-1}, x_{n-1} \oplus x_{n-2}, \ldots, x_2 \oplus x_1)$$

donc

$$D_n(x) \rightarrow D_n(y).$$

Il est donc intéressant d'étendre la définition de D à des cycles :

Définition 24 *[48] Soit $C = [\alpha_0\alpha_1 \ldots \alpha_{k-1}]$ un k-cycle de G_n. On note $\Gamma = D_k(C)$ le k-cycle défini par :*

$$\Gamma = [\gamma_0\gamma_1 \ldots \gamma_{k-1}]$$

avec

$$\begin{aligned} \gamma_i &= \alpha_i \oplus \alpha_{i+1} \qquad i = 1, \ldots, k-2 \\ et\ \gamma_{k-1} &= \alpha_{k-1} \oplus \alpha_0 \end{aligned} \tag{4.2}$$

Cette extension nous amène à un nouveau résultat :

Théorème 11 *[48] Un k-cycle Γ dans G_{n-1} est l'image par D d'un k-cycle primitif dans G_n si et seulement si $W(\Gamma) \equiv 0 \mod 2$.*

Preuve : Considérons un cycle $C = [\alpha_0\alpha_1 \ldots \alpha_{k-1}]$ primitif de G_n. Si $\Gamma = D_k(C)$, nous avons par définition :

$$\Gamma = [(\alpha_0 \oplus \alpha_1)\ (\alpha_1 \oplus \alpha_2)\ \ldots (\alpha_{k-1} \oplus \alpha_0)]$$

par conséquent :

$$\begin{aligned} W(\Gamma) \mod 2 &= \bigoplus_{i=0}^{k-1} \gamma_i \\ &= \bigoplus_{i=0}^{k-1} \alpha_i \oplus \alpha_{(i+1 \mod k)} \\ &= (\alpha_0 \oplus \alpha_1) \oplus (\alpha_1 \oplus \alpha_2) \oplus \ldots (\alpha_{k-1} \oplus \alpha_0) \\ &= 0, \end{aligned}$$

ce qui entraîne : C primitif $\Rightarrow W(\Gamma) \mod 2 = 0$.

Considérons maintenant un k-cycle $\Gamma = [\gamma_0\gamma_1 \ldots \gamma_{k-1}]$ de poids pair dans G_{n-1}. Définissons $C = [\alpha_0\alpha_1 \ldots \alpha_{k-1}]$, un k-cycle dans G_n, avec :

$$\begin{aligned} \alpha_0 &= 0 \text{ ou } 1 \\ \alpha_i &= \alpha_0 \oplus \bigoplus_{j=0}^{i-2} \gamma_j \qquad i = 1, \ldots, k-1. \end{aligned}$$

Comme $W(\Gamma) \equiv 0 \mod 2$, nous avons

$$\gamma_{k-1} = \bigoplus_{i=0}^{k-2} \gamma_i$$

or $\alpha_{k-1} = \alpha_0 \oplus \bigoplus_{i=0}^{k-2} \gamma_i$, ce qui entraîne

$$\gamma_{k-1} = \alpha_{k-1} \oplus \alpha_0$$

ce qui est cohérent avec la définition de l'application de D sur un cycle.

Il est alors facile de voir que $D_k(C) = D_k(\overline{C}) = \Gamma$. Étant donné que Γ est un k-cycle de G_{n-1}, tous les états sont distincts, et par conséquent, les vecteurs formés par les α_i sont tous distincts, mais aussi, si l'on considère deux de ces mots, alors ils ne peuvent pas être duaux l'un de l'autre, ce qui implique que C est primitif. ◇

De manière analogue, nous obtenons :

Lemme 6 *[48] L'image par D d'un 2k-cycle auto-dual est un k-cycle. Si*

$$C = [\alpha_0, \alpha_1, \ldots \alpha_{2k-1}],$$

alors

$$D_{2k}(C) = \Gamma = [\alpha_0 \oplus \alpha_1, \alpha_1 \oplus \alpha_2, \ldots, \alpha_{k-2} \oplus \alpha_{k-1}]$$

Ceci découle du lemme 5.

Théorème 12 *[48] Un k-cycle Γ dans G_{n-1} est l'image par D d'un 2k-cycle C auto-dual dans G_n si et seulement si $W(\Gamma) \equiv 1 \mod 2$.*

Preuve : Si l'on considère le lemme précédent, on s'aperçoit que :

$$\begin{aligned} W(\Gamma) \mod 2 &= \sum_{i=0}^{k-1} \gamma_i \\ &= (\alpha_0 \oplus \alpha_1) \oplus (\alpha_1 \oplus \alpha_2) \oplus \ldots \oplus (\alpha_{k-1} \oplus \alpha_k) \\ &= \alpha_0 \oplus \alpha_k \end{aligned}$$

Or, pour un cycle auto-dual, nous avons $\alpha_i = \overline{\alpha_{i+k}}$, et donc $\alpha_0 \oplus \alpha_k = 1$, ce qui implique que $W(\Gamma)$ doit être impair. Tout comme les théorèmes précédents, si nous posons C comme étant :

$$\alpha_i = \gamma_0 \oplus \gamma_1 \oplus \ldots \oplus \gamma_{i-2} \qquad i = 1, \ldots, k-1$$

$$C = [0, \alpha_1, \ldots, \alpha_{k-1}, 1, \overline{\alpha_1}, \ldots, \overline{\alpha_{2k-1}}]$$

Alors nous avons bien $D_{2k}(C) = \Gamma$. ◇

4.7.3 Application : construction de suites de De Bruijn

Nous allons maintenant utiliser cette analyse des cycles dans G_n afin de construire des suites de De Bruijn [48]. Il est facile de voir que le poids d'un cycle correspondant à une suite de De Bruijn d'ordre n est nécessairement de poids pair puisque tous les motifs d'une longueur n donnée apparaissent une et une seule fois.

Considérons donc une suite de De Bruijn d'ordre $n-1$, et Γ le cycle correspondant dans G_{n-1}. Comme $W(\Gamma) \equiv 0 \mod 2$, nous sommes face aux conditions du théorème 11 : ainsi, Γ est l'image par D d'un cycle primitif dans G_n de longueur 2^{n-1}. De plus, comme $W(\Gamma) \equiv 0 \mod 2$, nous sommes certains que le cycle dont Γ est l'image n'est pas auto-dual. Par conséquent, il existe deux cycles C et $\overline{C}$ de longueur 2^{n-1} dans G_n tels que $D_n(C) = D_n(\overline{C}) = \Gamma$.

Il est envisageable de construire un cycle de longueur 2^n en prenant pour base ces deux cycles si l'on est capable de trouver un état x de C qui a pour successeurs potentiels un état de C et un état de $\overline{C}$.

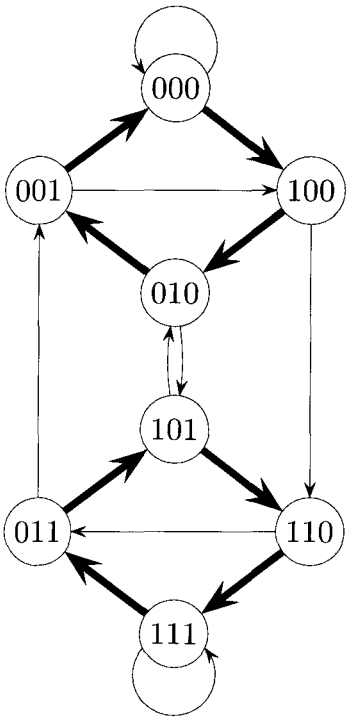


FIG. 4.3 – Deux cycles duaux de longueur 4 dans le graphe d'ordre 3

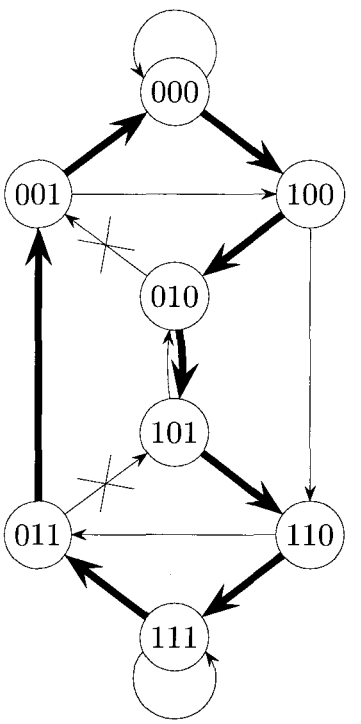


FIG. 4.4 – Modification du successeur de β et du prédecesseur de $\overline{\beta}$

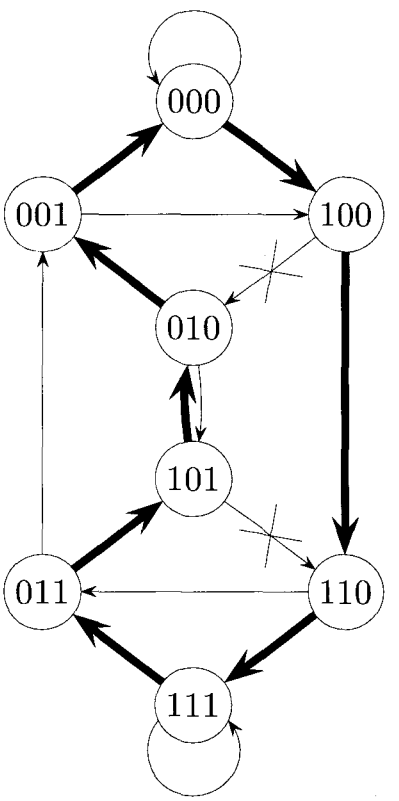


FIG. 4.5 – Modification du prédécesseur de β et du successeur de $\overline{\beta}$

Un tel état existe pour tout n ; considérons l'état

$$\beta_n = (n \mod 2, (n-1) \mod 2, \ldots, 0, 1, 0, 1).$$

Il est évident que si $\beta_n \in C$ alors $\overline{\beta_n} \in \overline{C}$. Considérons maintenant les successeurs possible de β_n : il s'agit des états

$$\begin{cases} s_n^0 = (0, \overline{n \mod 2}, \overline{(n-1) \mod 2}, \ldots, 0, 1, 0, 1) \\ s_n^1 = (1, \overline{n \mod 2}, \overline{(n-1) \mod 2}, \ldots, 0, 1, 0, 1) \end{cases}$$

L'état β_n possède donc un successeur potentiel sur C et l'autre sur $\overline{C}$: il existe donc une possibilité de fusionner les deux cycles C et $\overline{C}$ en modifiant le successeur de l'état β_n dans C, ainsi que le successeur du prédécesseur actuel de l'état $\overline{\beta_n}$ dans $\overline{C}$ (figure 4.4).

Une deuxième manipulation similaire est possible en modifiant cette fois le successeur du prédécesseur de l'état β et le successeur de l'état $\overline{\beta}$ (figure 4.5).

Forme normale algébrique

Il est assez simple de déterminer une *forme normale algébrique* (voir annexe A) de la fonction de rétroaction d'un registre à décalage générant la suite ainsi construite. Le principal problème est de déterminer la FNA du registre ayant pour graphe de transition les cycles C et $\overline{C}$. Nous allons utiliser la connaissance de $D_n(C) = D_n(\overline{C}) = \Gamma$.

Soit γ la fonction de rétroaction associée au registre de taille $n-1$ engendrant Γ, et soient x et y deux états de C tels que $x \longrightarrow y$.

Étant donné que Γ est l'image par D_n de C, il apparaît que $D_n(\mathrm{x}) \longrightarrow D_n(\mathrm{y})$, ce qui signifie que si $\mathrm{x} = (x_{n-1}, x_{n-2}, \ldots, x_0)$ alors $\mathrm{y} = (f(x), x_{n-1}, \ldots x_1)$. Or,

$$D_n(\mathrm{x}) = (x_{n-1}, x_{n-2}, \ldots, x_0) = (x_{n-1} \oplus x_{n-2}, x_{n-2} \oplus x_{n-3}, \ldots, x_1 \oplus x_0)$$

et

$$D_n(\mathrm{y}) = D_n((f(x), x_{n-1}, \ldots, x_1)) = (f(x) \oplus x_{n-1}, x_{n-1} \oplus x_{n-2}, \ldots, x_2 \oplus x_1),$$

et comme $D_n(\mathrm{x}) \longrightarrow D_n(\mathrm{y})$, nous obtenons que

$$x_{n-1} \oplus f(x) = \gamma(D_n(\mathrm{x}))$$

ou encore :

$$f(x) = x_{n-1} \oplus \gamma(D_n(\mathrm{x})).$$

À partir de cette fonction de rétroaction, il est donc possible d'obtenir une fonction générant une suite de De Bruijn. Prenons la première modification énoncée plus haut. Le principe étant de modifier le successeur de β_n et le successeur du prédécesseur de l'état $\overline{\beta_n}$, il suffit de modifier la table de vérité de la fonction de rétroaction de la manière suivante :

$$f_1(x) = x_{n-1} \oplus \gamma(D_n(\mathrm{x})) \oplus x_1^{e_1} x_2^{e_2} x_3^{e_3} \ldots x_{n-1}^{e_{n-1}}$$

où $x^1 = x, x^0 = \overline{x}$ et $e_i = i \mod 2$. Pour la seconde méthode, nous obtenons un résultat similaire :

$$f_2(x) = x_{n-1} \oplus \gamma(D_n(\mathrm{x})) \oplus \overline{x_1^{e_1} x_2^{e_2} x_3^{e_3} \dots x_{n-1}^{e_{n-1}}}$$

Cette méthode de construction est de grand intérêt puisqu'elle permet, à partir d'une suite d'ordre $n-1$, de construire deux suites d'ordre n de manière peu coûteuse, et ainsi à partir d'une suite de De Bruijn de petit ordre, de construire un grand nombre de suites de De Bruijn différentes de grand ordre. Malheureusement, il n'est pas possible de construire la totalité des $2^{2^{n-1}-n}$ suites. Je présenterai dans le chapitre 5, un système de chiffrement par flot fondé sur ce principe.

Chapitre 5

Élaboration d'un nouveau système de chiffrement par flot

5.1 Introduction

Je vais décrire dans ce chapitre la construction d'un système de chiffrement par flot, fondé sur l'utilisation de *registres à décalage à rétroaction non linéaire*, ou *NLFSRs* (*Non Linear Feedback Shift Registers*). Le but de ce système sera de construire des suites pseudo-aléatoires possédant de bonnes propriétés statistiques, mais aussi facilement calculables sur une machine ayant des capacités de calcul limitées.

Ce système n'est pas le seul à utiliser ce type de primitives. Nous pouvons par exemple citer *Achterbahn* [33]. La principale nouveauté de cette construction réside dans la dépendance entre les fonctions de rétroaction des NLFSRs et l'initialisation du système.

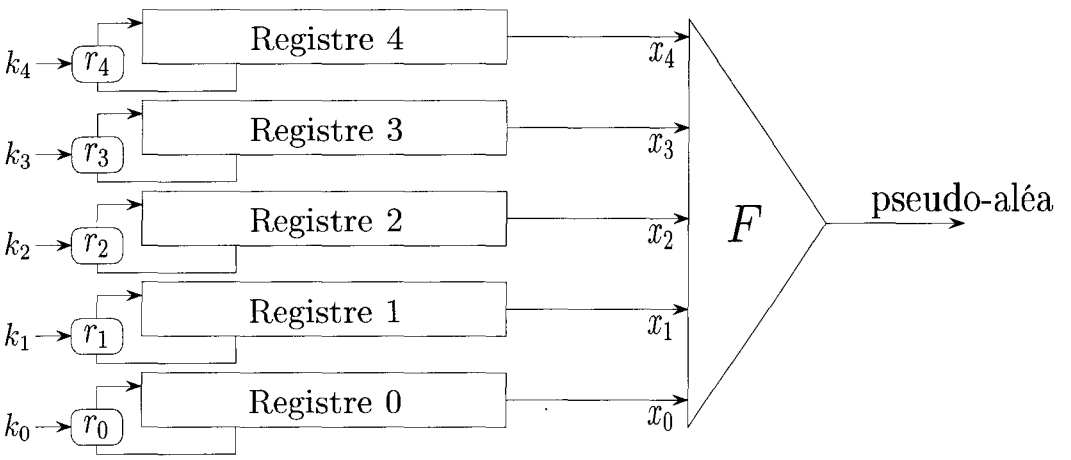
Le modèle repose sur celui des registres combinés, à la différence que nous n'utiliserons pas ici de LFSRs, mais plutôt des NLFSRs . De plus, le générateur sera dynamique, c'est-à-dire qu'à la différence du modèle des registres combinés les fonctions de rétroactions utilisées pour les registres seront totalement décrites par la clé du système. Ces choix sont motivés par la nécessité d'avoir des suites possédant une complexité linéaire importante, afin d'éviter les attaques de type Berlekamp-Massey [4, 50], ainsi que la possibilité d'éviter les attaques algébriques [18, 20] en multipliant le nombre d'équations nécessaires à la description du système.

5.2 Générateur fondé sur des NLFSRs à rétroaction dynamique combinés

5.2.1 Vue générale

Le principe général du générateur est de combiner cinq registres à décalage à rétroaction non linéaire grâce à une fonction $F(x_4, x_3, x_2, x_1, x_0)$. Chaque registre est de taille fixe. Nous noterons L_i la taille du registre $i \in [0..4]$: $L_i = i + 45$. La principale différence avec les schémas classiques est que les fonctions de rétroaction utilisées par les

registres sont entièrement décrites par la clé du système : nous noterons $r_i(I)$ la fonction de rétroaction du registre i, avec I l'initialisation du système.



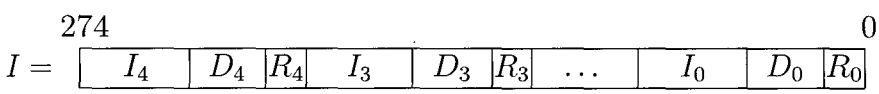
5.2.2 Fonctionnement

Découpage de l'initialisation

L'initialisation du système, composée de 275 bits, est découpée en 5 blocs B_i de L_i+8 bits. Chacun de ces blocs représente la *manière de construire le registre* i ; ils seront découpés eux même en trois parties :

- la *racine* du générateur de fonction, notée R_i, de 4 bits,
- le *descripteur de sauts* du générateur de fonction, noté D_i, de 4 bits.
- l'*initialisation* du registre i, soit L_i bits, notée I_i.

Pour résumer, l'initialisation I est découpée de la manière suivante :



Pour des raisons évoquées plus loin, la période du générateur est approximativement de 2^{230} bits.

Construction des fonctions de rétroaction

La partie la plus importante du cryptosystème est la construction des fonctions de rétroaction des registres à décalage en fonction de la clé. Nous supposons que l'initialisation I a correctement été découpée en suivant les indications du point précédent. Nous disposons alors pour chaque registre i d'un triplet (I_i, R_i, D_i) décrivant la fonction de rétroaction, ainsi que l'initialisation du registre.

Le principe de construction de la table de vérité des fonctions g_i repose sur les travaux de Lempel sur les D-homomorphismes [48] ainsi que sur la méthode décrite dans [15] (voir chapitre 4).

