

Quelques algorithmes de cryptanalyse du registre filtré

Sabine Leveiller

▶ **To cite this version:**

Sabine Leveiller. Quelques algorithmes de cryptanalyse du registre filtré. domain_other. Télécom ParisTech, 2004. English. NNT : . pastel-00000840

HAL Id: pastel-00000840
https://pastel.hal.science/pastel-00000840

Submitted on 25 Nov 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

TELECOM PARIS
école nationale supérieure des télécommunications

Thèse

présentée pour obtenir le grade de docteur

de l'Ecole Nationale Supérieure des Télécommunications

Spécialité : Electronique et Communications

Sabine Leveiller

Quelques algorithmes de cryptanalyse du registre filtré

Soutenue le 23 janvier 2004 devant le jury composé de

Jacques Stern	**Président**
Anne Canteaut	**Rapporteurs**
Marc Fossorier	
Thomas Johansson	**Examinateurs**
Antoine Joux	
Joseph Boutros	**Directeurs de thèse**
Philippe Guillot	
Gilles Zémor	

Remerciements

Lors de la rédaction de mon rapport de thèse, les seules lignes qui me venaient naturellement à l'esprit, fréquemment de surcroît, étaient celles qui auraient dû constituer ces remerciements. Malheureusement, toute l'énergie mobilisée jadis a été consommée, et me voilà *presque* devant l'angoisse d'une page blanche au moment de les rédiger. *Presque*, puisque j'ai quand même un grand nombre de personnes à remercier, inconditionnellement, et donc de la matière à travailler ; le seul frein est la mise en forme de ces lignes, qui sera loin, très loin croyez-moi, de celles que j'avais mentalement rédigées il y a quelques mois, et qui étaient ou plutôt me semblaient fort spirituelles. Le fantôme de génie que j'avais cru entrevoir a retrouvé sa lampe (qu'il n'avait en fait jamais quittée, mais c'est une autre histoire, Sigmund), le loup a tiré sur la bobinette et le renard a mangé son fromage. Tout s'est bien terminé ; allons donc à l'essentiel, et efforçons-nous de n'oublier personne.

Je tiens tout d'abord à remercier M. Stern de l'École Normale Supérieure, qui m'a fait l'honneur de présider mon jury de thèse. Toute ma gratitude s'adresse également à M. Joux du DCSSI ainsi qu'à M. Johansson de l'Université de Lund qui ont très gentiment accepté d'être examinateurs.

Un très grand merci à mes deux rapporteurs, dont la lecture extrêmement approfondie de ce manuscrit a permis d'en améliorer la qualité : Marc Fossorier de l'Université de Hawaii, tout d'abord, avec lequel j'ai eu la chance d'avoir d'intéressantes discussions au cours de l'été 2002, et qui, bien qu'à distance, a accepté d'endosser le rôle de rapporteur. Anne Canteaut de l'INRIA ensuite, qui, durant ces trois années de thèse, m'a soutenue et m'a fait l'honneur de s'intéresser à mes travaux. Je garde notamment un excellent souvenir des réunions « décodage itératif » que nous avons faites à l'ENST à l'initiative de Gilles Zémor, au cours desquelles j'ai beaucoup appris, et dont je ressortais en état d'hyper-motivation, rechargée à bloc et prête à affronter de nouveau les affres et les déconvenues subséquentes à la soumission d'articles de cryptographie...

J'en profiterai pour remercier – c'est de circonstance – deux autres participants : Jean-Pierre Tillich, pour ses idées, son soutien et sa gentillesse, ainsi que Gregory Olloco, qui vaque dorénavant à d'autres occupations sous des cieux plus rentables, mais peut-on l'en blâmer ?