Pour rappel, le D-homomorphisme $D_n : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{n-1}$ est défini par [48] :

$$D_n((x_{n-1}, x_{n-2}, \ldots x_0)) = (x_{n-1} \oplus x_{n-2}, x_{n-2} \oplus x_{n-3}, \ldots, x_1 \oplus x_0).$$

Nous définissons alors la *puissance k-ième de D*, $D_n^k : \mathbb{F}_2^n \mapsto \mathbb{F}_2^{n-k}$ comme étant :

$$D_n^k(\mathrm{x}) = D_{n-k+1} \circ D_{n-k+2} \circ \ldots \circ D_{n-1} \circ D_n(\mathrm{x}).$$

Nous pouvons alors montrer facilement par récurrence les propriétés suivantes [15] :

- Si m est une puissance de 2 positive, alors

$$D_{m+1}^m((x_m, x_{m-1}, \ldots, x_0)) = (x_0 \oplus x_m),$$

- Si m est une puissance de 2 positive, et $n > m$, alors

$$D_n^m((x_{n-1}, x_{n-2}, \ldots, x_0)) = (x_{n-1} \oplus x_{n-m-1}, x_{n-2} \oplus x_{n-m-2}, \ldots, x_0 \oplus x_m).$$

Afin de simplifier l'écriture des algorithmes qui suivront, nous introduisons quelques définitions [15] :

Définition 25 *Soient les vecteurs de* $\mathbb{F}_2^n$ *:*

$$\begin{aligned}
\mathbf{0}_n &= (0, 0, 0 \ldots, 0) \\
\beta_n &= (\quad n \mod 2, \ldots, 0, 1, 0, 1) \\
\overline{\beta_n} &= ((n-1) \mod 2, \ldots, 1, 0, 1, 0) \\
\mathbf{1}_n &= (1, 1, 1, \ldots, 1)
\end{aligned}$$

Définition 26 *Soient les fonctions :*

$$\begin{array}{lrcl}
S: & \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2^n \\
& (x_{n-1}, x_{n-2}, \ldots, x_0) & \longmapsto & (0, x_{n-1}, x_{n-2}, \ldots, x_1) \\
R: & \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2^{n-1} \\
& (x_{n-1}, x_{n-2}, \ldots, x_0) & \longmapsto & (x_{n-1}, x_{n-2}, \ldots, x_1) \\
P_i: & \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2 \\
& (x_{n-1}, x_{n-2}, \ldots, x_0) & \longmapsto & x_i \\
U: & \mathbb{F}_2^n & \longrightarrow & \mathbb{F}_2 \\
x = (x_{n-1}, x_{n-2}, \ldots, x_0) & & \longmapsto & 1 \textit{ si } x = \beta_n, \ 0 \textit{ sinon}
\end{array}$$

On remarquera que les vecteurs β et $\overline{\beta}$ sont en fait les nœuds du graphe de de Bruijn où l'on peut effectuer une fusion pour obtenir, à partir des deux antécédents par D d'un cycle de de Bruijn d'ordre $n-1$, un cycle de de Bruijn d'ordre n (*cf* chapitre 4).

Grâce à ces fonctions primitives, il est alors possible de décrire la construction d'une fonction f de de Bruijn d'ordre n à partir d'une fonction f' d'ordre $n-1$. En appliquant la méthode décrite au chapitre 4, nous obtenons pour $\mathrm{x} = (x_{n-1}, x_{n-2}, \ldots, x_0) \in \mathbb{F}_2^n$:

$$f(\mathrm{x}) = P_{n-1}(\mathrm{x}) \oplus f'(D(\mathrm{x})) \oplus U(R(S(\mathrm{x}))).$$

Il est possible de généraliser cette méthode : dans [15], les auteurs proposent une méthode permettant, à partir d'une fonction f' d'ordre k, de calculer une fonction f d'ordre n, $n > k$, de manière efficace. Cette méthode est fondée sur la possibilité de calculer D^k pour un k quelconque positif de manière efficace, et sur un algorithme appelé *algorithme* Δ capable de calculer efficacement par dichotomie la valeur de :

$$\Delta_i^n(\mathrm{x}) = U(\mathrm{x}) \oplus U(D(\mathrm{x})) \oplus \ldots \oplus U(D^{n-i-1}(\mathrm{x})).$$

L'algorithme Δ est celui-ci [15] :

En entrée nous avons $n, i,$ et x

\# Tests préliminaires
si $\mathrm{x} = \beta_n$, alors $\Delta_i^n(\mathrm{x}) := 1$. Fin
si $\mathrm{x} = \overline{\beta_n}$ ou $\mathrm{x} = \mathbf{0}_n$ ou $\mathrm{x} = \mathbf{1}_n$ alors $\Delta_i^n(\mathrm{x}) := 0$. Fin
$d := n - i + 1$
si $D^d(\mathrm{x}) \neq \mathbf{0}_{n-d}$ alors $\Delta_i^n(x) := 0$. Fin

\# Début de la dichotomie
$k := 1, m := d - 1$
tant que $k \leq m$ faire :
$t := \lfloor \frac{k+m}{2} \rfloor$
tests:
si $D^t(\mathrm{x}) = \beta_{n-t}$ alors $\Delta_i^n(\mathrm{x}) := 1$. Fin
si $D^t(\mathrm{x}) = \overline{\beta_{n-t}}$ alors $\Delta_i^n(\mathrm{x}) := 0$. Fin
si $D^t(\mathrm{x}) = \mathbf{1}_{n-t}$ alors $t := t - 1$. Retourner au début des tests
si $D^t(\mathrm{x}) = \mathbf{0}_{n-t}$ alors $m := t - 1$ sinon $k = t$
Fin de la boucle

La fonction $f(\mathrm{x})$ d'ordre n est alors exprimée sous la forme [15] :

$$f(\mathrm{x}) = P_{k-1}(D^{n-k}(S(\mathrm{x}))) \oplus f'(D^{n-k}(\mathrm{x})) \oplus \Delta_{k-1}^{n-1}(R(S(\mathrm{x})))$$

Nous pouvons apporter une amélioration très intéressante à l'algorithme proposé par les auteurs de [15] : en effet, à partir d'une fonction d'ordre k, l'algorithme n'est capable de fournir qu'une unique fonction d'ordre n, $n > k$. Il suffit de remarquer que pour toute fonction de de Bruijn f' d'ordre $n - 1$, les fonctions suivantes sont bien des fonctions de de Bruijn d'ordre n :

$$\begin{aligned}
&P_{n-1}(\mathrm{x}) \oplus f'(D(\mathrm{x})) \oplus U(R(S(\mathrm{x}))) \\
&P_{n-1}(\mathrm{x}) \oplus f'(D(\mathrm{x})) \oplus U(\mathbf{1}_{n-1} \oplus R(S(\mathrm{x}))) \\
&P_{n-1}(\mathrm{x} \oplus \mathbf{1}_\mathrm{n}) \oplus f'(D(\mathrm{x} \oplus \mathbf{1}_\mathrm{n})) \oplus U(R(S(\mathrm{x} \oplus \mathbf{1}_\mathrm{n}))) \oplus 1 \\
&P_{n-1}(\mathrm{x} \oplus \mathbf{1}_\mathrm{n}) \oplus f'(D(\mathrm{x} \oplus \mathbf{1}_\mathrm{n})) \oplus U(\mathbf{1}_{n-1} \oplus R(S(\mathrm{x} \oplus \mathbf{1}_\mathrm{n}))) \oplus 1
\end{aligned}$$

La première forme est celle utilisée par les auteurs de [15].

La seconde consiste à choisir l'état $\overline{\beta_n}$ plutôt que l'état β_n pour opérer la fusion des cycles dans le graphe de de Bruijn. Il est donc évident que cette fonction sera elle aussi une fonction engendrant une suite de de Bruijn.

Pour les deux dernières formes, il suffit de remarquer que si $(s_n)_n$ est une suite de de Bruijn, alors $(\overline{s_n})_n$ en est une aussi : si l'on connaît la fonction de rétroaction $f(\mathrm{x})$ d'un registre à décalage engendrant $(s_n)_n$, alors la fonction de rétroaction du registre à décalage engendrant $(\overline{s_n})_n$ est $\overline{f(\overline{\mathrm{x}})}$, ou encore $f(\mathrm{x} \oplus \mathbf{1}_n) \oplus 1$.

Il est alors possible à partir d'une fonction de de Bruijn d'ordre $n-1$ de construire quatre fonctions de de Bruijn d'ordre n. Malheureusement, on s'aperçoit très vite que les tables de vérité des fonctions de la première et de la deuxième forme sont très proches, tout comme le sont celles des fonctions de troisième et quatrième forme.

Etant donné que le système de chiffrement fait intervenir une initialisation dans la construction de ses fonctions de rétroaction, il est inenvisageable de construire deux fonctions proches à partir de deux initialisations différentes. Ainsi, parmi ces quatre formes, nous ne retiendrons que la première et la quatrième qui maximisent le nombre de différences entre leur tables de vérités.

Nous retiendrons donc pour la construction de fonctions de de Bruijn d'ordre n à partir d'une fonction f' de de Bruijn d'ordre k, les fonctions :

$f_1(\mathrm{x}) = P_{k-1}(D^{n-k}(S(\mathrm{x}))) \oplus f'(D^{n-k}(\mathrm{x})) \oplus \Delta'^{n-1}_{k-1}(R(S(\mathrm{x})), 0)$
$f_2(\mathrm{x}) = P_{k-1}(D^{n-k}(S(\mathrm{x} \oplus \mathbf{1}_n))) \oplus f'(D^{n-k}(\mathrm{x} \oplus \mathbf{1}_n)) \oplus \Delta'^{n-1}_{k-1}(R(S(\mathrm{x} \oplus \mathbf{1}_n)), 1) \oplus 1$

où $\Delta'^n_i(\mathrm{x}, T)$ est *l'algorithme* Δ *modifié* :

$\Delta'^n_i(\mathrm{x}, T)$:
si $T = 0$ alors
$p_\beta = \beta$ et $p_{\overline{\beta}} = \overline{\beta}$
sinon
$p_\beta = \overline{\beta}$ et $p_{\overline{\beta}} = \beta$

si $\mathrm{x} = p_{\beta_n}$, alors $\Delta^n_i(\mathrm{x}) := 1$. Fin
si $\mathrm{x} = p_{\overline{beta_n}}$ ou $\mathrm{x} = \mathbf{0}_n$ ou $\mathrm{x} = \mathbf{1}_n$ alors $\Delta^n_i(\mathrm{x}) := 0$. Fin
$d := n - i + 1$
si $D^d(\mathrm{x}) \neq \mathbf{0}_{n-d}$ alors $\Delta^n_i(\mathrm{x}) := 0$. Fin

$k := 1, m := d - 1$
tant que $k \leq m$ faire :
$t := \lfloor \frac{k+m}{2} \rfloor$
tests:
si $D^t(\mathrm{x}) = p_{\beta_{n-t}}$ alors $\Delta^n_i(\mathrm{x}) := 1$. Fin
si $D^t(\mathrm{x}) = p_{\overline{\beta_{n-t}}}$ alors $\Delta^n_i(\mathrm{x}) := 0$. Fin
si $D^t(\mathrm{x}) = \mathbf{1}_{n-t}$ alors $t := t - 1$. Retour au début des tests
si $D^t(\mathrm{x}) = \mathbf{0}_{n-t}$ alors $m := t - 1$ sinon $k = t$
Fin de la boucle
Fin de $\Delta'^n_i(\mathrm{x}, T)$

Le principe de construction de la fonction de rétroaction du registre i est celui-ci :

- on choisit une fonction $r_i^4(\mathrm{x})$ parmi les seize fonctions de de Bruijn d'ordre 4 : ce choix est guidé par la partie R_i de la clé ;
- à partir de la fonction $r_i^4(\mathrm{x})$ d'ordre 4 choisie, on construit une fonction de de Bruijn $r_i^6(\mathrm{x})$ d'ordre 6 grâce à la construction exposée précédemment, avec $n = 6$ et $k = 4$. Le choix entre la forme f_1 et la forme f_2 est guidé par le bit de poids faible de D_i ;
- de la même manière, à partir de la fonction $r_i^6(\mathrm{x})$ d'ordre 6, on construit une fonction de de Bruijn $r_i^8(\mathrm{x})$ d'ordre 8, guidé par le deuxième bit de poids le plus faible de D_i ;
- puis on construit la fonction $r_i^{10}(\mathrm{x})$ à partir de $r_i^8(\mathrm{x})$;
- on construit ensuite la fonction $r_i^{L_i-32}(\mathrm{x})$ à partir de $r_i^{10}(\mathrm{x})$;
- enfin, on construit la fonction $r_i^{L_i}(\mathrm{x})$ à partir de $r_i^{L_i-32}(\mathrm{x})$.

Si on les laisse telles quelles, les fonctions $r_i^{L_i}$ étant des fonctions de de Bruijn, la période de chaque registre i est donc de 2^{L_i}. La période globale du générateur pseudo-aléatoire est donc le plus petit commun multiple de ces périodes, c'est à dire 2^{49} dans ce cas. Cette période est bien évidemment beaucoup trop courte : nous allons donc l'augmenter en modifiant la période de chaque registre.

Pour ce faire, nous allons volontairement exclure l'état 0_{L_i} du cycle de chaque registres, en modifiant les valeurs des vecteurs $(0, 0, \ldots, 0, 0)$ et $(0, 0, \ldots, 0, 1)$ de la table de vérité des fonctions de rétroaction. Ainsi la période du générateur sera donc $\mathrm{ppcm}(2^{45} - 1, 2^{46} - 1, \ldots, 2^{49} - 1) > 2^{230}$, soit plus de $1,9 \times 10^{56}$ téra-octets, ce qui devrait être largement suffisant pour un usage dans un *réseau de petits objets portables sécurisés.*

Fonction de combinaison

La fonction de combinaison doit être choisie avec soin : c'est elle qui permet d'augmenter la *complexité linéaire* de la suite produite, et permet ainsi d'échapper à l'attaque de Berlekamp-Massey [4, 50] : elle doit donc être du plus haut degré possible. De plus, pour éviter les attaques par corrélation, la fonction doit avoir un ordre de résilience et une non-linéarité le plus haut possible.

La fonction choisie est :

$$\begin{aligned} F(x_4, x_3, x_2, x_1, x_0) = \; & x1.x3.x4 + x0.x2.x4 + x0.x1.x4 + x0.x1.x2 \\ & + x0.x2.x3 + x0.x1.x3 + x3.x4 + x2.x4 + x2.x3 \\ & + x1.x3 + x0.x3 + x0.x2 + x0.x1 + x1 + x0 + 1 \end{aligned}$$

Il s'agit d'une fonction équilibrée, de non-linéarité 12, de degré algébrique 3 et 1-résiliente. De par son ordre de résilience, la fonction de combinaison permet donc au système d'espérer une sécurité de l'ordre de $2^{45+46} = 2^{91}$ au moins, puisqu'il faudra considérer une recherche exhaustive des initialisations de deux registres au minimum.

Initialisation à partir de la clé et du vecteur initial

La clé secrète du système K est une chaîne de 128 bits. Cette chaîne va être utilisée conjointement à un *vecteur initial* (IV) public de 64 bits. Le rôle de ce vecteur est d'éviter de pouvoir récupérer de l'information à partir de deux messages qui auraient été chiffrés avec la même clé. La taille de l'IV à été choisie pour éviter d'être confronter à une trop forte probabilité de collision, et de se retrouver dans un cas similaire à celui du WiFi [38]. Concernant la clé secrète, elle est suffisament longue pour pouvoir éviter, à l'heure actuelle et avec des moyens raisonnables, toute tentative d'attaque par force brute.

Les bits constituant la clé de chiffrement sont concaténés aux bits du vecteur initial pour former un mot de 192 bits. Ce mot va alors servir d'état interne à un registre à décalage à rétroaction linéaire de 192 bits. Le polynôme de rétroaction $P(X) = X^{192} + X^{135} + X^{66} + X^2 + 1$ de ce registre a été choisi primitif. La fonction de rétroaction associée est donc :

$$r(x_{191}, x_{190}, \ldots, x_0) = x_0 \oplus x_{57} \oplus x_{126} \oplus x_{190}.$$

Le registre est alors avancé 8 192 fois pour permettre à l'état interne de se diffuser dans la suite. Ensuite, 275 bits sont extraits afin d'obtenir l'état initial du générateur pseudo-aléatoire. Pour des raisons de sécurité, les cinq registres du générateur pseudo-aléatoire sont eux aussi avancés 256 fois sans être utilisés pour le chiffrement.

Implémentation

Le générateur a été implémenté en `C` (voir annexe C). Afin d'accélerer le processus de production du pseudo-aléa, le programme précalcule un ensemble de tables de vérité. L'ensemble de ces tables occupe à l'issue du processus un total de 31 744 octets, soit 31Ko. Le générateur est capable de construire des séquences pseudo-aléatoires à une vitesse d'environ 8Mb/s sur un Pentium IV à 2,6GHz, ce qui reste raisonnable par rapport aux objets mis en oeuvre.

Il reste cependant une piste à explorer concernant l'implémentation en *hardware*. En effet, le principal goulot d'étranglement de l'implémentation réside dans la recherche dichotomique opérée par l'algorithme Δ *modifié*. Cet algorithme effectue un très grand nombre de comparaisons afin de déterminer si une puissance k^{e} de l'homomorphisme D d'un vecteur possède un motif particulier. Cette recherche est assez coûteuse lorsqu'on l'exprime de manière logicielle, mais il est tout à fait raisonnable de penser qu'elle puisse se révèler très efficace une fois implémentée en hardware.

Imaginons la méthode suivante : pour un registre de taille n dont l'état interne est x, on construit une matrice binaire $\mathcal{M}$ de trente-trois lignes par n colonnes. La valeur de la i^{e} ligne et j^{e} colonne, $\mathcal{M}_{i,j}$, est définie par (les indices sont numérotés à partir de 0) :

$$\mathcal{M}_{i,j} = \begin{cases} \text{la } j^{\text{e}} \text{ composante de } D_n^i(\text{x}) \text{ si } j \leq n - i \\ 0 \text{ sinon} \end{cases}$$

Chaque éléments de $\mathcal{M}$ peut être calculé de manière indépendante de tous les autres, ce qui permet de réaliser les calculs en parallèle. Si l'on regarde plus précisément pour chacune des cinq matrices correspondant aux cinq registres du système proposé, on s'apperçoit alors que pour chacune d'elles, l'expression faisant intervenir le plus grand nombre de

variables est une somme de trente-deux éléments : on peut donc l'exprimer sous la forme d'un arbre de profondeur $log_2(32) = 5$, où chaque nœud correspond à un `OU-EXCLUSIF`.

Le temps de calcul de la totalité des éléments des cinq matrices correspond donc au temps qu'il faut pour calculer l'élément comportant le plus grand nombre d'opérations. Dans notre cas, le temps total de calcul de $\mathcal{M}$ est équivalent au temps qu'il faut à un signal électrique pour traverser cinq portes `OU-EXCLUSIF`.

À partir de la connaissance de $\mathcal{M}$, il est très rapide de déterminer la valeur de $\Delta'^{n}_{n-32}(\mathrm{x}, T)$ avec $T \in 0, 1$: il suffit de parcourir l'ensemble des lignes de la matrice, et de déterminer si l'une d'elle est une alternance de 0 et de 1 se terminant par T. Une fois de plus, cette recherche peut être menée en parallèle sur la totalité des trente trois lignes de la matrice, en combinant des portes `ET` et `NON-ET`. Notons qu'il est d'ailleurs possible d'imaginer tester la présence d'un mot de la forme β ou la présence d'un mot $\overline{\beta}$ dans le même temps. Le temps de cette recherche est donc équivalent au temps qu'il faut à un signal électrique pour traverser deux portes logiques.

Le coût financier de la réalisation du circuit capable d'effectuer ce calcul pour les cinq registres peut être évalué en fonction du nombre total de portes logiques qu'il faut utiliser pour le réaliser. Si l'on considère les matrices de manière naïve, il faut 27 030 portes `OU-EXCLUSIF` pour les réaliser, mais ce chiffre peut être très largement réduit en considérant la très grande redondance des calculs effectués. Une première simplification triviale, résultante d'une factorisation, permet déjà de réduire ce nombre à 15 955 portes. Il est tout à fait envisageable de réduire encore grandement ce nombre, en séquentialisant le calcul des matrices. Ainsi, le débit serait réduit par 5, mais le nombre de portes chuterait d'autant.

Le système de chiffrement complet pourrait donc être décomposé en deux parties : une partie logicielle permettant d'effectuer les pré-calculs des tables de vérité des fonctions de de Bruijn de petit ordre. Le résultat serait stocké dans une table de 31Ko accessible à la partie matérielle, qui correspondrait à l'algorithme Δ'.

Dans un premier temps, la partie matérielle pourrait être implémentée grâce à l'utilisation de *FPGAs* (*Field Programmable Gate Array*), qui sont des circuits comportant un très grand nombre de portes logiques dont les interconnexions sont programmables. Les modèles récents de FPGA permettent d'obtenir des circuits où le temps nécessaire pour traverser une porte logique est d'environ $0,4ns$. A cette vitesse, on peut raisonnablement estimer une production du pseudo-aléa d'environ 200 à 250Mb/s, soit environ 30Mo/s ! Dans ces conditions, ce système de chiffrement deviendrait déjà tout à fait exploitable, et facilement utilisable dans une machine présentant une architecture très contrainte.

Nous allons maintenant regarder comment se comporte le générateur par rapport aux différentes attaques sur les systèmes de chiffrement par flot que nous avons présentées au chapitre 2.

5.3 Étude statistique

Avant toute chose, il est nécessaire de vérifier la qualité statistique des suites : il doit être difficile de déterminer si une suite est issue de ce générateur, ou bien s'il s'agit

réellement d'une suite aléatoire.

Pour tenter de mesurer la qualité des suites, nous utilisons la batterie de tests du *NIST* [57], regroupée dans un programme appellé *Sts* [55], ainsi que la série de tests *Diehard* [75]. Ces tests n'ont révélé aucun biais particulier. Les résultats se trouvent à l'annexe B.

5.4 Comportement face aux attaques connues

5.4.1 Attaque de Berlekamp-Massey

Nous savons que pour pouvoir résister à cette attaque, un générateur pseudo-aléatoire se doit de construire une suite dont la complexité linéaire est la plus haute possible.

Nous voyons ici tout l'intérêt des registres à décalage à rétroaction non linéaire, qui, par rapport aux LFSRs, permettent de conserver un registre de petite taille, tout en produisant des suites dont la complexité linéaire est encadrée par le résultat connu :

Théorème 13 *[27] Soient $(s_n)_{n\geq 0}$ une suite de de Bruijn d'ordre k. La complexité linéaire $\lambda(s_n)$ de la suite vérifie $2^{k-1}+k \leq \lambda(s_n) \leq 2^k - 1$.*

Nous avons vu précédemment que la complexité linéaire de la suite produite dépend de la forme de la fonction de combinaison utilisée par le système. La complexité linéaire des suites se situe donc entre 2^{140} et 2^{143}. Cette complexité exclut totalement la possibilité d'utiliser l'algorithme de Berlekamp-Massey.

Malheureusement, le critère de complexité linéaire, vérifié par le programme de test précédent, n'est pas suffisant. Imaginons le cas d'une suite engendrée par un registre à décalage à rétroaction linéaire de période maximale. Si l'on modifie la fonction de rétroaction de ce registre afin d'obtenir une suite de de Bruijn (voir chapitre 4), nous obtenons une suite de complexité linéaire élevée. Il a été d'ailleurs démontré dans [14] que les suites construites de cette manière atteignent la borne maximale, c'est-à-dire une complexité linéaire de $2^k - 1$ pour un registre de taille k. Il est bien évident que ces suites ne sont par très intéressantes, puisque la quasi totalité de leur termes sont issus d'une récurrence linéaire.

Il est nécessaire alors d'étudier le *profil de complexité linéaire* de la suite : il s'agit de la suite des complexités linéaires des sous-suites de $(s_n)_{n\geq 0}$.

Nous connaissons le profil d'une suite réellement aléatoire ; il faut donc vérifier que le profil des suites générées par les différents registres collent au mieux au profil attendu par la théorie. L'espérance de la complexité linéaire λ d'une suite réellement aléatoire de i bits ainsi que sa variance, sont définies, pour i assez grand, par [51] :

$$E(\lambda) = \begin{cases} \frac{i}{2} + \frac{2}{9} \text{ si } i \text{ est pair} \\ \frac{i}{2} + \frac{5}{18} \text{ sinon} \end{cases}$$

$$Var(\lambda) \approx \frac{86}{81}$$

Malheureusement, il n'est pas possible de déterminer le profil de complexité linéaire de l'ensemble des suites produites par ce générateur. Une analyse à été menée sur un grand nombre d'initialisations, et le profil semble idéal pour tous les états testés.

5.4.2 Attaque par corrélation

Toujours à cause de la non linéarité des fonctions de rétroaction des registres, seule l'attaque par corrélation générale peut fonctionner, l'attaque par corrélation rapide utilisant la linéarité des fonctions de rétroaction. Dans notre cas, la fonction de combinaison est 1-résiliente, ce qui signifie que l'attaque par corrélation doit considérer au moins deux registres à la fois pour pouvoir fonctionner. C'est pourquoi, une attaque de ce type devra essayer au minimum 2^{81} combinaisons pour ces deux registres, suivie d'une recherche de l'initialisation des autres registres.

5.4.3 Attaques algébriques

La non-linéarité de ce système le préserve des attaques algébriques. En effet, dans ce cas-ci, il n'est pas possible d'écrire le système sous la forme d'une série d'équation $F(L^i(K))$ avec L une fonction linéaire d'actualisation de l'état interne, K la clé globale du système et F la fonction non-linéaire de sortie.

Notre générateur peut s'écrire comme un ensemble de $2^{5\times(4+4)} = 2^{40}$ générateurs différents, du fait que la rétroaction du système est en grosse partie construite par la clé. Imaginons qu'un attaquant puisse réussir à obtenir assez d'information pour déterminer lequel de ces 2^{40} systèmes est réellement utilisé. Il doit maintenant écrire le générateur sous la forme d'un système d'équations.

Chacun des registres i permettant de générer une suite de de Bruijn d'ordre L_i, leur polynôme de rétroaction est de degré $L_i - 1$. De ce fait, la fonction L utilisée pour l'attaque algébrique est de degré $\max(L_i) = 49$. Il n'est pas facile de faire une approximation linéaire de cette fonction dans le cadre d'une attaque algébrique.

5.4.4 Attaques par compromis temps/mémoire

La méthode utilisée pour déterminer l'état initial du générateur en fonction de la clé et du vecteur initial prend en compte le résultat donné au chapitre 2, c'est-à-dire que la taille de l'initialisation en bits doit être au moins deux fois plus grande que la taille de la clé. Ainsi, il ne sera pas plus facile de déterminer la valeur de l'initialisation par une recherche exhaustive, que de rechercher la clé directement.

Chapitre 6

Sécurité et bibliothèques logicielles font-elles bon ménage ?

6.1 Introduction

Avec la démocratisation des accès à Internet, et des réseaux en général, le besoin de sécurité et de confidentialité se fait de plus en plus ressentir. La plupart des logiciels du marché intègrent désormais un système de chiffrement permettant aux utilisateurs de pouvoir échanger des informations de manière sécurisée. Afin de faciliter la tâche des développeurs de logiciels, les primitives cryptographiques se trouvent directement incluses dans les systèmes d'exploitation.

Malgré tout, l'implémentation de primitives cryptographiques dans les systèmes d'exploitation ne peut garantir une sécurité absolue ; en effet, plusieurs points doivent être observés avec attention.

Les attaques susceptibles d'être menées contre de tels systèmes sont très souvent dues à une mauvaise conception du système de chiffrement lui-même. A ce titre, nous pourrons citer le système de chiffrement utilisé dans les *DVD* qui est devenu totalement inefficace dès lors que l'algorithme de chiffrement a été découvert. Il s'agit ici d'une attaque montrant une faiblesse conceptuelle du système de chiffrement.

Nous pouvons aussi trouver des attaques visant l'implémentation d'un protocole particulier ; récemment, il a été montré que le fait de donner trop d'informations à un attaquant quant au rejet d'une clé pouvait l'amener à reconstruire petit à petit des informations capitales [41]. Malgré le fait que l'algorithme de chiffrement ne souffre pas de faiblesse connue à ce jour, l'implémentation logicielle visée se trouve, elle, à la merci de nombreuses attaques.

Enfin, nous trouvons des attaques relatives à l'architecture logicielle des applications. En effet, avec les langages de programmation objet, l'heure est à la réutilisabilité et à la factorisation de code. Nous verrons dans cet article que cette architecture logicielle ne peut être préconisée si l'on souhaite préserver la sécurité.

6.2 Services et bibliothèques de fonctions

L'orientation actuelle des systèmes d'exploitation est d'offrir un maximum de services, utilisables de manière simple et efficace. Ainsi, nous trouvons dans la plupart des systèmes d'exploitation des bibliothèques de fonctions capables de fournir aux développeurs la possibilité de décoder des flux vidéos, ou encore de piloter une carte graphique très simplement.

Le grand intérêt de ce type de service est de simplifier la tâche des programmeurs, mais aussi de pouvoir factoriser au maximum le code des applications, et donc de minimiser la taille des fichiers exécutables finaux. En effet, beaucoup d'applications ont en commun un grand nombre de fonctionnalités.

Pour ce qui concerne la sécurité, nous trouvons désormais des bibliothèques de fonctions dont le but est d'offrir au programmeur un maximum de primitives cryptographiques, capables d'effectuer du chiffrement, de créer des clés ou encore de signer. Dans ce créneau, nous trouvons par exemple les bibliothèques *OpenSSL* [58] sous *Linux* ou encore la *CryptoAPI* [11] de *Microsoft* sous *Windows*.

Ces produits permettent une grande flexibilité quant à la création d'applications offrant des fonctionnalités de sécurisation. En effet, ils permettent de faire de la cryptographie sans pour autant devoir connaître dans le détail le fonctionnement des protocoles utilisés. En très peu de temps, et sans aucune connaissance cryptographique, il est facile de créer un programme dont toutes les transactions seront chiffrées de manière totalement transparente.

6.3 Problèmes relatifs à l'organisation centralisée

En règle générale, ces services sont implémentés sous forme de bibliothèques de fonctions regroupées dans un ou plusieurs fichiers, et dont le chargement est effectué par le système d'exploitation, dynamiquement, au démarrage de l'application. Cette implémentation prend la forme de fichiers *so* sous *Linux* ou encore de fichiers *DLL* sous *Windows*.

Cette organisation présente un avantage non négligeable : celui de permettre une mise à jour simple et efficace, mise à jour capable d'agrémenter les applications utilisant ces services de fonctionnalités supplémentaires, ou encore permettant de résoudre des problèmes de sécurité connus.

Pour tous les services habituels, cette architecture représente ce que l'on peut imaginer de mieux ; par contre, en ce qui concerne les services cryptographiques, l'organisation centralisée peut devenir extrêmement néfaste. La bibliothèque de fonctions cryptographiques représente dans ce cas le talon d'Achille de toutes les applications. En effet, les développeurs donnent implicitement toute leur confiance à cette bibliothèque pour ce qui concerne la sécurité de leurs applications. Imaginons alors qu'elle vienne à être corrompue d'une manière ou d'une autre par un programme extérieur, et qu'elle en arrive à fournir des informations relatives aux clés privées utilisées par les utilisateurs de la machine, ou encore, qu'elle fournisse à l'extérieur des informations sur le texte clair à l'origine des cryptogrammes qu'elle émet. Aucune des applications l'utilisant ne serait en mesure de

déterminer ce défaut puisque l'application en elle-même n'est pas modifiée ; la corruption de la bibliothèque cryptographique impliquerait alors un effondrement de toute la sécurité proposée par l'application.

L'aspect centralisé se retourne ici contre les applications sécurisées, puisqu'une seule modification risque de donner lieu à de graves conséquences pour l'ensemble des applications.

6.4 Application pratique

Afin d'illustrer nos propos, nous avons écrit un programme s'attaquant à la bibliothèque cryptographique d'un système d'exploitation grand public : *Windows XP*. Cette attaque n'est en rien spécifique à ce système d'exploitation, et aurait pu être menée sur un système tel que *Linux* ou *Mac OS*.

6.4.1 Aperçu de la CryptoAPI

La *CryptoAPI* est une bibliothèque de fonctions permettant aux utilisateurs d'effectuer des opérations cryptographiques telles que le chiffrement, ou encore la signature de messages. Cette bibliothèque est très intéressante du point du vue du programmeur d'applications, dans le sens où il n'a pas à se soucier de comment introduire une part de sécurité dans l'échange de messages dans son application. De plus, le fait de centraliser les primitives cryptographiques permet aussi de pouvoir agrémenter des applications existantes de nouveaux protocoles de chiffrement, et ce, sans devoir modifier quoi que ce soit dans le code de l'application.

La CryptoAPI est organisée sous forme arborescente ; chaque méthode de chiffrement se voit implémentée dans un *fournisseur de services cryptographiques* (*Cryptographic Service Provider*, ou *CSP*), qui est alors attaché à la CryptoAPI elle même. Notre première tâche fut donc de déterminer précisement où se situaient les fonctions de génération de clés pour les algorithmes de chiffrement *RSA*, *DES* ou encore *AES*, algorithmes les plus couramment utilisés aujourd'hui.

6.4.2 Organisation de la CryptoAPI

Les CSP sont implémentés sous forme de *fichiers DLL* (*Dynamically Linked Library*) qui sont, comme leur nom l'indique, des bibliothèques de fonctions *chargées dynamiquement*.

Notre tâche sera donc d'identifier avec précision le ou les fichiers DLL chargés lors d'opération de création de clés.

Pour ce faire, nous avons écrit un petit programme faisant appel à la CryptoAPI afin de chiffrer un texte quelconque grâce à l'algorithme *DES*, avec une clé de session construite à la volée. A l'aide d'un outil de surveillance système [63] nous pouvons alors déterminer quelles sont les DLLs chargées lors de l'utilisation de notre programme.

Parmi celles-ci, nous trouvons **ADVAPI32.DLL** et **RSAENH.DLL**. La première DLL est en fait une couche logicielle représentant la CryptoAPI telle qu'un programmeur peut l'utiliser. La seconde est chargée par **ADVAPI32.DLL** dans le but d'obtenir les fonctions qui sont réellement utilisées pour créer les clés et chiffrer les messages.

6.4.3 Analyse des DLLs

L'étape suivante fut le désassemblage partiel de **RSAENH.DLL** (*ingénierie inverse*, ou reverse engineering). Grâce à la documentation officielle du système d'exploitation, nous apprenons que toutes les DLLs utilisées comme fournisseur de services cryptographique dans la CryptoAPI se doivent d'offrir certaines fonctions comme *CPGenRandom*, *CPGenKey* ou encore *CPEncrypt*. Nous décidons alors de désassembler la DLL **RSAENH.DLL** à la recherche de ces fonctions.

L'étude approfondie de la fonction *CPGenRandom* nous apprend que la génération de pseudo-aléa est dépendante d'une unique fonction système nommée *SystemFunction036*, importée depuis **ADVAPI32.DLL**. Le résultat de cet appel est ensuite haché grâce à *SHA-1*.

En désassemblant la fonction *CPGenKey*, nous remarquons alors que la création des clés *DES/AES* passe aussi pas ces mêmes fonctions de pseudo-aléa. Par conséquent, le simple fait de modifier la fonction d'aléa aura un impact sur la création des clés pour ces deux algorithmes de chiffrement. Il s'agit donc bien ici d'une faiblesse, du point de vue génie logiciel, qui peut se révéler lourde de conséquences, comme nous allons le montrer.

6.4.4 Création d'un patch

Si l'on souhaite modifier le code de la fonction de pseudo-aléa de la CryptoAPI, deux solutions s'offrent à nous :

- modifier le ou les fichier(s) DLL sur le disque, ce qui aurait pour conséquence de charger en mémoire du code corrompu, et de forcer le système à exécuter notre code ;
- modifier directement la projection en mémoire du fichier DLL.

La première méthode n'a pas été retenue du fait de sa relative inélégance. De plus, nous aurions eu à contourner une protection système qui vérifie que les fichiers importants ne sont pas modifiés pendant l'exécution d'un processus (protection *SFC*, ou *System File Checker*).

Pour ce qui concerne la deuxième méthode, le problème majeur fut de trouver une méthode permettant d'écrire en mémoire sans devoir subir un refus de la part du système. La technique utilisée repose sur une faille dans le système de protection de la mémoire de Windows XP.

6.4.5 Présentation de la faille

Les *Windows Objects* sont des fichiers virtuels permettant de piloter des périphériques, ou des objets systèmes abstraits tels que des verrous. Ce type de fichier est très largement

utilisé dans la plupart des systèmes d'exploitations actuels.

L'idée de départ est d'utiliser ces Windows Objects, et en particulier l'objet nommé *\Device\PhysicalMemory* pour modifier des octets en mémoire. Bien entendu, les Windows Objects sont soumis à des droits d'accès, et pour ce qui concerne l'objet qui nous intéresse, seul l'utilisateur *SYSTEM* à le droit de l'utiliser en écriture (et donc de modifier la mémoire comme bon lui semble). Nous allons utiliser le fait que, sous Windows, la plupart des utilisateurs utilisent au quotidien un compte de type *administrateur*. Cette pratique est très dangereuse à de nombreux point de vue, mais il s'agit de la configuration par défaut du système, et très peu de gens sont réellement mis en garde contre cela.

Il existe un problème très important dans la gestion des droits sous Windows : en effet, les comptes administrateurs sont des comptes tout puissant, capable de passer outre toutes les restrictions. En effet, ces comptes ont, par exemple, le droit de changer les droits d'accès de tout fichier, même ceux qui ne leur appartiennent pas. Nous allons utiliser ceci pour modifier les droits d'accès au fichiers virtuels qui nous intéressent afin de nous octroyer les droits d'écriture.

6.4.6 Transformation d'une adresse virtuelle en adresse physique

L'évolution des logiciels a amené une consommation de plus en plus importante de mémoire vive. Cette ressource est particulièrement onéreuse, comparativement aux solutions de stockage de masse. Afin de palier le manque de mémoire vive, les systèmes d'exploitation implémentent un mécanisme de mémoire virtuelle : il s'agit de simuler une quantité de mémoire vive importante, en effectuant, de manière transparente pour l'utilisateur, des échanges permanents entre la mémoire vive réelle et les espaces de stockage de masse comme les disques durs.

Ainsi, tout programme s'exécutant n'a accès qu'à des adresses virtuelles. L'objet *\Device\PhysicalMemory* n'accédant pour sa part qu'à des adresses physiques (réelles), il nous faut trouver un moyen de forcer la présence en mémoire physique des données que l'on souhaite modifier, et trouver un moyen de convertir une adresse virtuelle en adresse physique.

Il existe une fonction qui permet de répondre à nos besoins : la fonction *MmGetPhysicalAddress*. Malheureusement, nous ne disposons pas de droits suffisants pour pouvoir l'utiliser.

6.4.7 Organisation des droits grâce aux RING

Toute tâche exécutée sur le processeur se voit contrôlée par un système de *RING* permettant de savoir quels sont les droits du processus quant à la possibilité de lire ou d'écrire à un endroit particulier de la mémoire, ou encore quant à la possibilité d'utiliser certaines instructions processeur particulières. Il existe, sur les processeurs de type *x86*, un total de 4 RINGs : RING0 est celui ayant le plus de droit (ce RING est offert par le processeur au premier processus activant le mode protégé du processeur), tandis que le RING3 est celui accueillant les processus utilisateur (RING ayant le moins de droits).

Lorsqu'un processus tente d'effectuer une opération non permise par son rang, une exception système est déclenchée, et un gestionnaire d'erreurs système est alors exécuté afin de traiter l'événement.