Un très grand merci maintenant à mes directeurs de thèse, qui sont, chacun à leur façon, de véritables personnalités : Philippe Guillot, à l'origine du sujet de thèse, qui m'a convertie à la transformée de Walsh et aux joies des fonctions booléennes. Il est pour moi un modèle de

rigueur et de clairvoyance mathématiques. Merci également à Joseph Boutros, personnage haut en couleurs qui m'a fait l'honneur de m'introniser dans la Boutros-connection et dont les coups de gueule récurrents contre une administration poussiéreuse m'ont été grandement profitables. Enfin, Gilles Zémor, toujours disponible, prêt à me recevoir dans son bureau et à me faire profiter de ses connaissances, de l'intuition et du recul impressionnant qui le caractérisent, le tout sans mépris, sans orgueil ni fausse modestie : simplement. Sa gentillesse n'a d'égale que sa finesse, son honnêteté, sa rigueur et... son accent britannique.

Je tiens à remercier très sincèrement mes amis et collègues pour leurs conseils avisés et/ou leurs relectures attentives : Sandrine Fontenelle, François-Xavier Bergot, Sébastien Vincent, Philippe Painchault, Olivier Le Roch, Éric Garrido, Alain Sauzet, Olivier Orcière, Olivier Lemaire et Christine Poli.

Merci également à l'ensemble de l'équipe crypto-composants de Thales Communications, aux différents chefs d'unité, de service et de laboratoire qui se sont succédés : la liberté qui m'a été octroyée durant cette thèse CIFRE pourrait figurer dans le Guiness Book.

Un clin d'oeil à la bande des docteurs, impétrants ou accomplis, que j'ai eu le privilège de côtoyer ces trois années durant : Alireza (interprète de H$\bar{a}$fez), Amal (ostensible, voire ostentatoire), Bruno (bagouzophobe), Cat (alias Callas, l'AROP, c'est TOP), Cédric (Tupper), Céline et Bertrand (amis d'un évangéliste), Désiré (celui qui se fait), Drine et Fonto (fans des Beetles), Fwanswa (buveur de lune), FX (écoute, écoute, surtout ne fais pas de bruit), Ghaya (compagne de labeur), le Goss (chat-man herbophile), Hadi (adepte du skouairrre), Hedi (Chnah welik), Kito (the sound of silence), Loïc (swimming cool), Mariam (Iranienne à plus de 50 % américanisée), Mhmmad (t'es où ?), Olivier (Harry) Pothier (*Always* on top, surtout en été), Stefan (aimanteur de Dunhill Light), Vasolis, Manolis et Iannis (guitaris' de Moustakis), ainsi que tous les autres thésards du labo.

Un grand merci aux amis que je n'ai pas encore cités, qui ont eu la patience d'écouter mes complaintes de thésarde égarée, et qui, à la lecture de ces lignes – on peut rêver – cherchent désespérément leur nom : Bébert, Caro, Cyp, Estelle, Franck, Hélène, Karims, Mai-Anh, Marceau, Maÿlis, Nabil, Rico, Winnie, Yiol, Seb, Yiol, Seb, Yiol, Seb...

Merci à ma famille, dont la singularité est une source d'inspiration certaine.

Merci à mes parents, mon frère (ave).

Table des matières

Résumé **i**

Abstract **iii**

Abréviations **vi**

Notations **viii**

Liste des tableaux **ix**

Liste des figures **xii**

1 Présentation du système **1**
1.1 Introduction à la cryptographie 1
1.2 Récurrence linéaire - Séquence PN 3
1.3 Introduction aux codes correcteurs 8
1.4 Quelques notions sur les fonctions booléennes scalaires 12
1.5 Quelques notions de théorie de l'information 17
1.6 Lien entre la cryptanalyse du registre filtré et le décodage d'erreurs dans un mot de code bruité par un canal 20
1.7 Notations utilisées dans ce document 24

2 État de l'art sur la cryptanalyse des registres filtrés **27**
2.1 Quelques attaques déterministes 29
2.1.1 Recherche de corrélations par bloc, R. J. Anderson [1] 29
2.1.2 Attaque par inversion dans le cas d'une fonction linéaire par rapport à sa première ou sa dernière variable, J. Golic [41] 30
2.1.3 Cryptanalyse via une recherche arborescente, J. Golic *et al.* [42] 31
2.2 Attaques probabilistes 31
2.2.1 Recherche de multiples creux du polynôme générateur 33
2.2.2 Opérations sur la matrice génératrice 36
2.3 Tables comparatives de complexité 55
2.4 Conclusion 57