6.4.8 Analyse de *MmGetPhysicalAddress*

La fonction *MmGetPhysicalAddress* permet de transformer une adresse virtuelle en une adresse physique. Pour ce faire, elle a besoin d'avoir un accès à une table particulière renseignée par le système d'exploitation, dont l'adresse virtuelle est `0x3FD00000`. Malheureusement, cette table n'est pas accessible aux processus tournant en RING3. Il est donc nécessaire d'atteindre le niveau de privilège du RING0 pour pouvoir utiliser cette fonction.

6.4.9 Acquisition du RING0

Nous allons modifier le premier vecteur de l'*IDT* (*Interrupt Descriptor Table*) ; il s'agit d'une table contenant des pointeurs de fonctions que le système utilise lorsqu'une exception matérielle ou logicielle est déclenchée. Ainsi, par exemple, lorsqu'une division par zéro se produit dans un programme, l'exception numéro 0 est levée, et la fonction pointée par la première adresse (vecteur 0) de l'IDT est exécutée.

Si l'on modifie cette table afin que ce vecteur pointe sur une fonction nous appartenant (il est facile d'obtenir l'adresse physique de l'IDT grâce à une instruction spéciale du processeur : *sidt*), dès qu'une division par zéro se produira, c'est notre code qui sera exécuté, mais avec les privilèges du RING0. Une fois le vecteur installé, il suffira alors de déclencher *volontairement* une division par zéro pour que notre code soit exécuté avec les pleins pouvoirs.

6.4.10 Modification de la CryptoAPI

Une fois le problème de l'écriture en mémoire virtuelle réglé, nous pouvons modifier certains octets en mémoire afin d'injecter dans le code de la CryptoAPI, le code d'un générateur pseudo-aléatoire de très faible entropie. Il existe une dernière protection vérifiant l'intégrité des données en mémoire grâce à une somme de contrôle, mais elle est très facilement contournable en modifiant une seule instruction, et ne nécessite donc pas de réelles explications.

Suite à nos manipulations, nous avons dorénavant une bibliothèque cryptographique totalement fonctionnelle, mais dont le générateur pseudo-aléatoire utilisé pour la création des clés est très faible. Nous allons voir comment exploiter cette nouvelle faille.

6.4.11 Application pratique du patch

Afin de démontrer la dangerosité de la centralisation des fonctions cryptographiques, nous allons maintenant exploiter le générateur que nous avons injecté dans le système.

Pour cela, penchons nous sur un scénario d'envoi de message chiffrés, tels que proposés par les logiciels de messagerie comme *Outlook*. Ces logiciels utilisent le standard PKCS#7 [67].

L'envoi d'un message électronique chiffré s'effectue ainsi : on considère qu'Alice et Bob souhaitent échanger un message. Ils disposent chacun de leur couple clé publique/clé privée, et on suppose qu'Alice connaît la clé publique de Bob. Les algorithmes de chiffrement asymétrique étant calculatoirement très lourds, on leur préfère en général des algorithmes de chiffrement à clé secrète, par exemple par flot.

Le standard fonctionne alors ainsi :

- une *clé de session* K_s pseudo-aléatoire est générée par Alice,
- cette clé est chiffré grâce à un algorithme comme RSA, en utilisant la clé publique de Bob pour obtenir $RSA_{Bob}(K_s)$,
- le message à transmettre M est chiffré grâce à un algorithme comme RC4, et en utilisant la clé K_s afin d'obtenir $RC4_{K_s}(M)$,
- le couple $(RSA_{Bob}(K_s), RC4_{K_s}(M))$ est transmis à Bob,
- Bob peut déchiffrer $RSA_{Bob}(K_s)$ grâce à sa clé privée, et ainsi retrouver K_s la clé de session,
- Bob déchiffre enfin le message M à partir de la connaissance de $RC4_{K_s}(M)$ et de K_s.

Si l'on suppose qu'Alice utilise un ordinateur sur lequel le patch présenté a été exécuté, le clé de session qu'elle va créer va être choisie dans un espace de clé fortement réduit, puisque cette clé aura été construite à partir du générateur que nous avons injecté dans la bibliothèque cryptographique. L'effet pervers du système est que même si Alice observe ce qui résulte du processus de chiffrement de son message, elle ne peut pas s'apercevoir du problème, puisque le message est bel et bien chiffré correctement suivant le protocole.

Pour un attaquant écoutant le trafic réseau entre Alice et Bob, il est alors possible de récupérer le message chiffré. L'espace des clés de session étant très limité, il lui suffira alors de tester toutes les clés possibles, et ainsi de retrouver celle qui à été utilisée pour cette échange. Il n'est pas nécessaire alors d'attaquer le système de chiffrement asymétrique pour retrouver la clé.

Un point très intéressant dans cette attaque est qu'elle ne dépend pas des logiciels qu'Alice aura choisi pour transmettre son message. En effet, la plupart des logiciels de messagerie connus utilisent pour leurs échanges sécurisés la bibliothèque cryptographique offerte par le système d'exploitation.

6.5 Conclusion

Ce chapitre est destiné à mettre en évidence la réelle dangerosité de l'utilisation de bibliothèques cryptographiques centralisées. Cette organisation permet en effet une attaque extrêmement ciblée du système, qui résulte en un effondrement global de la sécurité des applications. Nous avons été en mesure de construire un programme capable de corrompre la sécurité d'un système d'exploitation grand public en injectant un générateur pseudo-aléatoire biaisé [80], et de manière totalement transparente pour l'utilisateur, ainsi que pour les applications elles-mêmes. Ce programme nous a ainsi permis de montrer les

conséquences réelles d'une telle attaque, c'est-à-dire, la possibilité de déchiffrer au vol n'importe quelle conversation sécurisée.

Afin d'éviter ce genre de problèmes, nous ne pouvons que préconiser, pour les applications critiques, l'emploi de bibliothèques statiques dont le code est directement intégré dans l'application cible, ou encore de laisser aux programmeurs d'applications, la charge de créer eux-mêmes leur propres primitives cryptographiques. La sécurité des applications est en effet trop importante pour être déléguée à une autre application sur laquelle le programmeur n'a aucun contrôle.

De plus, cette méthode n'aurait pas pu être utilisée si le compte utilisateur avait eu moins de privilèges. Or, le seul compte créé lors de l'installation par défaut de ce système d'exploitation est un compte ayant les pleins pouvoirs. Malheureusement, la plupart des utilisateurs réguliers de ce système ne prennent pas la peine de créer un compte utilisateur simple, à cause des contraintes qui leur seront imposées par la suite. Il s'agit là, pourtant, d'un problème récurrent, à l'origine de la très rapide et très large diffusion des *virus* et autres *chevaux de Troie* : un système utilisé en mode utilisateur, même sans antivirus, a beaucoup moins de chance d'être contaminé par un virus informatique qu'un système utilisé avec des droits administrateur sur lequel un logiciel antivirus est installé. La création d'un compte sans privilèges aurait très largement réduit les risques de l'attaque que nous présentons.

Conclusion

Cette étude des systèmes de chiffrement à flot à permis de montrer la très grande diversité des primitives habituellement utilisées pour produire du pseudo-aléa, mais aussi le très grand nombre de menances les concernant.

La multiplication du nombre d'attaques concernant les systèmes de chiffrement à flot nous a conduit à nous intéresser à de nouvelles primitives de génération de pseudo-aléa. Dans cette optique, les suites de de Bruijn nous ont semblé être très intéressantes : elles disposent en effet de très bonnes propriétés statistiques, tout en restant exprimable de manière concise à partir d'une quantité très faible de bits.

Nos efforts se sont alors principalement concentrés sur les méthodes de constructions de telles suites, tout en gardant à l'esprit notre but premier, à savoir la nécessité de pouvoir utiliser ces algorithmes sur des machines ayant des ressources en mémoire, en puissance de calcul, mais aussi en autonomie très limitées.

Le système de chiffrement que je propose dans cette thèse n'en est qu'à ses débuts. Il souffre à l'heure actuelle, par rapport aux autres systèmes de chiffrement à flot existants, d'un manque de vitesse. Mais, bien qu'imparfait, celui-ci présente l'intérêt non négligeable de fournir une très grande résistance aux attaques connues. De plus, ce système ouvre la voie à de nouvelles possibilités d'utilisations concernant les registres à décalage à rétroaction non linéaire.

Le problème de vitesse est essentiellement dû au fait que, bien que la plupart des calculs soient indépendants les uns des autres, nous n'avons pas d'autres choix que de les séquencialiser lors d'une implémentation logicielle. Ainsi, une mise en application matérielle, telle qu'envisagée dans le chapitre 5, permettrait de tirer parti de cette indépendance entre les différents calculs, et ainsi obtenir des vitesses de chiffrement rivalisant avec la plupart des systèmes à flot actuels. Cette implémentation matérielle a été discutée au sein de l'équipe *RD2P*, dont je fais partie, afin d'envisager une mise en application sur FPGA dans un avenir proche.

Parallèlement, il reste encore plusieurs voies à étudier concernant la sécurité du système de chiffrement. Par exemple, il serait bon de pouvoir quantifier exactement le gain que procure le fait que les fonctions de rétroactions des registres ne soient pas fixées, mais calculées en fonction d'une partie de la clé. Mon idée sur ce sujet, que je ne peux pour l'instant pas démontrer sans une étude rigoureuse, me laisse à penser que le fait de consacrer une partie de l'initialisation à la construction des fonctions de rétroaction est une bonne chose : prenons l'exemple des attaques par corrélations, où l'on dispose d'une initialisation de n bits, de r registres, et où k des n bits d'initialisation servent à

décrire les fonctions de rétroaction. Dans ce cas, l'attaque doit être menée sur chacun des 2^k systèmes différents, disposant de registres d'une taille moyenne de $\frac{n-k}{r}$ bits, alors que si les n bits sont tous consacrés à l'initialisation des registres, l'attaque aurait été à mener sur un seul système ayant des registres d'une taille moyenne de $\frac{n}{r}$ bits. Le fait de consacrer k bits de l'initialisation à cette construction permet donc approximativement de multiplier le coût des attaques évoquées dans le chapitre 2 par 2^k.

C'est dans ces directions que mes futurs travaux se dirigeront, afin de pouvoir obtenir, dans un avenir relativement proche, un système de chiffrement à flot performant, offrant un haut niveau de sécurité vis à vis des attaques connues, et surtout, utilisant des primitives cryptographiques novatrices.

Cette thèse m'aura aussi permis de travailler dans le domaine des attaques d'implémentations logicielles de sytèmes de chiffrement. Moyennant un effort non négligeable d'*ingénierie inverse* (*reverse engineering*) sur la bibliothèque cryptographique du système d'exploitation *Windows XP*, nous avons été capable d'y injecter un générateur pseudo-aléatoire de très faible entropie, démontrant ainsi la dangerosité des architectures centralisés auxquelles les développeurs de logiciels doivent déléguer toute leur confiance. De ce fait, si la bibliothèque au cœur du système vient à être compromise, aucun logiciel ne peut s'en rendre compte. Cette démonstration s'est inscrite dans un cadre plus vaste d'étude de dissimulation de trappes dans les systèmes cryptographiques [80].

Annexe A

Définitions et notations

Je présente dans ce chapitre les notations qui sont utilisées tout au long de ce document. Il s'agit ici de définir avec le lecteur un vocabulaire commun permettant une meilleure compréhension.

A.1 Le corps $\mathbb{F}_2$

Définissons tout d'abord le cadre de travail : $\mathbb{F}_2$ est le corps à deux éléments, généralement notés $\{0, 1\}$, sur lequel sont définis deux opérations élémentaires notées $\oplus$ et $\odot$. Ces deux opérations sont les équivalents de l'addition et de la multiplication dans le corps à deux éléments, mais dont les résultats sont calculés modulo 2. Il arrivera donc par la suite, par abus de notation, que le symbole $\oplus$ soit remplacé par le symbole $+$, tout comme le symbole $\odot$ pourra être remplacé par le symbole ., voir même tout simplement retiré si aucune ambiguïté n'est possible (à l'instar de la multiplication usuelle).

Nous pouvons définir ces deux opérations par leur *table de vérité*, ce qui correspond à faire la liste exhaustive de toutes les combinaisons possibles de valeurs :

$\oplus$	0	1
0	0	1
1	1	0

$\odot$	0	1
0	0	0
1	0	1

Nous pourrons remarquer que ces fonctions sont équivalentes aux primitives classiques que l'on rencontre sur les microprocesseurs, à savoir le `OU-EXCLUSIF` pour l'opérateur $\oplus$ et le `ET` pour l'opérateur $\odot$.

On définit aussi l'espace vectoriel $\mathbb{F}_2^n$. L'addition et la multiplication peuvent être étendues à cet espace, en considérant qu'il suffit d'effectuer l'opération pour chaqun des bits des vecteurs. Ainsi, si $\mathrm{x} = (x_{n-1}, x_{n-2}, \ldots, x_0) \in \mathbb{F}_2^n$ et $\mathrm{y} = (y_{n-1}, y_{n-2}, \ldots, y_0) \in \mathbb{F}_2^n$, alors :

$$\mathrm{x} \oplus \mathrm{y} = (x_{n-1} \oplus y_{n-1}, x_{n-2} \oplus y_{n-2}, \ldots, x_0 \oplus y_0)$$

$$\mathrm{x} \odot \mathrm{y} = (x_{n-1} \odot y_{n-1}, x_{n-2} \odot y_{n-2}, \ldots, x_0 \odot y_0)$$

A.2 Les fonctions booléennes

Une *fonction booléenne* est une fonction prenant ses valeurs dans $\mathbb{F}_2^n$ et qui y associe une valeur de $\mathbb{F}_2^m$. En d'autres termes, il s'agit d'une fonction travaillant sur un vecteur de n bits, et qui associe à ce vecteur un vecteur de m bits. Nous nous intéresserons essentiellement au cas $m = 1$.

Nous rencontrerons beaucoup de ces fonctions tout au long de ce document. Il existe plusieurs moyens de les représenter qui sont tous totalement équivalents.

A.2.1 Table de vérité

Il est possible de représenter toute fonction booléenne par sa *table de vérité*, que l'on assimilera à un vecteur de $\mathbb{F}_2^{2^n}$, noté $\vec{f}$. Par convention, la i^{e} composante de ce vecteur, en numérotant les indices à partir de 0, correspond à la valeur de $f(u)$, où u est le codage binaire sur n bits de l'entier i. Ce type de représentation est généralement totalement inutilisable en pratique. En effet, le nombre de valeurs contenues dans une table de vérité croît exponentiellement en fonction du nombre de variables de f, ainsi si la fonction est définie sur $\mathbb{F}_2^n$, alors la table de vérité contiendra 2^n valeurs.

A.2.2 Forme normale disjonctive

La *forme normale disjonctive* d'une fonction est une autre représentation de la fonction utilisant trois opérateurs logiques : le `OU`, le `ET` et le `NON`.

Pour la construire, on extrait de sa table de vérité les entrées de $\mathbb{F}_2^n$ pour lesquelles elle est non nulle, et pour chacune de ces entrées, on associe une conjonction des littéraux.

Considérons l'exemple de la section précédente.

$$f = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

La liste des vecteurs de $\mathbb{F}_2^3$ pour lesquels f est non nulle est donc

$$(0,0,1), (0,1,1), (1,0,1), (1,1,0), (1,1,1).$$

À chacun de ces vecteurs, nous associons une conjonction, qui prend la valeur 1 uniquement pour le vecteur considéré :

$$\overline{x_2}\,\overline{x_1}\,x_0,\ \overline{x_2}\,x_1\,x_0,\ x_2\,\overline{x_1}\,x_0,\ x_2\,x_1\,\overline{x_0},\ x_2\,x_1\,x_0.$$

Notre fonction f s'écrit donc comme la disjonction de ces conjonctions :

$$f(x_2, x_1, x_0) = \overline{x_2}\,\overline{x_1}\,x_0 \vee \overline{x_2}\,x_1\,x_0 \vee x_2\,\overline{x_1}\,x_0 \vee x_2\,x_1\,\overline{x_0} \vee x_2\,x_1\,x_0.$$

Cette représentation est très intéressante lorsqu'il s'agit de représenter une fonction *creuse*, c'est-à-dire, une fonction dont la table de vérité comporte une grande majorité de 0.

A.2.3 Forme normale algébrique

La forme normale algébrique (ou *FNA*) d'une fonction est assez similaire à la forme normale disjonctive. Nous allons cette fois-ci n'utiliser non plus trois, mais deux opérateurs logiques : les opérateurs `OU-EXCLUSIF` et `ET`. La forme normale algébrique est une représentation sous forme polynômiale de la fonction. On assimile l'ensemble des fonctions booléennes à l'anneau des polynomes à coefficients dans $\mathbb{F}_2$, où pour toute variable, on identifie x^2 à x.

Par exemple, si l'on considère la fonction f dont la table de vérité $\vec{f}$ est donnée par

$$\vec{f} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

son expression sous forme normale algébrique est donnée par

$$f(x_2, x_1, x_0) = x_0x_1x_2 \oplus x_1x_2 \oplus x_0.$$

A.3 Transformation de Walsh-Hadamard

Je présente dans le chapitre 2 les critères d'*immunité aux corrélations* et de *non linéarité* d'une fonction. Afin de déterminer les valeurs de ces critères, il n'est en général pas envisageable de tester de manière exhaustive la distance de la fonction à l'ensemble des fonctions linéaires, ou encore de tester l'équilibre de la fonction lorsque l'on fixe un certain nombre d'entrées.

Pour ce faire, on utilise généralement la *transformation de Walsh-Hadamard*. Cette transformation simple permet d'obtenir un grand nombre d'informations au sujet de la fonction étudiée. La transformée de Walsh-Hadamard d'une fonction f correspond au calcul de la *transformée de Fourrier* de la *fonction signe* de f. C'est une fonction notée $\widehat{\chi_f} : \mathbb{F}_2^n \mapsto \mathbb{C}$, et son expression est donnée par :

$$\forall u \in \mathbb{F}_2^n,\ \widehat{\chi_f}(u) = \sum_{v \in \mathbb{F}_2^n} (-1)^{u \cdot v \oplus f(u)},$$

où $u \cdot v$ représente le produit scalaire usuel de u et v. La transformation est inversible, et on a

$$\widehat{\chi_f}^{-1} = \frac{1}{2^n}\widehat{\chi_f}.$$

La transformée de Walsh d'une fonction f permet de caractériser un certain nombre de propriétés de cette fonction.

Équilibre

La fonction f est *équilibrée* si sa table de vérité contient autant de 0 que de 1. L'équilibre de la fonction est lié à la valeur de $\widehat{\chi_f}(0)$:

$$f \text{ est équilibrée} \iff \widehat{\chi_f}(0) = 0.$$

Plus généralement, $\widehat{\chi_f}(0)$ est lié au *poids* $\omega_H(f)$ de la fonction f, c'est-à-dire au nombre de composantes non nulles de sa table de vérité :

$$\omega_H(f) = 2^{n-1} - \frac{\widehat{\chi_f}(0)}{2}.$$

Non-linéarité

La *non-linéarité* de f, plus petite distance de Hamming de f aux fonctions affines, notée $nl(f)$ peut être déterminée par :

$$nl(f) = \frac{1}{2}\left(2^n - \max_{i=0,\ldots,2^n-1} |\widehat{\chi_f}(i)|\right).$$

Cette affirmation découle directement de l'expression des composantes de $\widehat{\chi_f}$.

Résilience

La fonction f est *t-résiliente* si, lorsque l'on fixe au plus t variables arbitrairement, la distribution des valeurs de f reste équilibrée. Ainsi, f est t-résiliente si :

$$\forall u \in [0..2^n - 1],\ \omega_H(u) \leq t,\ \widehat{\chi_f}(u) = 0.$$

On notera que les critères de non-linéarité nl, de résilience t et le degré d de la forme normale algébrique de f vérifie l'inégalité de Siegenthaler [70] :

$$t < nl - d.$$

Annexe B

Résultats des tests statistiques

B.1 Présentation des tests statistiques

Les tests présentés ici font partis de la batterie de test du programme Sts.

Frequency

Il s'agit du test le plus simple. Dans une suite aléatoire, tous les symboles de l'alphabet utilisés doivent apparaître avec la même fréquence. Il nous faut donc vérifier que les suites produites contiennent sensiblement autant de 0 que de 1.

Block frequency

Ce test constitue une extension du test précédent. Le principe est, cette fois-ci, de découper la suite en blocs de taille M, puis de vérifier que dans chaque bloc de M bits, la proportion de 0 et de 1 est sensiblement la même.

Runs

On définit un *run* comme étant une suite ininterrompue d'un même symbole. Ainsi, la suite 0001100010111101 est constituée de huit runs : 000, 11, 000, 1, 0, 1111, 0 et 1. La *taille d'un run* est le nombre de symboles qui le constitue. Ce test vérifie que le nombre de runs n'est pas trop élevé ou trop faible, et que la fréquence de changement de run n'est pas trop rapide ou trop lente.

Longest Runs of Ones

La suite est découpée en blocs de M symbols. Dans chacun de ces blocs, le test vérifie que la taille du plus long run de 1 est bien conforme à ce que l'on attend d'une suite aléatoire.

Rank

Le principe de ce test est de construire des matrices binaires aléaroires à partir de la suite, puis d'en déterminer le rang. Il s'agit de découvrir s'il existe des relations linéaires entre des blocs issues de la suite. Les matrices construites sont de taille 32×32. Le test classe les matrices en trois catégories : les matrices de rang maximum (32), celles de rang immédiatement inférieur (31) et enfin, celles dont le rang est inférieur à 31. Il est procédé ensuite à une comparaison avec le modèle théorique de répartition des rang de matrices binaires aléatoires.

Spectral DFT

Le but de ce test est de rechercher des motifs apparaissant de manière périodique dans la suite. Pour cela, le test calcule le spectre de Fourier de la suite, et vérifie qu'il n'y a pas de pic pour un motif particulier.

Overlapping et Non-overlapping templates

Il s'agit ici de rechercher un ensemble de motifs pré-définis de $2^{17} = 131\,072$ bits dans la suite. Si l'on considère que la suite est composée de n bits, les deux tests se déroulent ainsi, pour chaque motif :

1. on démarre à l'index $i = 0$
2. on compare les symboles $s_i, s_{i+1}, \ldots, s_{i+2^{17}-1}$ au motif recherché.
3. dans le cas d'une correspondance, on augmente un compteur de 1. S'il s'agit du test non recouvrant, on avance i de $2^{17} - 1$ bits.
4. i est avancé de 1, et si $i < n - 2^{17}$ on recommence à la deuxième étape.

Le compteur est ensuite comparé au modèle théorique.

Universal

Ce test tente de déterminer si une suite est facilement compressible sans perte d'information. En effet, une suite aléatoire possède un taux d'information très important, et n'est pas facilement compressible, alors que, par exemple, un texte structuré contient une grande redondance et est facilement compressible. Pour ce faire, le test mesure la distance qui sépare des motifs identiques dans la suite.

Linear complexity

Ce test est très important d'un point de vue cryptographique, puisqu'il s'agit d'appliquer l'attaque de Berlekamp-Massey [4, 50] sur la suite.

Pour cela, la suite de n bits est découpée en blocs de M bits (dans notre cas, $n = 4\,194\,304\,000$ et $M = 2000$). Pour chacun de ces blocs, on calcule alors la complexité linéaire grâce à l'algorithme de Berlekamp-Massey. On compare alors la complexité linéaire de chacun de ces blocs avec la valeur prévue par la théorie, c'est-à-dire, pour n

suffisamment grand, si n est pair, la complexité linéaire d'une suite aléatoire de n bits est de $\frac{n}{2} + \frac{2}{9}$, et si n est impair, la complexité est alors $\frac{n}{2} + \frac{5}{18}$ [51, 64].

Serial

Ce test vérifie que chaque motif de m bits apparaît dans la suite avec une fréquence proche de celle attendue par la théorie. Remarquons au passage que pour $m = 1$, ce test est équivalent au test de fréquence (Frequency test) décrit précédemment. Dans une suite aléatoire, la distribution des motifs de taille donnée doit être uniforme. On vérifie donc ici que tous les motifs de m bits ($m = 16$ dans notre cas) sont équiprobables.

Approximate entropy

Ce test est assez similaire au Serial test. Il vise à estimer le taux d'information (entropie) de la suite. Le taux d'information mesure en quelque sorte l'intérêt qu'apporte l'apparition d'un symbole dans une suite. Par exemple, si à chaque fois que le motif 00101 apparait dans la suite, il est suivi du bit 0, alors ce bit ne nous apporte aucune information. Si par contre, le bit 0 a autant de chance d'apparaître que le bit 1, alors l'information apportée par ce bit est importante.

Le test procède donc à une extraction de toutes les sous-suites de $m+1$ bits de la suite (se recouvrant) et regarde, pour chacune de ces sous-suites, si la probabilité d'apparition du bit 0 en tant que dernier bit de la sous-suite, sachant que l'on a les m premiers bits, est la même que celle du bit 1.

Cumulative sums

Il s'agit ici d'effectuer un parcours aléatoire d'une partie des n bits de la suite, et d'en calculer la somme. On vérifie ensuite que cette somme est proche de $n/2$.

Random excursion

Le principe de ce test est ici de remplacer dans la suite, les 0 par des -1, puis de construire une nouvelle suite $(S_n)_{n\geq 0} = 0, s_0, s_0 + s_1, s_0 + s_1 + s_2, \ldots$. Cette suite est découpée en cycles, un cycle étant une partie de $(S_n)_{n\geq 0}$ commençant et se terminant par un 0. La fréquence d'apparition des entiers $-4, -3, -2, -1, 1, 2, 3$ et 4 est alors testée pour chacun des cycles obtenus.

B.2 Résultats de la série de tests de Sts

Le couple vecteur d'initialisation/clé utilisé pour ces tests est :

IV = `90ffac6c01d0cdfe`
clé = `87a409dc3d703955ede68d58398b0b12`

Pour information, les premiers octets de pseudo-aléa construits par le générateur avec ce couple sont :

```
0b 50 9f 20 f3 87 23 97 35 17 51 45 62 82 ff b7
48 5d 21 9b f2 75 ac db 6b fd b8 b6 9c d2 22 30
93 45 f1 dc 94 c5 c5 f6 78 9f 2f 03 05 a4 1c 86
a4 71 85 a4 e8 e9 84 52 52 38 ab df 9c 7f 8d 49
```

Les tests ont été pratiqués sur un fichier de 500Mo (environ 4,2 milliards de bits). De ce fichier, le programme Sts extrait 1 000 suites de 1 000 000 de bits. Les paramètres utilisés pour les tests, et préconisés par la documentation fournie avec le programme, sont fournis dans le tableau B.1.

Nom du test	**Valeur du paramètre**
Block frequency	20 000
Overlapping Templaces	9
Non-overlapping Templates	9
Serial	16
Approximate Entropy	10
Linear Complexity	2 000
Universal	Block length : 7, steps : 1 280

TAB. B.1 – Paramètres des tests

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
89	106	103	100	101	108	97	98	95	103	0.972382	0.9890	frequency
100	85	113	87	106	93	111	95	112	98	0.399442	0.9870	block-frequency
97	97	102	100	111	97	103	95	111	87	0.854708	0.9900	cumulative-sums
91	109	92	94	84	106	121	88	106	109	0.183547	0.9860	cumulative-sums
103	89	97	104	99	93	106	118	96	95	0.733899	0.9900	runs
97	107	83	113	108	103	98	93	116	82	0.201189	0.9870	longest-run
103	109	111	105	102	97	103	85	94	91	0.739918	0.9870	rank
100	113	103	111	90	92	124	106	75	86	0.025535	0.9890	fft
113	104	100	97	93	94	103	94	108	94	0.894918	0.9880	nonperiodic-templates
91	119	96	109	108	85	90	87	115	100	0.152902	0.9890	nonperiodic-templates
89	90	105	103	83	116	102	95	112	105	0.352107	0.9860	nonperiodic-templates
108	93	104	98	100	103	97	88	121	88	0.455937	0.9920	nonperiodic-templates
100	108	99	107	93	123	77	89	96	108	0.114712	0.9880	nonperiodic-templates
89	96	88	105	94	111	100	104	101	112	0.715679	0.9920	nonperiodic-templates
92	86	107	101	105	101	102	107	99	100	0.917870	0.9920	nonperiodic-templates
97	102	87	108	90	100	102	88	106	120	0.410055	0.9940	nonperiodic-templates
95	111	112	99	86	102	101	103	94	97	0.792508	0.9860	nonperiodic-templates
105	102	111	84	112	89	97	107	107	86	0.371941	0.9880	nonperiodic-templates
96	86	99	110	95	100	95	94	116	109	0.599693	0.9860	nonperiodic-templates
91	116	111	113	101	83	95	102	103	85	0.224821	0.9930	nonperiodic-templates
111	100	93	117	102	86	99	81	109	102	0.271619	0.9870	nonperiodic-templates
75	94	132	116	90	98	104	86	100	105	0.006613	0.9920	nonperiodic-templates
116	95	111	105	96	99	104	89	105	80	0.345650	0.9870	nonperiodic-templates
120	105	99	92	105	105	108	96	84	86	0.295391	0.9890	nonperiodic-templates

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
99	96	101	131	95	99	89	92	81	117	0.030806	0.9870	nonperiodic-templates
98	95	103	107	107	95	112	86	95	102	0.807412	0.9910	nonperiodic-templates
103	90	122	117	95	78	100	100	84	111	0.039073	0.9930	nonperiodic-templates
113	88	97	126	98	89	94	91	115	89	0.069863	0.9870	nonperiodic-templates
101	115	112	89	85	99	108	81	115	95	0.125200	0.9940	nonperiodic-templates
99	101	88	110	96	103	98	93	103	109	0.901959	0.9900	nonperiodic-templates
88	105	102	117	103	90	90	85	110	110	0.278461	0.9890	nonperiodic-templates
85	95	115	100	112	95	99	97	108	94	0.581082	0.9880	nonperiodic-templates
91	93	112	93	104	105	112	93	96	101	0.765632	0.9910	nonperiodic-templates
96	80	85	106	121	106	125	101	95	85	0.016149	0.9900	nonperiodic-templates
99	99	93	106	89	103	101	104	95	111	0.924076	0.9930	nonperiodic-templates
104	112	110	103	94	99	102	87	86	103	0.653773	0.9890	nonperiodic-templates
103	106	103	103	97	100	103	103	88	94	0.975012	0.9960	nonperiodic-templates
100	87	104	103	95	107	96	111	93	104	0.859637	0.9850	nonperiodic-templates
99	108	99	107	84	115	94	104	106	84	0.401199	0.9890	nonperiodic-templates
106	111	91	95	93	111	96	112	95	90	0.618385	0.9880	nonperiodic-templates
106	87	115	112	99	110	95	98	93	85	0.368587	0.9920	nonperiodic-templates
110	104	96	97	102	85	114	109	90	93	0.538182	0.9920	nonperiodic-templates
106	89	100	100	100	95	98	114	102	96	0.910091	0.9900	nonperiodic-templates
112	118	94	98	94	89	101	108	98	88	0.457825	0.9910	nonperiodic-templates
103	101	117	96	104	105	97	89	90	98	0.749884	0.9850	nonperiodic-templates
113	104	99	76	106	102	89	112	96	103	0.281232	0.9880	nonperiodic-templates
90	104	99	77	95	108	130	96	94	107	0.043368	0.9900	nonperiodic-templates
93	100	98	93	96	104	102	103	113	98	0.955835	0.9890	nonperiodic-templates
90	112	100	80	99	108	108	114	92	97	0.317565	0.9960	nonperiodic-templates
117	98	95	101	106	90	114	88	98	93	0.486588	0.9900	nonperiodic-templates
90	100	94	90	104	103	100	105	95	119	0.666245	0.9920	nonperiodic-templates
110	99	91	119	95	90	116	99	102	79	0.133404	0.9830	nonperiodic-templates
102	97	96	96	94	84	110	116	93	112	0.450297	0.9930	nonperiodic-templates
104	93	118	80	101	93	116	88	98	109	0.152044	0.9930	nonperiodic-templates
104	99	95	80	110	87	102	113	113	97	0.302657	0.9910	nonperiodic-templates
117	93	114	112	91	89	94	107	94	89	0.260930	0.9850	nonperiodic-templates
102	88	111	89	97	125	95	88	112	93	0.127393	0.9920	nonperiodic-templates
97	100	103	89	96	93	118	104	107	93	0.697257	0.9890	nonperiodic-templates
103	103	87	105	99	93	98	107	103	102	0.952152	0.9870	nonperiodic-templates
94	94	88	91	129	96	107	95	100	106	0.179584	0.9870	nonperiodic-templates
89	106	101	101	101	79	112	98	105	108	0.496351	0.9930	nonperiodic-templates
104	97	97	100	97	98	112	99	98	98	0.991468	0.9910	nonperiodic-templates
105	96	96	101	109	112	107	84	99	91	0.668321	0.9890	nonperiodic-templates
99	94	107	95	106	94	95	107	102	101	0.977480	0.9910	nonperiodic-templates
87	102	115	95	90	111	101	110	99	90	0.488534	0.9940	nonperiodic-templates
101	90	89	118	107	99	91	101	103	101	0.649612	0.9920	nonperiodic-templates
99	98	109	79	85	103	103	104	116	104	0.305599	0.9880	nonperiodic-templates
125	86	100	102	86	95	106	95	105	100	0.254411	0.9840	nonperiodic-templates
98	94	114	83	93	115	98	114	92	99	0.301194	0.9920	nonperiodic-templates
109	106	114	107	97	85	87	84	99	112	0.233162	0.9920	nonperiodic-templates
114	100	100	73	99	116	97	95	93	113	0.110734	0.9940	nonperiodic-templates
85	108	100	115	98	101	95	93	102	103	0.733899	0.9920	nonperiodic-templates
100	94	93	109	118	93	87	98	111	97	0.492436	0.9870	nonperiodic-templates
103	101	98	108	95	108	106	91	106	84	0.763677	0.9900	nonperiodic-templates
93	108	96	88	114	97	115	81	101	107	0.266235	0.9930	nonperiodic-templates
98	110	104	86	85	102	97	99	111	108	0.595549	0.9920	nonperiodic-templates
91	102	109	118	87	109	102	96	82	104	0.275709	0.9960	nonperiodic-templates
104	82	112	95	103	108	98	95	100	103	0.719747	0.9940	nonperiodic-templates
94	104	118	95	97	94	82	94	105	117	0.262249	0.9960	nonperiodic-templates
95	103	94	105	99	116	91	108	93	96	0.777265	0.9890	nonperiodic-templates
113	95	110	88	82	93	106	98	112	103	0.347257	0.9900	nonperiodic-templates
103	118	109	87	119	90	89	96	104	85	0.114712	0.9920	nonperiodic-templates
113	104	99	98	93	94	104	93	108	94	0.883171	0.9880	nonperiodic-templates
109	98	82	92	94	102	122	109	106	86	0.158133	0.9900	nonperiodic-templates
100	85	92	105	102	114	93	106	96	107	0.674543	0.9940	nonperiodic-templates
93	100	96	123	110	97	93	97	102	89	0.450297	0.9930	nonperiodic-templates
122	105	94	84	104	95	98	93	95	110	0.334538	0.9920	nonperiodic-templates
103	97	103	99	101	89	97	106	107	98	0.981417	0.9880	nonperiodic-templates
103	94	101	93	82	109	113	103	111	91	0.455937	0.9960	nonperiodic-templates
92	104	101	97	101	109	93	119	84	100	0.496351	0.9940	nonperiodic-templates
110	102	103	90	96	129	84	83	100	103	0.062036	0.9920	nonperiodic-templates
95	103	96	86	120	114	100	99	79	108	0.142062	0.9910	nonperiodic-templates
110	110	108	89	95	84	103	101	93	107	0.560545	0.9900	nonperiodic-templates
104	99	88	83	101	120	115	90	86	114	0.073872	0.9930	nonperiodic-templates
88	107	96	104	112	109	92	102	97	93	0.763677	0.9920	nonperiodic-templates
108	101	97	102	105	92	101	99	93	102	0.987492	0.9950	nonperiodic-templates
119	87	94	104	101	80	96	103	106	110	0.246750	0.9910	nonperiodic-templates
96	101	97	109	97	90	84	119	100	107	0.454053	0.9890	nonperiodic-templates
99	99	114	107	70	105	92	98	111	105	0.127393	0.9850	nonperiodic-templates
97	97	82	96	103	101	118	90	105	111	0.402962	0.9930	nonperiodic-templates
94	90	94	117	98	117	109	96	89	96	0.360287	0.9920	nonperiodic-templates
108	96	86	107	95	107	98	103	102	98	0.897763	0.9860	nonperiodic-templates