3 Application du décodage itératif au registre filtré **59**
3.1 Algorithme de Gallager et quelques-unes de ses versions approchées 60
3.1.1 Quelques notions sur les codes LDPC : Low-Density Parity Check codes . 61
3.1.2 Description de l'algorithme de décodage de Gallager : version α, Somme-Produit 62

3.1.3 Une première simplification de l'algorithme de Gallager : version β (Min-Sum souple) . . . 73
3.1.4 Une seconde simplification de l'algorithme de Gallager : version γ . . . 74
3.1.5 Une troisième simplification de l'algorithme de Gallager : version δ . . . 75
3.1.6 Algorithme du Min-Sum « discret » . . . 76
3.1.7 Comparaison des complexités des différentes versions de l'algorithme de Gallager . . . 77
3.1.8 Application de l'algorithme de Gallager et de ses versions approchées au registre filtré : quelques résultats de simulation . . . 78
3.2 Décodage itératif de la fonction booléenne : l'algorithme IDBF (Iterative Decoding of a Boolean Function) . . . 97
3.2.1 Description de l'algorithme IDBF . . . 97
3.2.2 Performances de l'algorithme IDBF seul . . . 101
3.2.3 Utilisation conjointe de l'IDBF et de l'algorithme B.P. . . . 104
3.2.4 Perspective : l'algorithme Gallager-IDBF appliqué à une fonction résiliente 106
3.3 Conclusion . . . 107

4 Cryptanalyse du registre filtré via l'utilisation d'un treillis 109
4.1 Algorithme du treillis déterministe . . . 109
4.1.1 Approche intuitive de l'algorithme du treillis . . . 110
4.1.2 Description de l'algorithme du treillis . . . 111
4.1.3 Première amélioration de l'algorithme du treillis : recherche de combinaisons linéaires constantes . . . 116
4.1.4 Une seconde amélioration de l'algorithme du treillis : l'algorithme Forward-Backward non intersectant (F.B.N.I.) . . . 119
4.1.5 Forward-Backward intersectant . . . 121
4.1.6 Quelques résultats de simulation . . . 123
4.1.7 Espérance du nombre de survivants . . . 125
4.1.8 Application à d'autres types d'automates . . . 128
4.1.9 Conclusion . . . 134
4.2 Treillis à métriques souples . . . 134
4.2.1 Cas d'une entrée très « éloignée » des autres : l'algorithme A.T.S.E. . . . 135
4.2.2 Cas où les espacements sont premiers entre eux . . . 139
4.3 Conclusion . . . 145

5 Le SOJA et ses dérivés 147
5.1 Hypothèses de travail - Fondement du principe SOJA . . . 148
5.2 L'algorithme SOJA : Soft Output Joint Algorithm . . . 151
5.2.1 Algorithme général . . . 151
5.2.2 SOJA-1 . . . 152
5.2.3 SOJA-2 . . . 161
5.2.4 Cas particulier des fonctions « plateau» . . . 166
5.2.5 Résultats de simulation, commentaires . . . 170
5.3 Version itérée de l'algorithme du SOJA : le SOJA itératif . . . 175
5.3.1 Description de l'algorithme itératif . . . 175
5.3.2 Résultats de simulation . . . 178
5.3.3 Les cycles de commutativité . . . 179

5.4 Algorithme «Probability-matching» : un algorithme à mi-chemin entre le SOJA et la cryptanalyse linéaire 180
5.4.1 Première approche de l'algorithme ; les équations de poids $d = 3$: un cas extrêmement favorable 182
5.4.2 Principe de l'algorithme « Probability-matching » 182
5.4.3 Calcul des probabilités associées au « Probability-matching » 183
5.4.4 Description complète de l'algorithme 187
5.4.5 Complexité de l'algorithme et calcul de la probabilité d'erreur associée au « Probability-matching » 187
5.4.6 Résultats de simulation - Comparaison des résultats expérimentaux à la théorie 192
5.4.7 Comparaison du « Probability-matching » et des algorithmes du SOJA . . 198
5.4.8 Conclusion 200
5.5 Conclusion 200

Conclusions **203**

Annexe A : fonctions booléennes utilisées **205**

Annexe B : structure de l'automate 90-150 **213**

Annexe C : codes convolutifs **215**

Annexe D : algorithme de Viterbi **217**

Annexe E : Turbo-Codes **219**

Annexe F : algorithme Forward-Backward **223**

Bibliographie **227**

Résumé

Les systèmes de chiffrement à flot (stream ciphers) sont couramment utilisés puisqu'ils permettent un chiffrement rapide des données tout en consommant peu d'énergie. Leur principe de fonctionnement est relativement simple : un générateur de pseudo-aléa produit une séquence binaire qui est X-orée au message clair, engendrant ainsi le message chiffré (cryptogramme). Le destinataire, muni d'un générateur identique, déchiffre le message reçu en lui appliquant la même opération. La sécurité de ce système repose entièrement sur la difficulté de produire la séquence pseudo-aléatoire : toute personne en mesure de la générer pourra en effet déchiffrer le cryptogramme qu'elle aura intercepté.

L'objet de cette thèse était de proposer de nouvelles attaques sur de tels systèmes. Plus précisément, nous nous sommes intéressés à la cryptanalyse à clair connu des registres filtrés : le générateur de pseudo-aléa est, dans ce cas, constitué d'un unique registre à décalage linéaire (LFSR) filtré par une fonction booléenne. La structure de ces deux composantes est supposée entièrement connue du cryptanalyste, lequel dispose, par ailleurs, de quelques bits de la séquence pseudo-aléatoire. Seule l'initialisation du registre à décalage est secrète et conditionne entièrement la sortie du générateur : elle constitue donc la clé secrète du système de chiffrement et notre objet sera précisément de la déterminer.

Le document sera organisé de la façon suivante : dans un premier chapitre, nous décrivons la structure du système de chiffrement qu'est le registre filtré, et nous intéressons aux théories auxquelles ses différentes constituantes renvoient naturellement. Nous proposons, dans un second chapitre, un état de l'art des attaques non-algébriques du registre filtré.
Suit un chapitre qui présente les fondements du décodage itératif et quelques résultats de simulation permettant d'évaluer l'impact de différents paramètres du système, celui du canal booléen notamment. Nous décrivons ensuite une nouvelle attaque dont le principe s'inspire de celui du décodage itératif, qui utilise non plus des contraintes linéaires mais celles de la fonction booléenne. Le quatrième chapitre est dédié aux attaques par treillis : nous présentons en premier lieu une attaque déterministe, qui, lorsque les conditions d'application sont réunies, permet de cryptanalyser le registre filtré très efficacement, et sans aucune probabilité d'erreur. Les conditions d'application de cet algorithme étant assez restrictives, nous avons étendu son principe à des systèmes moins spécifiques, perdant de ce fait le caractère déterministe de l'attaque originelle. Enfin, dans un dernier chapitre, nous avons développé deux algorithmes : le « SOJA » et le « Probability-Matching ». Tous deux exploitent conjointement la structure de la fonction booléenne et l'existence d'équations vectorielles, le SOJA afin de générer des probabilités *a posteriori* sur les bits de la séquence d'entrée, le Probability-Matching afin d'identifier des vecteurs d'entrée de la fonction présentant des propriétés remarquables.

Abstract

Stream-ciphers are commonly used in practical systems, because they enable fast encryption while consuming low power. Basically, these systems rely on the following : a pseudo-random sequence is produced by a generator ; it is further X-ored to the plain sequence, which is thus encrypted and can be safely sent to the receiver. The latter, who owns the same generator, decrypts the message by repeating exactly the same operations.
The purpose of this thesis was to try and cryptanalyze such generators. More precisely, the topic to be studied was that of plaintext attacks on filtered linear feedback shift registers (LFSRs). In our matter, the generator consists in a single LFSR which is filtered by a Boolean function. Basic hypothesis are that the cryptanalyst gets some bits at the output of the Boolean function, and that he knows all parameters of the encryption system, except the initial state of the LFSR. His goal is precisely to try and identify it : it constitutes the secret key on which the security of the whole encryption system solely rests.