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
102	89	91	102	100	103	105	87	116	105	0.643366	0.9930	nonperiodic-templates
84	116	120	90	114	99	106	82	103	86	0.038313	0.9890	nonperiodic-templates
102	94	110	96	101	102	99	92	97	107	0.970302	0.9890	nonperiodic-templates
106	111	105	103	85	92	102	82	101	113	0.368587	0.9890	nonperiodic-templates
98	100	87	109	96	98	120	99	101	92	0.595549	0.9930	nonperiodic-templates
86	86	111	107	106	119	105	83	96	101	0.167184	0.9970	nonperiodic-templates
102	112	105	103	89	93	109	100	83	104	0.597620	0.9840	nonperiodic-templates
93	95	107	99	92	95	111	115	98	95	0.751866	0.9920	nonperiodic-templates
98	96	101	105	95	80	103	91	127	104	0.159910	0.9870	nonperiodic-templates
95	92	92	113	113	87	94	118	108	88	0.197981	0.9870	nonperiodic-templates
101	96	99	86	81	102	107	119	101	108	0.308561	0.9880	nonperiodic-templates
92	121	98	104	91	97	88	110	91	108	0.347257	0.9910	nonperiodic-templates
104	103	107	80	105	110	92	107	98	94	0.583145	0.9920	nonperiodic-templates
82	104	101	105	93	116	103	83	114	99	0.233162	0.9890	nonperiodic-templates
111	101	97	92	96	99	106	104	96	98	0.970302	0.9860	nonperiodic-templates
106	91	103	107	95	95	111	107	87	98	0.771469	0.9890	nonperiodic-templates
102	96	93	94	105	101	97	121	88	103	0.601766	0.9950	nonperiodic-templates
94	90	112	100	87	110	106	101	89	111	0.506194	0.9900	nonperiodic-templates
86	103	102	99	93	81	104	115	113	104	0.314544	0.9940	nonperiodic-templates
105	103	109	95	90	106	90	102	89	111	0.717714	0.9880	nonperiodic-templates
123	97	97	89	100	96	96	90	103	109	0.446556	0.9900	nonperiodic-templates
88	93	101	97	94	113	98	95	117	104	0.593478	0.9890	nonperiodic-templates
120	104	90	100	104	87	87	107	107	94	0.347257	0.9940	nonperiodic-templates
89	108	107	91	105	117	90	87	102	104	0.420827	0.9940	nonperiodic-templates
105	119	90	96	105	85	102	104	97	97	0.544254	0.9880	nonperiodic-templates
107	93	109	103	91	96	108	104	82	107	0.597620	0.9930	nonperiodic-templates
90	100	88	114	105	97	110	90	99	107	0.612147	0.9910	nonperiodic-templates
112	88	110	93	103	96	93	107	106	92	0.678686	0.9920	nonperiodic-templates
75	111	97	115	93	104	102	104	108	91	0.207730	0.9920	nonperiodic-templates
96	97	103	97	107	106	89	103	107	95	0.950247	0.9950	nonperiodic-templates
94	96	102	98	104	112	98	101	96	99	0.982958	0.9930	nonperiodic-templates
83	102	111	109	99	90	102	109	104	91	0.556460	0.9940	nonperiodic-templates
97	115	94	103	99	89	90	106	105	102	0.773405	0.9900	nonperiodic-templates
92	102	97	93	103	91	125	104	106	87	0.302657	0.9870	nonperiodic-templates
100	88	115	116	103	103	96	95	87	97	0.473064	0.9900	nonperiodic-templates
104	110	103	103	87	100	97	98	88	110	0.779188	0.9910	nonperiodic-templates
114	108	94	111	79	95	103	115	93	88	0.158133	0.9870	nonperiodic-templates
88	115	91	100	94	96	94	112	113	97	0.474986	0.9890	nonperiodic-templates
81	118	109	97	91	100	97	110	91	106	0.288249	0.9890	nonperiodic-templates
101	87	112	95	103	96	91	105	110	100	0.769527	0.9940	nonperiodic-templates
99	84	99	101	104	110	90	104	114	95	0.624627	0.9920	nonperiodic-templates
99	99	105	88	111	109	100	86	107	96	0.705466	0.9880	nonperiodic-templates
84	90	89	102	100	94	122	104	98	117	0.158133	0.9910	nonperiodic-templates
95	106	116	88	104	116	100	88	105	82	0.199045	0.9890	nonperiodic-templates
101	96	98	97	116	95	102	100	97	98	0.952152	0.9880	nonperiodic-templates
90	113	107	100	94	106	102	90	101	97	0.830808	0.9950	nonperiodic-templates
99	101	107	98	99	108	96	106	90	96	0.968863	0.9900	nonperiodic-templates
103	95	104	107	114	101	91	92	88	105	0.729870	0.9880	nonperiodic-templates
93	109	108	103	83	113	102	108	91	90	0.428095	0.9890	nonperiodic-templates
102	115	92	102	105	103	87	84	105	105	0.528111	0.9870	nonperiodic-templates
91	95	103	95	104	92	88	108	103	121	0.457825	0.9900	nonperiodic-templates
109	100	87	99	84	111	99	100	104	107	0.643366	0.9920	nonperiodic-templates
116	99	108	94	96	95	91	85	105	111	0.484646	0.9940	nonperiodic-templates
103	118	108	87	120	91	89	96	103	85	0.116065	0.9920	nonperiodic-templates
98	99	116	103	121	98	93	83	98	91	0.250558	0.9880	overlapping-templates
113	85	100	94	110	95	94	104	104	101	0.715679	0.9920	universal
97	92	93	105	98	86	114	102	108	105	0.703417	0.9910	apen
55	60	66	71	68	65	67	53	57	77	0.504062	0.9922	random-excursions
52	63	66	63	69	68	65	61	68	64	0.937294	0.9875	random-excursions
64	51	67	69	64	65	61	68	63	67	0.916558	0.9844	random-excursions
58	66	59	71	69	66	59	59	69	63	0.935540	0.9953	random-excursions
68	69	56	65	63	63	64	65	68	58	0.975565	0.9859	random-excursions
54	73	37	71	74	70	48	67	78	67	0.003006	0.9969	random-excursions
64	73	66	61	60	69	50	59	73	64	0.630509	0.9906	random-excursions
57	53	74	74	68	68	64	69	60	52	0.397299	0.9875	random-excursions
60	56	65	71	65	81	73	63	55	50	0.186968	0.9953	random-excursions-variant
61	61	61	65	80	73	60	63	62	53	0.516651	0.9969	random-excursions-variant
64	62	68	58	68	64	63	76	55	61	0.834308	0.9953	random-excursions-variant
63	70	57	56	84	53	69	75	64	48	0.046288	0.9922	random-excursions-variant
62	67	72	50	60	63	68	56	68	73	0.581115	0.9969	random-excursions-variant
66	68	57	60	59	71	51	64	69	74	0.604104	0.9969	random-excursions-variant
66	50	65	57	65	78	63	59	76	60	0.336536	0.9937	random-excursions-variant
59	61	66	72	61	60	74	60	64	62	0.908250	0.9906	random-excursions-variant
67	44	65	73	65	50	76	63	76	60	0.069238	0.9937	random-excursions-variant
63	54	49	57	65	73	74	65	58	81	0.117333	0.9922	random-excursions-variant
69	41	64	53	55	63	76	67	79	72	0.023877	0.9875	random-excursions-variant
64	42	64	66	71	67	66	82	58	59	0.091812	0.9859	random-excursions-variant
57	59	60	70	65	63	72	65	55	73	0.763491	0.9828	random-excursions-variant

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	P-VALUE	PROPORTION	STATISTICAL TEST
61	52	64	61	68	58	79	64	69	63	0.584390	0.9828	random-excursions-variant
58	54	63	66	64	68	65	68	68	65	0.957668	0.9875	random-excursions-variant
56	65	64	67	62	63	60	66	74	62	0.948828	0.9875	random-excursions-variant
50	63	73	65	64	55	71	54	78	66	0.250497	0.9890	random-excursions-variant
54	59	73	63	53	62	67	71	68	69	0.640425	0.9890	random-excursions-variant
92	114	102	99	86	94	93	103	102	115	0.550347	0.9860	serial
99	116	106	107	81	83	91	101	110	106	0.207730	0.9870	serial
109	101	101	94	108	108	101	91	91	96	0.893482	0.9850	linear-complexity

B.3 Résultat des test de Diehard

Birthday spacings test

Bits used	mean	χ^2	p-value
1 to 24	15.89	19.7534	0.287035
2 to 25	15.55	15.1196	0.586867
3 to 26	15.69	18.8422	0.337665
4 to 27	15.58	25.7784	0.078594
5 to 28	15.93	21.9248	0.187630
6 to 29	15.89	16.6266	0.479931
7 to 30	16.16	7.1678	0.981270
8 to 31	15.88	18.8492	0.337255
9 to 32	15.70	18.9864	0.329313

degree of freedoms is : 17 p-value for χ^2 test on those 9 p-values : 0.306301

Overlapping 5-permutations test

sample 1 : χ^2=115.669376 with df=99 ; p-value= 0.120845

sample 2 : χ^2=80.435046 with df=99 ; p-value= 0.913721

Bitstream test

Bitstream	No. missing words	z-score	p-value
1	142175	0.62	0.267390
2	142297	0.91	0.182529
3	142319	0.96	0.169240
4	141658	−0.59	0.721472
5	142407	1.16	0.122459
6	141301	−1.42	0.922390
7	141914	0.01	0.495647
8	142258	0.81	0.207636
9	142025	0.27	0.393481
10	141633	−0.65	0.740741
11	142412	1.17	0.120105
12	141662	−0.58	0.718325
13	142194	0.67	0.252988
14	141723	−0.44	0.668346
15	141865	−0.10	0.541247
16	141867	−0.10	0.539392
17	141284	−1.46	0.927999
18	141841	−0.16	0.563421
19	141884	−0.06	0.523596
20	141613	−0.69	0.755644

Overlapping-Pairs-Sparse-Occupancy

Bits used	No. missing words	z-score	p-value
23 to 32	142048	0.4782	0.316264
22 to 31	141874	−0.1218	0.548482
21 to 30	141528	−1.3149	0.905733
20 to 29	142078	0.5816	0.280411
19 to 28	141763	−0.5046	0.693075
18 to 27	141941	0.1092	0.456519
17 to 26	141861	−0.1667	0.566179
16 to 25	141613	−1.0218	0.846569
15 to 24	142038	0.4437	0.328633
14 to 23	141791	−0.4080	0.658376
13 to 22	141764	−0.5011	0.691863
12 to 21	141561	−1.2011	0.885151
11 to 20	141755	−0.5322	0.702697
10 to 19	141578	−1.1425	0.873380

9 to 18	142164	0.8782	0.189925
8 to 17	142067	0.5437	0.293327
7 to 16	141678	−0.7977	0.787475
6 to 15	142074	0.5678	0.285076
5 to 14	141381	−1.8218	0.965759
4 to 13	141534	−1.2942	0.902209
3 to 12	141550	−1.2391	0.892340
2 to 11	142107	0.6816	0.247739
1 to 10	142128	0.7540	0.225414

Overlapping-Quadruples-Sparse-Occupancy

Bits used	No. missing words	z-score	p-value
28 to 32	142046	0.4633	0.321579
27 to 31	141546	−1.2316	0.890956
26 to 30	142183	0.9277	0.176783
25 to 29	141726	−0.6215	0.732851
24 to 28	141782	−0.4316	0.666994
23 to 27	141863	−0.1571	0.562398
22 to 26	141582	−1.1096	0.866413
21 to 25	141689	−0.7469	0.772432
20 to 24	142278	1.2497	0.105699
19 to 23	141578	−1.1232	0.869314
18 to 22	142265	1.2057	0.113974
17 to 21	141799	−0.3740	0.645798
16 to 20	141988	0.2667	0.394859
15 to 19	141921	0.0396	0.484222
14 to 18	141795	−0.3876	0.650829
13 to 17	141749	−0.5435	0.706604
12 to 16	141886	−0.0791	0.531517
11 to 15	141796	−0.3842	0.649574
10 to 14	142140	0.7819	0.217127
9 to 13	141868	−0.1401	0.555710
8 to 12	142325	1.4091	0.079410
7 to 11	142062	0.5175	0.302395
6 to 10	141287	−2.1096	0.982553
5 to 9	141881	−0.0960	0.538253
4 to 8	141927	0.0599	0.476118
3 to 7	142217	1.0429	0.148486
2 to 6	141320	−1.9977	0.977127
1 to 5	141708	−0.6825	0.752530

DNA test

Bits used	No. missing words	z-score	p-value
31 to 32	141864	−0.1337	0.553187
30 to 31	141879	−0.0895	0.535645
29 to 30	141943	0.0993	0.460441
28 to 29	141680	−0.6765	0.750635
27 to 28	141323	−1.7296	0.958148
26 to 27	141626	−0.8358	0.798361
25 to 26	141569	−1.0039	0.842292
24 to 25	141349	−1.6529	0.950823
23 to 24	141578	−0.9774	0.835808
22 to 23	142562	1.9253	0.027097
21 to 22	142292	1.1288	0.129487
20 to 21	141786	−0.3638	0.641998
19 to 20	141816	−0.2753	0.608461
18 to 19	141786	−0.3638	0.641998
17 to 18	141934	0.0728	0.470993
16 to 17	141891	−0.0541	0.521561
15 to 16	142437	1.5565	0.059789
14 to 15	142337	1.2616	0.103553
13 to 14	142200	0.8574	0.195603
12 to 13	142086	0.5212	0.301131
11 to 12	142075	0.4887	0.312526
10 to 11	142540	1.8604	0.031416
9 to 10	141838	−0.2104	0.583327
8 to 9	142063	0.4533	0.325165
7 to 8	142169	0.7660	0.221842
6 to 7	141944	0.1023	0.459271
5 to 6	141970	0.1790	0.428982
4 to 5	141847	−0.1839	0.572940
3 to 4	142349	1.2970	0.097322
2 to 3	142117	0.6126	0.270072
1 to 2	141413	−1.4641	0.928417

parking lot test

No. succeses	z-score	p-value
3516	−0.3196	0.625377
3505	−0.8219	0.794438
3501	−1.0046	0.842447
3526	0.1370	0.445521
3530	0.3196	0.374623

3541	0.8219	0.205562
3489	−1.5525	0.939730
3497	−1.1872	0.882429
3535	0.5479	0.291865
3518	−0.2283	0.590298

Square side=100, avg. no. parked=3515.80 sample std.=16.50 p-value of the χ^2 for those 10 p-values : 0.006801

minimum distance test

Sample no.	d^2	mean	equiv uni
5	0.1449	1.4640	0.135508
10	0.5668	1.1518	0.434278
15	1.8481	1.2832	0.843920
20	0.2329	1.1166	0.208731
25	0.7239	1.1295	0.516901
30	0.3452	1.0497	0.293139
35	0.1664	1.0067	0.154020
40	0.1616	1.0154	0.149908
45	0.0530	0.9615	0.051879
50	0.9304	0.9712	0.607440
55	1.2380	1.0427	0.711829
60	2.3920	1.0227	0.909645
65	0.0052	1.0157	0.005218
70	0.2688	1.0002	0.236735
75	2.0424	1.0063	0.871610
80	0.6418	0.9861	0.475350
85	1.2085	1.0334	0.703155
90	3.0255	1.0438	0.952201
95	0.0088	1.0665	0.008770
100	0.1655	1.0742	0.153256

Result of χ^2 test on 100 transformed $mindist^2$'s : p-value=0.702525

3D spheres test

sample no	r^3	equiv. uni.
1	17.284	0.437936
2	25.956	0.579028
3	43.154	0.762706
4	181.971	0.997679
5	30.527	0.638528
6	15.886	0.411119
7	5.939	0.179608
8	11.970	0.329014

9	7.907	0.231699
10	48.070	0.798573
11	15.758	0.408596
12	83.272	0.937697
13	80.627	0.931953
14	13.707	0.366754
15	95.404	0.958419
16	11.158	0.310600
17	34.913	0.687693
18	7.943	0.232620
19	12.169	0.333440
20	38.659	0.724351

p-value for χ^2 test on those 20 p-values : 0.326291

Squeeze test

Table of standardized frequency counts
$(obs - exp)^2/exp$ for j=(1,..,6), 7,...,47,(48,...)

−1.5	−1.2	−2.5	0.3	−1.0	1.4
−2.0	0.8	−1.4	−1.4	−0.5	−0.1
0.4	1.1	1.1	−0.3	1.6	−1.3
0.4	−0.5	−0.1	−0.3	−1.4	0.2
1.4	1.4	−2.0	0.1	2.3	−0.8
−1.3	−0.1	−0.3	1.8	−1.2	−0.1
−0.2	−1.3	−0.8	2.1	−0.6	−1.0
−0.1					

χ^2 with 42 degrees of freedom :59853498 z-score=1.947976, p-value=0.036279

Overlapping sums test

Test no	p-value
1	0.249176
2	0.597270
3	0.529569
4	0.001002
5	0.663584
6	0.527878
7	0.552203
8	0.773449
9	0.864953
10	0.003806

p-value for 10 χ^2 tests on 100 χ^2 tests :0.171575

Craps test

Observed	Expected
98223	98585.858586

z-score = -1.623, p-value = 0.94770
Analysis of Throws-per-Game :

Throws	Observed	Expected	χ^2	$\sum(O-E)^2/E$
1	66712	66666.7	0.031	0.031
2	37685	37654.3	0.025	0.056
3	27046	26954.7	0.309	0.365
4	19444	19313.5	0.882	1.247
5	13522	13851.4	7.834	9.082
6	9959	9943.5	0.024	9.106
7	7188	7145.0	0.258	9.364
8	5173	5139.1	0.224	9.588
9	3693	3699.9	0.013	9.601
10	2621	2666.3	0.770	10.370
11	1870	1923.3	1.479	11.849
12	1366	1388.7	0.372	12.221
13	1005	1003.7	0.002	12.223
14	725	726.1	0.002	12.225
15	542	525.8	0.497	12.722
16	394	381.2	0.433	13.155
17	296	276.5	1.369	14.524
18	195	200.8	0.169	14.694
19	145	146.0	0.007	14.700
20	131	106.2	5.783	20.484
21	288	287.1	0.003	20.486

χ^2 = 20.49 for 20 degrees of freedom, p = 0.42790
p-value for no. of wins : 0.947696
p-value for throws/game : 0.427900

Annexe C

Code source du système de chiffrement

Ce code source est une implémentation du système de chiffrement présenté au chapitre 5. Le code est en C de façon à être très facilement portable. Il a été pensé pour une architecture 32 bits.

Son fonctionnement est relativement simple : le programme compilé attend quatre paramètres :

- le vecteur d'initialisation, qui est une chaîne héxadécimale de 10 caractères,
- la clé de chiffrement/déchiffrement, qui est une chaîne héxadécimale de 25 caractères
- le fichier source
- le fichier destination.

C.1 Fichier Makefile

```
1  CC=gcc
2  CFLAGS=-O3 -g0 -ansi -Wall
3
4  bc2: main.o register.o
5          $(CC) main.o register.o -o bc2
6
7  main.o: main.c register.h config.h
8          $(CC) $(CFLAGS) -c main.c
9
10 register.o: register.c register.h config.h
11         $(CC) $(CFLAGS) -c register.c
12
13 clean:
14         rm -f *.o *~ expand.h bc2
```

C.2 Fichier config.h

```
1  #ifndef CONFIG_H_
2  #define CONFIG_H_
```

```
3
4   typedef unsigned int            PRIMITIVE_TYPE;
5
6   #define REGISTER_SIZE_BYTES     (REGISTER_SIZE*sizeof(PRIMITIVE_TYPE))
7   #define REGISTER_SIZE_BITS      (REGISTER_SIZE_BYTES<<3)
8
9   #define BIG_JUMP_SIZE           32
10
11  #define CIPHER_FILE_BLOCK       65536
12
13  #endif /*CONFIG_H_*/
```

C.3 Fichier main.c

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "register.h"
5
6   #define REGISTERS       5
7   #define COMBINAISON     0x64E98F31
8
9   const unsigned int      reg_size[] = {45, 46, 47, 48, 49};
10  unsigned int            reg_size_mbj[REGISTERS];
11  unsigned int            X[REGISTERS][2];
12  unsigned int            KL[6];
13  unsigned char           IV[8], K[16];
14  TRUTHTABLE              *n;
15  TRUTHTABLE              *tables[REGISTERS];
16  unsigned int            S[2], D[2], DS[2];
17  unsigned char           bit;
18
19  int getBitsFromKey(int n)
20  {
21          unsigned int    r = 0;
22          while(n--)
23          {
24                  bit  =  KL[0]     &1;
25                  bit ^= (KL[1]>>25)&1;
26                  bit ^= (KL[3]>>30)&1;
27                  bit ^= (KL[5]>>30)&1;
28                  KL[0] = (KL[0] >> 1) | ((KL[1]&1)<<31);
29                  KL[1] = (KL[1] >> 1) | ((KL[2]&1)<<31);
30                  KL[2] = (KL[2] >> 1) | ((KL[3]&1)<<31);
31                  KL[3] = (KL[3] >> 1) | ((KL[4]&1)<<31);
32                  KL[4] = (KL[4] >> 1) | ((KL[5]&1)<<31);
```

```
33                  KL[5] = (KL[5] >> 1) | (bit<<31);
34                  r = bit | (r << 1);
35          }
36          return r;
37  }
38
39  unsigned char bigRegGetBit(
40                  int reg,
41                  unsigned int order)
42  {
43          if (X[reg][0] == 1 && X[reg][1] == 0) return 1;
44
45          S[0]  = (X[reg][0] >> 1) | (X[reg][1] << 31);
46          S[1]  = X[reg][1] >> 1;
47          D[0]  = X[reg][0] ^ X[reg][1];
48          DS[0] = S[0] ^ S[1];
49
50          bit  = (DS[0] >> (order-1)) & 1;
51          bit ^= GET_TRUTHTABLE(tables[reg], D[0] & masks[order]);
52          bit ^= generalDelta(S, order+BIG_JUMP_SIZE-1, order-1);
53
54          return bit;
55  }
56
57  void cipherInit()
58  {
59          unsigned int            i, j, b, r, k;
60
61          /* Précalcul des tables de vérité */
62          for (i=0; i<REGISTERS; i++)
63          {
64                  reg_size_mbj[i] = reg_size[i] - BIG_JUMP_SIZE;
65                  r               = getBitsFromKey(4);
66                  k               = getBitsFromKey(3);
67                  n               = computeTruthtable(3, r, k);
68                  tables[i]       = jump(n, 10, reg_size_mbj[i]-10,
69                                         getBitsFromKey(1));
70                  X[i][0]         = getBitsFromKey(32);
71                  X[i][1]         = getBitsFromKey(reg_size[i]-32);
72                  FREE_TRUTHTABLE(n);
73          }
74
75          /* Avance pour diffusion de l'état initial */
76          for (j=0; j<256; j++)
77          {
78                  for (i=0; i<REGISTERS; i++)
79                  {
```