The thesis report starts with a description of LFSRs and Boolean functions, to further move on to coding and information theories, which naturally stem from the system standard model.
A second chapter gives an overview of the main published cryptanalysis methods on filtered LFSRs (except algebraic methods).
A chapter on iterative decoding follows : it first recalls iterative decoding basics. Simulation results are then provided, enabling us to evaluate the relevance of some parameters – the Boolean channel among others. Iterative decoding principles are then extended to the decoding of a non-linear structure, namely the Boolean function, to provide us with a new cryptanalysis algorithm.
The fourth chapter is devoted to a new kind of attacks : trellis-based attacks. First, we derive a deterministic attack which proves to be very efficient when usable. It is extended to a probabilistic attack, which can in return be applied to more general systems than the original deterministic one.
In a last chapter, SOJA and "Probability-matching" - two new algorithms - are eventually presented. Both of them amount to jointly using the knowledge of the Boolean function and vector equations in order to derive information on the secret key : the SOJA generates *a posteriori* probabilities on the input bits, the "Probability-Matching" tries to identify some input vectors whose components satisfy (highly probably) constant linear relationships.

Liste des abréviations

Nous listons ci-après l'ensemble des abréviations utilisées dans ce document.

- APP : A Posteriori Probability,
- ATSE : Algorithme du Treillis avec Suppression d'une Entrée,
- AWGN : Additive White Gaussian Noise,
- BP : Belief Propagation,
- BSC : Binary Symmetric Channel, canal binaire symétrique,
- FBI : Algorithme Forward-Backward Intersectant,
- FBNI : Algorithme Forward-Backward Non Intersectant,
- IDBF : Iterative Decoding of a Boolean function,
- LDPC : Low Density Parity-check Codes,
- LFSR : Linear Feedback Shift Register,
- pce : équation de parité (parity-check equation),
- PN : Pseudo-Noise,
- SOJA : Soft Output Joint Algorithm.

Notations

Les notations utilisées tout au long de ce document sont les suivantes :

- (a) : séquence produite par le registre à décalage qui constitue l'entrée de la fonction Booléenne ;
- $A_m(a_t)$: probabilité *a posteriori* partielle du bit a_t, *i.e.* privée de l'information apportée par la m-ième équation de parité de a_t ;
- $APP(a_t)$: probabilité *a posteriori* du bit a_t ;
- $\forall \mathbf{x} \in \{0,1\}^n,\ b(\mathbf{x}) = (\widehat{f_\chi})^{d-1}(\mathbf{x})$;
- Si A est un ensemble, $\mathrm{card}(A)$ désigne le cardinal de cet ensemble ;
- $\mathbf{C_{glob}}$ est la matrice génératrice globale du code déduit du LFSR : la séquence entière s'exprime en fonction de ses K premiers bits et de cette matrice ;
- d : poids des équations de parité ;
- $\Delta_{even}(\mathbf{x}) = P[z_1 + \ldots + z_{d-1} = 0 \pmod 2 \mid \mathbf{X} = \mathbf{x},\ f(\mathbf{X}), \mathbf{X} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n}]$;
- $\widetilde{\Delta}_{even}\Big(\mathbf{X}(t)\Big) = \frac{1}{\mu}\ \mathrm{card}\Big\{m \in [1,\mu] \text{ tel que } \ z_{t+\theta_{m,1}} + \ldots + z_{t+\theta_{m,d-1}} = 0 \mod 2\Big\}$. Il s'agit de la valeur expérimentale de Δ_{even} ;
- e : équation de parité ;
- E : équation vectorielle ;
- $\mathcal{E}^b(t)$ ensemble des équations de parité (binaires) auxquelles le t-ième bit appartient ;
- $\mathcal{E}^v(t)$ ensemble des équations vectorielles auxquelles le t-ième vecteur d'entrée de la fonction, $\mathbf{X}(t)$, appartient ;
- $Extr_m(a_t)$: information extrinsèque de a_t calculée dans sa m-ième équation de parité ;
- f : fonction Booléenne scalaire, $f_\chi = (-1)^f$: fonction signe associée à f ;
- $\widehat{f}$: transformée de Fourier (dite aussi « transformée de Walsh ») de la fonction f à valeurs dans $\mathbb{Z}$;
- Φ_t : fonction indicatrice des survivants dans un treillis à l'instant t ;
- $g(X)$: polynôme de rétroaction associé au registre à décalage ;
- $\mathbf{G}$: matrice de transition $(K \times K)$ associée au registre à décalage ;
- $\forall \mathbf{z} \in \{0,1\}^d,\ \forall \mathbf{x} \in \{0,1\}^n,$
  $\gamma_{\mathbf{z}}(\mathbf{x}) = \mathrm{card}\ \Big\{(\mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}}) \text{ tels que } \ f(\mathbf{x}) = z_0,\ f(\mathbf{Y^1}) = z_1,\ \ldots,$
  $f(\mathbf{X} + \sum_{i=1}^{d-2} \mathbf{Y^i}) = z_{d-1}\ \Big\}$;
- $\Gamma^t(\mathbf{x}) = P^{(i)}\left(\mathbf{X}(t) = \mathbf{x} \mid z^E, f\right)$: pour un vecteur $\mathbf{x}$ donné, probabilité que le vecteur

d'entrée de la fonction à l'instant t, $\mathbf{X}(t)$ soit égal à $\mathbf{x}$, conditionnée par la connaissance de f, de z et de l'équation vectorielle E ;
- $\forall \mathbf{x} \in \{0,1\}^n,\ h(\mathbf{x}) = (\widehat{f_\chi})^d$;
- K : degré de $g(X)$, nombre de « cases mémoires » du registre à décalage ;
- $\kappa(z)$: complexité linéaire de la suite (z) ;
- $\mathcal{L}$: fonction de log-vraisemblance, dite aussi « log-likelihood » ;
- λ_i : espacement entre la i-ème et la $(i-1)$-ième entrée de la fonction, relativement au registre filtré : $\mathbf{X}(t) = (a_t, a_{t+\lambda_0}, a_{t+\lambda_0+\lambda_1}, \ldots, a_{t+\sum_{i=0}^{n-2}\lambda_i})$;
- Λ est la mémoire du système. C'est le nombre minimal de décalages (« clock ») réguliers à effectuer pour qu'un bit de la séquence d'entrée (a), initialement à la première entrée de la fonction, sorte de la dernière entrée ;
- relation d'ordre de Lucas : $\forall\ \mathbf{t}, \mathbf{s} \in \{0,1\}^n, \mathbf{t} \preceq \mathbf{s} \Longleftrightarrow \forall\ 0 \leq i \leq n-1,\ \ t_i \leq s_i$;
- M : nombre total d'équations de parité ;
- μ_b : nombre moyen d'équations par bit ;
- μ_v : nombre moyen d'équations vectorielles par vecteur ;
- n : nombre d'entrées de la fonction f ;
- N : longueur de séquence que l'on utilise (en sortie de la fonction) pour cryptanalyser le système ;
- nb_{iter} : nombre d'itérations dans un algorithme itératif ;
- $Obs(a_t)$: observation du bit a_t ;
- p : probabilité de transition d'un canal binaire symétrique ;
- r_f : fonction d'auto-corrélation de la fonction f ;
- Supp(f) désigne le support d'une fonction $f\ : A \longrightarrow B\ : Supp(f) = \{x \in A \text{ tq } \ f(x) \neq 0\}$;
- $S_m(t)$: indices des bits intervenant dans la m-ième équation du bit a_t ;
- $S'_m(t)$ indices des bits intervenant dans la m-ième équation du bit a_t privée de t : $S'_m(t) = S_m(t) \backslash \{t\}$;
- $Sp(f)$ désigne le spectre d'une fonction $f\ : A \longrightarrow B : Sp(f) = \{\widehat{f}(\mathbf{u})\}_{\mathbf{u} \in A}$;
- $\mathbf{X}(t) = (X_1, \ldots, X_n)$ vecteur d'entrée de la fonction f à l'instant t ;
- $\mathcal{X}(t)$: ensemble des vecteurs d'entrée de la fonction auxquels le t-ième bit, a_t, appartient ;
- $\Xi_{\ell_\mathbf{u}}(t)$: probabilité que la forme linéaire du vecteur d'entrée $\mathbf{X}(t) \longmapsto \mathbf{u}.\mathbf{X}(t)$ soit égale à 1, conditionnée par la connaissance de la fonction f, de la séquence reçue (z) et des équations vectorielles vérifiées par $\mathbf{X}(t)$, *i.e.* $\Xi_{\ell_\mathbf{u}}(t) = P\Big(\mathbf{u}.\mathbf{X}(t) = 1 \mid f, (z), \mathcal{E}^v(t)\Big)$;
- (z) : séquence disponible en sortie de la fonction booléenne, résultant du filtrage par f de la séquence (a) ;
- z^E : ensemble des bits de la séquence (z), en sortie de la fonction f, dont les antécédents sont liés par l'équation vectorielle E ;
- $z^\mathcal{E} = \bigcup_{E\ \in\ \mathcal{E}^v(t)} z^E$. La dépendance temporelle de $z^\mathcal{E}$ existe bel et bien, mais nous l'omettons pour simplifier les notations.