```
80                          b = bigRegGetBit(i, reg_size_mbj[i]);
81                          X[i][0]   = (X[i][0] >> 1) | (X[i][1] << 31);
82                          X[i][1] >>= 1;
83                          X[i][1]  |= b << ((reg_size_mbj[i]) & 0x1F);
84                  }
85          }
86  }
87
88  void cipherDo(unsigned char *buffer, unsigned int size)
89  {
90          unsigned int            i, l;
91          unsigned char           r, b, c;
92
93          c       = 0;
94          l       = 0;
95          size <<= 3;
96          while (l++ < size)
97          {
98                  r = 0;
99                  for (i=0; i<REGISTERS; i++)
100                 {
101                         b = bigRegGetBit(i, reg_size_mbj[i]);
102                         X[i][0]   = (X[i][0] >> 1) | (X[i][1] << 31);
103                         X[i][1] >>= 1;
104                         X[i][1]  |= b << ((reg_size_mbj[i]) & 0x1F);
105                         r |= b << i;
106                 }
107                 c <<= 1; c |= (COMBINAISON>>r)&1;
108                 if ((l&7)==0) *buffer++ ^= c;
109         }
110
111 }
112
113 void cipherEnd()
114 {
115         unsigned int            i;
116
117         for (i=0; i<REGISTERS; i++)
118                 FREE_TRUTHTABLE(tables[i]);
119 }
120
121 int main(int argc, char **argv)
122 {
123         int             i, blocs, rem;
124         unsigned int    size;
125         unsigned char   *buffer, c, t;
126         FILE            *s, *d;
```

```
127
128         if (argc != 5)
129         {
130                 fprintf(stderr, "Usage : %s IV key source destination\n",
131                         argv[0]);
132                 return EXIT_FAILURE;
133         }
134
135         /* Lecture de la clé */
136         if (strlen(argv[1]) != 16)
137         {
138                 fprintf(stderr,
139                         "IV argument should be a string" \
140                         " of 16 hexadecimal characters.\n");
141                 return EXIT_FAILURE;
142         }
143         if (strlen(argv[2]) != 32)
144         {
145                 fprintf(stderr,
146                         "Key argument should be a string" \
147                         " of 32 hexadecimal characters.\n");
148                 return EXIT_FAILURE;
149         }
150
151         for (i=0; i<16; i++)
152         {
153                 if (argv[1][i] >= '0' && argv[1][i] <= '9')
154                         t = argv[1][i] - '0';
155                 else if (argv[1][i] >= 'a' && argv[1][i] <= 'f')
156                         t = argv[1][i] - 'a' + 10;
157                 else if (argv[1][i] >= 'A' && argv[1][i] <= 'F')
158                         t = argv[1][i] - 'A' + 10;
159                 else
160                 {
161                         fprintf(stderr,
162                                 "IV is not an hexadecimal string.\n");
163                         return EXIT_FAILURE;
164                 }
165                 if (!(i & 1)) c = t << 4; else IV[i>>1] = c | t;
166         }
167
168         for (i=0; i<32; i++)
169         {
170                 if (argv[2][i] >= '0' && argv[2][i] <= '9')
171                         t = argv[2][i] - '0';
172                 else if (argv[2][i] >= 'a' && argv[2][i] <= 'f')
173                         t = argv[2][i] - 'a' + 10;
```

```
174                 else if (argv[2][i] >= 'A' && argv[2][i] <= 'F')
175                         t = argv[2][i] - 'A' + 10;
176                 else
177                 {
178                         fprintf(stderr,
179                                 "Key is not an hexadecimal string.\n");
180                         return EXIT_FAILURE;
181                 }
182
183                 if (!(i & 1)) c = t << 4; else K[i>>1] = c | t;
184         }
185
186         KL[0] = IV[0] | (IV[1]<<8) | (IV[2]<<16) | (IV[3]<<24);
187         KL[1] = IV[4] | (IV[5]<<8) | (IV[6]<<16) | (IV[7]<<24);
188
189         KL[2] = K[ 0] | (K[ 1]<<8) | (K[ 2]<<16) | (K[ 3]<<24);
190         KL[3] = K[ 4] | (K[ 5]<<8) | (K[ 6]<<16) | (K[ 7]<<24);
191         KL[4] = K[ 8] | (K[ 9]<<8) | (K[10]<<16) | (K[11]<<24);
192         KL[5] = K[12] | (K[13]<<8) | (K[14]<<16) | (K[15]<<24);
193
194         for (i=0; i<256; i++) getBitsFromKey(32);
195         cipherInit();
196
197         /* Ouverture des fichiers */
198         s = fopen(argv[3], "rb");
199         if (!s)
200         {
201                 fprintf(stderr, "Unable to open source file.\n");
202                 return EXIT_FAILURE;
203         }
204
205         d = fopen(argv[4], "wb");
206         if (!d)
207         {
208                 fprintf(stderr, "Unable to create destination file.\n");
209                 fclose(s);
210                 return EXIT_FAILURE;
211         }
212
213         buffer = (unsigned char *)malloc(CIPHER_FILE_BLOCK);
214
215         fseek(s, 0, SEEK_END);
216         size = ftell(s);
217         fseek(s, 0, SEEK_SET);
218
219         /* Chiffrement ou déchiffrement */
220         blocs = size / CIPHER_FILE_BLOCK;
```

```
221             rem   = size % CIPHER_FILE_BLOCK;
222
223             while(blocs--)
224             {
225                     printf("."); fflush(stdout);
226                     fread(buffer, CIPHER_FILE_BLOCK, 1, s);
227                     cipherDo(buffer, CIPHER_FILE_BLOCK);
228                     fwrite(buffer, CIPHER_FILE_BLOCK, 1, d);
229             }
230             if (rem)
231             {
232                     printf("."); fflush(stdout);
233                     fread(buffer, rem, 1, s);
234                     cipherDo(buffer, rem);
235                     fwrite(buffer, rem, 1, d);
236             }
237             printf("\n");
238             cipherEnd();
239
240             free(buffer);
241             fclose(s);
242             fclose(d);
243
244             return EXIT_SUCCESS;
245     }
```

C.4 Fichier register.c

```
1   #include <stdlib.h>
2   #include <string.h>
3   #include "register.h"
4
5   /* Tables de vérité des 16 fonctions de de Bruijn d'ordre 4 */
6   const unsigned short DB4[16] = {
7           26213, 22181, 27221, 23189, 22169, 22101, 26201, 26261,
8           26009, 25941, 25961, 26969, 21849, 21909, 21929, 22937
9   };
10
11  const unsigned int masks[33] = {
12          0x00000000, 0x00000001, 0x00000003, 0x00000007,
13          0x0000000F, 0x0000001F, 0x0000003F, 0x0000007F,
14          0x000000FF, 0x000001FF, 0x000003FF, 0x000007FF,
15          0x00000FFF, 0x00001FFF, 0x00003FFF, 0x00007FFF,
16          0x0000FFFF, 0x0001FFFF, 0x0003FFFF, 0x0007FFFF,
17          0x000FFFFF, 0x001FFFFF, 0x003FFFFF, 0x007FFFFF,
18          0x00FFFFFF, 0x01FFFFFF, 0x03FFFFFF, 0x07FFFFFF,
```

```
19          0x0FFFFFFF, 0x1FFFFFFF, 0x3FFFFFFF, 0x7FFFFFFF,
20          0xFFFFFFFF
21  };
22
23  const unsigned int alpha[33] = {
24          0x00000000, 0x00000000, 0x00000002, 0x00000002,
25          0x0000000A, 0x0000000A, 0x0000002A, 0x0000002A,
26          0x000000AA, 0x000000AA, 0x000002AA, 0x000002AA,
27          0x00000AAA, 0x00000AAA, 0x00002AAA, 0x00002AAA,
28          0x0000AAAA, 0x0000AAAA, 0x0002AAAA, 0x0002AAAA,
29          0x000AAAAA, 0x000AAAAA, 0x002AAAAA, 0x002AAAAA,
30          0x00AAAAAA, 0x00AAAAAA, 0x02AAAAAA, 0x02AAAAAA,
31          0x0AAAAAAA, 0x0AAAAAAA, 0x2AAAAAAA, 0x2AAAAAAA,
32          0xAAAAAAAA
33  };
34
35  const unsigned int delta[33] = {
36          0x00000000, 0x00000001, 0x00000001, 0x00000005,
37          0x00000005, 0x00000015, 0x00000015, 0x00000055,
38          0x00000055, 0x00000155, 0x00000155, 0x00000555,
39          0x00000555, 0x00001555, 0x00001555, 0x00005555,
40          0x00005555, 0x00015555, 0x00015555, 0x00055555,
41          0x00055555, 0x00155555, 0x00155555, 0x00555555,
42          0x00555555, 0x01555555, 0x01555555, 0x05555555,
43          0x05555555, 0x15555555, 0x15555555, 0x55555555,
44          0x55555555
45  };
46
47  /* Calcul de la table de vérité d'une
48     fonction de de Bruijn d'ordre 4 + 2*jumps */
49  TRUTHTABLE * computeTruthtable(
50                  unsigned int jumps,
51                  unsigned int root,
52                  unsigned int key)
53  {
54          int                     j, from_order, to_order;
55          TRUTHTABLE              *from, *to;
56          unsigned int            x;
57          unsigned int            S, RS, DRS, D, DS, DDS, PDDS, DD;
58          unsigned char           URS, UDRS, hk, hn;
59
60          from_order = 4;
61          from       = ALLOCATE_TRUTHTABLE(from_order);
62          for (j=0; j<1<<from_order; j++)
63                  SET_TRUTHTABLE(from, j, (DB4[root]>>j)&1);
64
65          to_order = 6;
```

```
66          to       = ALLOCATE_TRUTHTABLE(to_order);
67          ZERO_TRUTHTABLE(to, to_order);
68
69
70          for (j=0; j<jumps; j++)
71          {
72                  unsigned int comp = (key >> j) & 1;
73                  const unsigned int *ptr = comp ? alpha : delta;
74
75                  for (x=0; x<1<<to_order; x++)
76                  {
77                          if (comp) x ^= masks[to_order];
78
79                          S    = x >> 1;
80                          RS   = S & masks[to_order-1];
81                          DRS  = (RS ^ (RS >> 1)) & masks[to_order-2];
82                          D    = (x ^ (x >> 1)) & masks[to_order-1];
83                          DS   = (S ^ (S >> 1)) & masks[to_order-1];
84                          DDS  = (DS ^ (DS >> 1)) & masks[to_order-2];
85                          PDDS = DDS >> (from_order-1);
86                          DD   = (D ^ (D >> 1)) & masks[to_order-2];
87
88                          URS  = RS  == ptr[to_order-1] ? 1 : 0;
89                          UDRS = DRS == ptr[to_order-2] ? 1 : 0;
90
91                          hk = GET_TRUTHTABLE(from, DD);
92
93                          hn = PDDS ^ hk ^ URS ^ UDRS;
94
95                          if (comp) x ^= masks[to_order];
96
97                          SET_TRUTHTABLE(to, x, hn ^ comp);
98                  }
99
100                 free(from);
101
102                 if (j != jumps - 1)
103                 {
104                         from       = to;
105                         from_order = to_order;
106                         to_order  += 2;
107                         to = (TRUTHTABLE *)malloc(1<<(to_order-3));
108                         memset(to, 0, 1<<(to_order-3));
109                 }
110         }
111
112         return to;
```

```
113  }
114
115  unsigned int Dp(unsigned int x, unsigned int p)
116  {
117          int                             i;
118          unsigned        int             r = x;
119
120          for (i=0; i<32; i++)
121          {
122                  if (p&1) r ^= (r >> (1<<i));
123                  p >>= 1;
124                  if (!p) break;
125          }
126
127          return r;
128  }
129
130  void generalDp(
131                  unsigned int x[2],
132                  unsigned int r[2],
133                  unsigned int p)
134  {
135          int             i, q;
136
137          /* Cas particulier pour p = 32 ou p = 33
138             sachant que 99,1% des cas seront ceux-ci */
139          switch(p)
140          {
141                  case 32: r[0] = x[0] ^ x[1];
142                           return;
143                  case 33: r[0]  = x[0] ^ x[1];
144                           r[0] ^= r[0] >> 1;
145                           return;
146                  default:
147                           break;
148          }
149
150          r[0] = x[0]; r[1] = x[1];
151
152          for (i=0; i<5; i++)
153          {
154                  if (p&1)
155                  {
156                           q     = 1<<i;
157                           r[0] ^= (r[0] >> q) | (r[1] << (32 - q));
158                           r[1] ^= (r[1] >> q);
159                  }
```

```
160                 p >>= 1;
161                 if (!p) return;
162         }
163 }
164
165 unsigned char Delta(
166                 unsigned int x,
167                 unsigned int n,
168                 unsigned int i,
169                 int type)
170 {
171         const unsigned int      *p_alpha, *p_delta;
172         unsigned int            d, k, m, t, D;
173
174         if (!type)
175         {
176                 p_alpha = alpha;
177                 p_delta = delta;
178         }
179         else
180         {
181                 p_alpha = delta;
182                 p_delta = alpha;
183         }
184
185         if (x == p_delta[n]) return 1;
186         if (x == 0)          return 0;
187         if (x == masks[n])   return 0;
188         if (x == p_alpha[n]) return 0;
189
190         d = n - i + 1;
191         if ((Dp(x, d) & masks[n-d]) != 0) return 0;
192
193         k = 1; m = d-1;
194         while (k <= m)
195         {
196                 t = (k+m)>>1;
197                 D = Dp(x, t) & masks[n-t];
198                 if (D == p_delta[n-t]) return 1;
199                 if (D == p_alpha[n-t]) return 0;
200                 while (D == masks[n-t])
201                 {
202                         t--;
203                         D = Dp(x, t) & masks[n-t];
204                         if (D == p_delta[n-t]) return 1;
205                         if (D == p_alpha[n-t]) return 0;
206                 }
```

```
207
208                 if (D == 0) m = t-1; else k = t;
209         }
210
211         return 0;
212 }
213
214 int isSomething(
215                 unsigned int x[2],
216                 unsigned int n,
217                 const unsigned int *ptr)
218 {
219         if (n <= 32)
220                 return (x[0] == ptr[n]) && (x[1] == 0);
221         else
222         {
223                 return (x[0] == ptr[32]) && (x[1] == ptr[n-32]);
224         }
225 }
226
227 int isZero(unsigned int x[2])
228 {
229         return (x[0] == 0) && (x[1] == 0);
230 }
231
232 void maskIt(unsigned int x[2], unsigned int n)
233 {
234         if (n <= 32)
235         {
236                 x[0] &= masks[n]; x[1]  = 0;
237         }
238         else
239                 x[1] &= masks[n-32];
240 }
241
242 unsigned char generalDelta(
243                 unsigned int x[2],
244                 unsigned int n,
245                 unsigned int i)
246 {
247         unsigned int            d;
248         unsigned int            k, m, t, D[2], r[2];
249
250         if (isSomething(x, n, delta)) return 1;
251         if (isZero(x))                return 0;
252         if (isSomething(x, n, masks)) return 0;
253         if (isSomething(x, n, alpha)) return 0;
```

```
254
255         d = n - i + 1;
256
257         r[0]  = x[0] ^ x[1];
258         r[0] ^= r[0] >> 1;
259         if (r[0] & masks[n-d]) return 0;
260
261         k = 1; m = d-1;
262         while (k <= m)
263         {
264                 t = (k+m)>>1;
265                 generalDp(x, D, t); maskIt(D, n-t);
266                 if (isSomething(D, n-t, delta)) return 1;
267                 if (isSomething(D, n-t, alpha)) return 0;
268                 while (isSomething(D, n-t, masks))
269                 {
270                         t--;
271                         generalDp(x, D, t); maskIt(D, n-t);
272                         if (isSomething(D, n-t, delta)) return 1;
273                         if (isSomething(D, n-t, alpha)) return 0;
274                 }
275
276                 if (isZero(D)) m = t-1; else k = t;
277         }
278
279         return 0;
280 }
281
282 TRUTHTABLE * jump(
283                 TRUTHTABLE *t,
284                 unsigned int order,
285                 unsigned int jump_len,
286                 int type)
287 {
288         unsigned int    x;
289         int             to_order = order + jump_len;
290         unsigned char   PDS, hk, D;
291
292         TRUTHTABLE      *r = ALLOCATE_TRUTHTABLE(to_order);
293         ZERO_TRUTHTABLE(r, to_order);
294
295         for (x=0; x<1<<to_order; x++)
296         {
297                 if (type) x ^= masks[to_order];
298                 PDS = (Dp(x>>1, jump_len) >> (order-1)) & 1;
299                 hk  = GET_TRUTHTABLE(t, Dp(x, jump_len) & masks[order]);
300                 D   = Delta(x>>1, to_order-1, order-1, type);
```

```
301                     if (type) x ^= masks[to_order];
302                     SET_TRUTHTABLE(r, x, PDS ^ hk ^ D ^ type);
303             }
304
305             return r;
306     }
```

C.5 Fichier register.h

```
 1   #ifndef REGISTER_H_
 2   #define REGISTER_H_
 3
 4   #include "config.h"
 5
 6   typedef unsigned char TRUTHTABLE;
 7
 8   #define GET_TRUTHTABLE(TT,X) \
 9           ((TT[(X)>>3]>>((X)&7))&1)
10   #define SET_TRUTHTABLE(TT,X,V) \
11           (TT[(X)>>3] = (TT[(X)>>3]&(~(1<<((X)&7))))|((V)<<((X)&7)))
12   #define TRUTHTABLE_LEN(O) \
13           (((1<<(O))+7)>>3)
14   #define ALLOCATE_TRUTHTABLE(O) \
15           (TRUTHTABLE*)malloc(TRUTHTABLE_LEN(O));
16   #define ZERO_TRUTHTABLE(TT,O) \
17           memset(TT, 0, TRUTHTABLE_LEN(O))
18   #define FREE_TRUTHTABLE(TT) \
19           free(TT)
20
21   extern const unsigned int       masks[33];
22
23   TRUTHTABLE *    computeTruthtable(
24                           unsigned int jumps,
25                           unsigned int root,
26                           unsigned int key);
27   TRUTHTABLE *    jump(
28                           TRUTHTABLE *t,
29                           unsigned int order,
30                           unsigned int jump_len,
31                           int type);
32   unsigned char   generalDelta(
33                           unsigned int x[2],
34                           unsigned int n,
35                           unsigned int i);
36
37   #endif /*REGISTER_H_*/
```

Table des matières

1 Systèmes de chiffrement à flot **7**

1.1 Introduction 7
1.2 Présentation générale du chiffrement à flot 7
1.3 Primitives 8
1.3.1 Générateurs de pseudo-aléa 8
1.3.2 Les registres à décalage 9
1.3.3 Registre non singulier 10
1.3.4 Modification de la table de vérité 11
1.3.5 Registres à décalage à rétroaction linéaire 12
1.3.6 FCSR 15
1.4 Les T-fonctions 17
1.5 BBS 17
1.6 Exemples de système de chiffrement 18
1.6.1 Shrinking Generator 18
1.6.2 Registre filtré 18
1.6.3 Registres combinés 19
1.6.4 A5/1 19
1.6.5 F-FCSR 20
1.6.6 RC4 20
1.6.7 E0 21
1.6.8 Achterbahn 22

2 Attaques des chiffrement à flot **25**

2.1 Introduction 25
2.2 Attaque de Berlekamp-Massey 25
2.3 Attaque par corrélation 26
2.3.1 Attaque par corrélation rapide 27
2.4 Attaques algébriques 29
2.4.1 Attaques algébriques par approximation 29
2.4.2 Attaques algébriques et multiples de petit degré 29
2.4.3 Attaques algébriques augmentées 30
2.5 Attaque par compromis temps-mémoire 30
2.6 Synthèse des attaques 32

3 Attaque sur les T-Fonctions **35**
3.1 Introduction . . . 35
3.2 Les T-Fonctions . . . 35
3.2.1 Définitions et notations . . . 35
3.2.2 Inversibilité d'une fonction paramétrique . . . 38
3.2.3 Construction de suites pseudo-aléatoires . . . 41
3.3 Contexte de l'attaque, définitions et faits . . . 46
3.4 L'attaque . . . 48
3.5 Un exemple d'attaque . . . 49
3.6 L'algorithme . . . 52
3.7 Améliorations possibles . . . 53
3.8 Conclusion . . . 54

4 Les suites de De Bruijn **55**
4.1 Introduction . . . 55
4.2 Définitions . . . 55
4.3 Prefer one . . . 57
4.4 Construction à partir d'un LFSR . . . 58
4.5 Construction à partir de PCR . . . 59
4.5.1 Définition d'un PCR . . . 59
4.5.2 Construction de suites de De Bruijn . . . 60
4.6 Construction à partir de CCR . . . 61
4.7 Le D-homomorphisme . . . 62
4.7.1 Définitions . . . 62
4.7.2 Propriétés de D . . . 64
4.7.3 Application : construction de suites de De Bruijn . . . 66

5 Élaboration d'un système de chiffrement **71**
5.1 Introduction . . . 71
5.2 Générateur à NLFSRs dynamiques . . . 71
5.2.1 Vue générale . . . 71
5.2.2 Fonctionnement . . . 72
5.3 Étude statistique . . . 78
5.4 Comportement face aux attaques connues . . . 79
5.4.1 Attaque de Berlekamp-Massey . . . 79
5.4.2 Attaque par corrélation . . . 80
5.4.3 Attaques algébriques . . . 80
5.4.4 Attaques par compromis temps/mémoire . . . 80

6 Sécurité et bibliothèques logicielles **81**
6.1 Introduction . . . 81
6.2 Services et bibliothèques de fonctions . . . 82
6.3 Problèmes relatifs à l'organisation centralisée . . . 82
6.4 Application pratique . . . 83

6.4.1 Aperçu de la CryptoAPI 83
6.4.2 Organisation de la CryptoAPI 83
6.4.3 Analyse des DLLs 84
6.4.4 Création d'un patch 84
6.4.5 Présentation de la faille 84
6.4.6 Transformation d'une adresse virtuelle en adresse physique 85
6.4.7 Organisation des droits grâce aux RING 85
6.4.8 Analyse de *MmGetPhysicalAddress* 86
6.4.9 Acquisition du RING0 86
6.4.10 Modification de la CryptoAPI 86
6.4.11 Application pratique du patch 86
6.5 Conclusion 87

A Définitions et notations **91**
A.1 Le corps $\mathbb{F}_2$ 91
A.2 Les fonctions booléennes 92
A.2.1 Table de vérité 92
A.2.2 Forme normale disjonctive 92
A.2.3 Forme normale algébrique 93
A.3 Transformation de Walsh-Hadamard 93

B Résultats des tests statistiques **95**
B.1 Présentation des tests statistiques 95
B.2 Résultats de la série de tests de Sts 98
B.3 Résultat des test de Diehard 101

C Code source du système de chiffrement **109**
C.1 Fichier Makefile 109
C.2 Fichier config.h 109
C.3 Fichier main.c 110
C.4 Fichier register.c 115
C.5 Fichier register.h 122

Index

2-adique, 15
état
    conjugué, 62
    dual, 62
    interne, 8

A5
    A5/1, 19
    A5/2, 19
Achterbahn, 71
algorithme
    Δ, 74
    Δ modifié, 75
    XL, 29
attaque
    algébrique, 29
    de Berlekamp-Massey, 25, 79
    par algébrique, 80
    par corrélation, 26, 80
    par corrélation rapide, 27
automate à états finis, 8

bases de Gröbner, 29
BBS, 17
bibliothèque logicielle, 81

CCR, 61
cheval de Troie, 88
chiffrement
    à bloc, 6
    à flot, 6, 7
clé
    de chiffrement, 8
codes correcteurs, 28
collisions, 21
compléxité linéaire, 14
CryptoAPI, 82
cryptogramme, 8

CSP, 83
cycle
    auto-dual, 63
    duaux, 63
    poids, 63
    primitif, 63

D-Homomorphisme, 62, 72
De Bruijn
    graphe, 55
    suite, 55
Diehard, 79
DLL, 82
DVD, 81

E0, 21
engendrée, 9
eSTREAM, 20

$\mathbb{F}_2$, 91
F-FCSR, 20
FCSR, 15
fonction
    équilibrée, 94
    à sens unique, 30
    à un cycle, 42
    booléenne, 92
    creuse, 93
    d'Euler, 60
    d'initialisation, 8
    de combinaison, 19
    de combinaison à mémoire, 21
    de rétroaction, 9
    de rétroaction dynamique, 71
    de sortie, 8
    de transition, 8
    inversible, 17, 37, 38
    paramétrique, 38

primitive, 17, 35
signe, 93
t-résiliente, 94
triangulaire, 17, 35, 36
forme normale
algébrique, 93
disjonctive, 92
FSR, 9

générée, 9
GSM, 19

horloge, 8

immunité
algébrique, 30
aux corrélations, 32, 93

k-cycle, 62

Lempel, 62
LFSR, 12, 58
Linux, 82

masque jetable, 8
Microsoft, 82

NIST, 79
NLFSR, 71
non linéarité, 93

One Time Pad, 8
OpenSSL, 82

paradoxe des anniversaires, 31
paramètre, 42
patch, 84
PCR, 59
poids
d'un cycle, 63
de Hamming, 63
polynôme
annulateur, 29
d'initialisation, 13
de rétroaction, 13
irréductible, 14
primitif, 14
prefer one, 57
pseudo-aléa, 8

quantité de ressemblance, 27

récurrence linéaire, 12
réseau
Bluetooth, 21
WiFi, 20
réseau de Feistel triangulaire, 37
résilience, 94
rétroaction dynamique, 71
RC4, 20
registre
à décalage, 9
à rétroaction linéaire, 12
a décalage avec retenue, 15
combinés, 19
singulier, 10
reverse engineering, 84

série génératrice, 12
Shrinking Generator, 18
singulier, 10
so, 82
stream cipher, 7
Sts, 79
système d'équations surdéterminé, 29

T-fonction, 17, 35
test statistique
(Non) Overlapping Templates, 96
Approximate entropy, 97
Block Frequency, 95
Cumulative sums, 97
Frequency, 95
Linear complexity, 96
Longuest Runs of Ones, 95
Random excursion, 97
Rank, 96
Runs, 95
Serial, 97
Spectral DFT, 96
Universal, 96
transformée de Walsh-Hadamard, 93

vecteur d'initialisation, 8
vecteur initial, 77
virus, 88

WEP, 20
Windows, 82

Bibliographie

[1] G. Ars and JC. Faugère. Gröbner Attack of nonlinear generators with a small number of key bits. *Preprint*, 2005.

[2] S. Babbage. A space/time trade-of in exhaustive search attacks on stream ciphers. *European Convention on Security and Detection*, vol.408, 1995.

[3] Thierry Berger, François Arnault, and Cédric Lauradoux. Design and properties of a new pseudorandom generator based on a filtered FCSR automaton. `http://www.ecrypt.eu.org/stream/ffcsr.html`.

[4] E.R Berlekamp. Algebraic Coding Theory. *McGraw-Hill, New York*, 1967.

[5] Eli Biham and Orr Dunkelman. Cryptanalysis of the A5/1 GSM Stream Cipher. In *INDOCRYPT*, pages pp.43–51, 2000.

[6] Alex Biryukov, Adi Shamir, and David Wagner. Real Time Cryptanalysis of A5/1 on a PC. *Lecture Notes in Computer Science*, vol.1978, 2001.

[7] Lenore Blum, Manuel Blum, and Michael Shub. A simple unpredictable pseudo-random number generator. *SIAM Journal on Computing*, vol.15 :pp.364–383, may 1986.

[8] V. Bénony, F. Recher, É. Wegrzynoski, and C. Fontaine. Cryptanalysis of a Particular Case of Klimov-Shamir Pseudo-Random Generator. *Lecture Notes in Computer Science*, vol.3486 :pp.313–322, 2005.

[9] A. Canteaut and E. Filiol. Ciphertext only reconstruction of stream ciphers based on combination generators. *Fast Software Encryption - FSE 2000*, vol.1978, 2000.

[10] A. Canteaut and M. Trabbia. Improved fast correlation attacks using parity-check equations of weight 4 and 5. In *Advances in Cryptology - EUROCRYPT'2000*, number 1807 in Lecture Notes in Computer Science, pages pp.573–588. Springer Verlag, 2000.

[11] Microsoft Security Developer Center. `http://msdn.microsoft.com/security`.

[12] ACI CESAM. Courbes Elliptiques pour la Sécurité des Appareils Mobiles. `http://www.di.ens.fr/~pointche/Projets/CESAM`.

[13] D.Vianne C.Fontaine, E.Filliol. A New Fast Block Cipher Design : COS Ciphers. page pp.135, 2001.

[14] A.H. Chan, R.A. Games, and E.L Key. On the Complexities of de Bruijn Sequences. *Journal of Combinatorial Theory*, Series A 33 :pp.233–246, 1982.

[15] Taejoo Chang, Bongjoo Park, Yun Hee Kim, and Iickho Song. An Efficient Implementation of the D-Homomorphism for Generation of de Bruijn Sequences. *IEEE Transactions on Information Theory*, vol.45(4), may 1999.

[16] V. Chepyshov, T.Johansson, and B.Smeets. A simple algorithm for fast correlation attacks on stream ciphers. *Fast Software Encryption 2000 - Lecture Notes in Computer Science*, vol.1978, 2000.

[17] D. Coppersmith, H. Krawczyk, and Y. Mansour. The shrinking generator. *Advances in Cryptology - CRYPTO'93*, vol.773, 1993.

[18] N. Courtois. Higher order correlation attacks, XL algorithm and cryptanalysis of TOYOCRYPT. *Lecture Notes in Computer Science*, vol.2587, 2002.

[19] N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient algotirhms for solving overdefined systems of multivariate polynomial equations. *Advances in cryptography - EUROCRYPT 2000, Lecture Notes in Computer Science*, vol.1807 :pp.392–407, 2000.

[20] N. Courtois and W. Meier. Algebraic attacks on combiners with memory and several outputs. *Lecture Notes in Computer Science*, 2005.

[21] N.G. de Bruijn. A combinatorial problem. *Nederl. Akad. Wetensch*, Proc.49 :pp.758–764, 1946.

[22] ECRYPT. eSTREAM. `http://www.ecrypt.eu.org/stream/index.html`.

[23] Horst Feistel. Cryptography and Computer Privacy. *Scientific American*, vol.228(5) :pp.15–23, May 1973.

[24] Fluhrer and Mantin et Shamir. Weakness in the Key scheduling Algorithm of RC4. *Selected Areas in Cryptography - SAC 2001 - Lecture Notes in Computer Science*, vol.2259, 2001.

[25] Fluhrer and McGrew. Statistical Analysis of the Alleged RC4 Key stream Generator. *FSE*, 2000.

[26] Caroline Fontaine. *Contribution à la recherche de fonctions booléennes hautement non linéaires et au marquage d'images en vue de la protection des droits d'auteur.* PhD thesis, Université Paris VI, 1998.

[27] H. Fredricksen. A survey of full length nonlinear shift register cycle algorithms. *Society for Industrial and Applied Mathematics*, vol.24(2) :pp.195–221, April 1982.

[28] Berndt Gammel, Rainer Gottfert, and Oliver Kniffler. Improved Boolean Combining Functions for Achterbahn. `http://www.ecrypt.eu.org/stream/papersdir/072.pdf`.

[29] Berndt M. Gammel and Rainer Göttfert. Linear Filtering of Nonlinear Shift-Register Sequences. In *Proceedings of The International Workshop on Coding and Cryptography - WCC 2005*, March 14-18 2005.

[30] Berndt M. Gammel, Rainer Göttfert, and Oliver Kniffler. Improved Boolean Combining Functions for Achterbahn. ECRYPT Stream Cipher Project Report 2005/072, `http://www.ecrypt.eu.org/stream/papersdir/072.pdf`.

[31] Berndt M. Gammel, Rainer Göttfert, and Oliver Kniffler. Status of Achterbahn and Tweaks. `http://www.ecrypt.eu.org/stream/papersdir/2006/027.pdf`.

[32] Berndt M. Gammel, Rainer Göttfert, and Oliver Kniffler. The Achterbahn Stream Cipher, A Proposal for a PROFILE 2-Stream Cipher to ECRYPT's Call for Stream Cipher Primitives. *Symmetric Key Encryption Workshop - SKEW*, May 26-27 2005.

[33] Berndt M. Gammel, Rainer Göttfert, and Oliver Kniffler. The Achterbahn Stream Cipher, Specification, Version 1.0. *ECRYPT Stream Cipher Project Report 2005/002*, 28 april 2005.

[34] J. Golic. Cryptanalysis of alleged A5 stream cipher. *Advances in Cryptology - EUROCRYPT'97 - Lecture Notes in Computer Science*, vol.1233 :pp.239–255, 1997.

[35] Solomon W. Golomb. *Shift Register Sequences.* Aegean Park Press, 1982.

[36] G.S.Vernam. Cipher printing telegraph systems for secret wire and radio telegraphic communications. *Journalof The American Institute for Electrical Engineers*, vol.55 :pp.109–115, 1926.

[37] M.E. Hellman. A cryptanalytic time-memory trade-off. *IEEE Transaction on Information Theory*, vol.26 :pp.401–406, 1980.

[38] IEEE. P802.11 Wireless LANs Unsafe at any key size; An analysis of the WEP encapsulation. `citeseer.ist.psu.edu/558358.html`.

[39] E.Dawson J.D.Golic, A.Clark. Generalized inversion attack on nonlinear filter generators. *IEEE Transactions on Computers*, vol.49(10) :pp.1100–1109, 2000.

[40] Thomas Johansson, Willi Meier, and Frédéric Muller. Cryptanalysis of Achterbahn. `http://www.ecrypt.eu.org/stream/papersdir/064.pdf`.

[41] P. Junod. Problèmes d'implémentation. *MISC numéro 4*, novembre-décembre 2002.

[42] Pascal Junod. Cryptographic Secure Pseudo-Random Bits Generation : The Blum-Blum-Shub Generator. `http://crypto.junod.info/bbs.pdf`, 1999.

[43] ACI KAA. Knowledge Authentification Ambient project|. `http://citi.insa-lyon.fr/kaa`.

[44] A. Klapper and M. Goresky. 2-adic shift registers. *In Proc. of 1993 Cambridge Security Workshop, Fast Software Encryption*, vol.809 of Lecture Notes in Computer Science, 1994.

[45] Alexander Klimov and Adi Shamir. A New Class Of Invertible Mappings. *Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, 2002.

[46] Alexander Klimov and Adi Shamir. Cryptographic Applications of T-functions. *SAC*, 2003.

[47] Alexander Klimov and Adi Shamir. The TF-i family of stream ciphers. *The State of the Art of Stream Ciphers - SASC*, 2004.

[48] Abraham Lempel. On a Homomrphism of the de Bruijn Graph and Its Applications to the Design of Feedback Shift Registers. *IEEE Transactions on Computers*, vol.C-19(12) :pp.1204–1209, december 1970.

[49] C. Flye-Sainte Marie. Solution to problem number 58. *L'Intermédiaire des Mathématiciens*, vol.1 :pp.107–110, 1894.

[50] J.L Massey. Shift-register synthesis and BCH decoding. *IEEE Transaction on Information Theory*, vol.33 :pp.428–431, 1969.

[51] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Inc., 1997.

[52] J. Mitra and P. Sarkar. Time-Memory Trade-Off Attacks on Multiplications and T-Functions. *ASIACRYPT 2004*, pages pp.468–482, 2004.

[53] H. Molland. Improved Linear Consistency Attack on Irregular Clocked Keystream Generators. *Fast Software Encryption - FSE'04*, vol.3017 :pp.109–126, 2004.

[54] H. Molland and T. Helleseth. A linear weakness in the Klimov-Shamir T-function. *Proceedings, 2005 IEEE Int. Symposium on Information Theory*, 2005.

[55] NIST. A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. `http://csrc.nist.gov/rng/rng2.html`.

[56] NIST. Aes. `http://csrc.nist.gov/CryptoToolkit/aes`.

[57] NIST. National institute of standards and technology. `http://csrc.nist.gov`.

[58] OpenSSL. `http://www.openssl.org`.

[59] E. Pasalic, T. Johansson, S. Maitra, and P. Sarkar. New constructions of resilient and correlation immune Boolean functions achieving upper bounds on nonlinearity, 2001.

[60] R.Anderso and M.Roe. A5. `http://jya.com/crack-a5.htm`.

[61] RD2P. Operating System and Networking for Smart Objects. `http://www.lifl.fr/RD2P`.

[62] R. Rivest. The RC4 encryption algorithm. RSA Data Security, 1992.

[63] Rohitab. API Monitor.

[64] Rainer A. Rueppel. *Analysis and Design of Stream Ciphers.* Springer-Verlag, 1986.

[65] P. Sarkar and S. Maitra. Highly nonlinear balanced boolean functions with important cryptographic properties, 2000.

[66] P. Sarkar and S. Maitra. New directions in Design of Resilient Boolean Functions, 2000.

[67] RSA Security. PKCS#7. `http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7`.

[68] ACI SI SERAC. Models and protocols for SEcuRity in wireless AdhoC networks. `http://www.lifl.fr/SERAC`.

[69] C. Shannon. Communication Theory of Secrecy Systems. *Bell System Technical Journal*, vol.28(4) :pp.656–715, 1949.

[70] T. Siegenthaler. Decrypting a class of stream ciphers using ciphertext only. *IEEE Transactions on computers*, C-34(1) :pp.81–84, 1985.

[71] L. Simpson, E. Dawson, J. Golic, and W. Millan. LILI Keystream Generator. In *Selected Areas in Cryptography conference SAC'2000*, 2000.

[72] S.Leveiller, J.Boutros, P.Guillot, and G.Zémor. Cryptanalysis of nonlinear filter generators with 0, 1-metric Viterbi decoding. *Cryptography and Coding - 8th IMA International Conference - Lecture Notes in Computer Science*, vol.2260 :pp.402–414, 2001.

[73] ACI SPLASH! Securing mobile ad hoc networks. `http://www.inrialpes.fr/planete/splash.html`.

[74] Boaz Tsaban. Theorical Cryptanalysis of the Klimov-Shamir Number Generator TF-1. *IACR*, 2005.

[75] Florida State University. Diehard battery of tests of randomness. `http://www.stat.fsu.edu/pub/diehard`.

[76] Henk C.A. van Tilborg. *Encyclopedia of Cryptography and Security.* Springer, 2005.

[77] S. Vaudenay and Y. Lu. Faster Correlation Attack on Bluetooth Keystream Generator E0. *Advances in Cryptology - CRYPTO 2004, Lecture Notes in Computer Science*, LNCS vol.3152 :pp.407–425, 2004.

[78] E. Zenner, M. Krause, and S. Lucks. Improved Cryptanalysis of the self-shrinking generator. *ACISP 2004*, vol.2119 :pp.21–35, 2001.

[79] B. Zhang, H. Wu, D. Feng, and F. Bao. Security Analysis of the generalized self-shrinking generator. *ICICS'04*, vol.3269 :pp.388–400, 2004.

[80] Éric Wegrzynowski. Des trappes dans les clés. *SSTIC*, 2003.