Liste des tableaux

1.1 Table des valeurs d'une fonction à $n = 3$ entrées. 12

2.1 Algorithme de recherche de corrélations par bloc de R.J. Anderson, FSE 94 [1]. . 30
2.2 Attaque déterministe sur une fonction linéaire par rapport à sa première ou sa dernière variable, J. Golic, FSE 96 [41]. 31
2.3 Attaque déterministe via une recherche arborescente, J. Golic *et. al.*, IEEE Transactions on Computers [42]. 32
2.4 Algorithme de W. Meier et O. Staffelbach, Journal of Cryptology 89, [66]. 36
2.5 Algorithme de recherche par table des multiples de poids d d'un polynôme, A. Canteaut et M. Trabbia, Eurocrypt 2000, [10]. 37
2.6 Algorithme du « list-decoding », M.J. Mihaljevìc *et. al.*, FSE 2001, [74]. 41
2.7 Algorithme de décodage itératif dur, M.J. Mihaljevìc *et. al.*, FSE 2000, [72]. . . . 43
2.8 Extraction d'un code convolutif, et attaque via l'algorithme de Viterbi, Johansson et Jönsson, Eurocrypt 99, [47]. 46
2.9 Décodage à maximum de vraisemblance d'un sous-code, Chepyzhov *et. al.*, FSE 2000, [16]. 48
2.10 Algorithme de reconstruction polynomiale n'utilisant pas la notion de i-préfixe, Johansson et Jönsson, CRYPTO 2000, [49]. 51
2.11 Deuxième algorithme de cryptanalyse utilisant la notion de i-préfixe, Johansson et Jönsson, Crypto 2000, [49]. 52
2.12 Algorithme optimisé de recherche d'équations de parité, Chose, Joux et Mitton, Eurocrypt 2002, [17]. 54
2.13 Résultats obtenus pour $N = 400000$ et $t = 2$. 56
2.14 Résultats obtenus pour $N = 40000$ et $t = 3$. 56
2.15 Reconstruction polynomiale, $p = 0,40$, $N = 400000$, $t = 2$. 56

3.1 Algorithme de Gallager, version α. 69
3.2 Algorithme de Gallager version α, écriture logarithmique. 72
3.3 Algorithme de Gallager version β, Min-Sum souple. 74
3.4 Algorithme de Gallager version γ. 75
3.5 Algorithme de Gallager version δ. 76
3.6 Algorithme de Gallager version « discrète ». 78
3.7 Complexités comparées des différentes versions de l'algorithme de Gallager sous l'hypothèse $N, \mu_b \gg d$. 78
3.8 Taux de succès des différents algorithmes B.P. en fonction de la probabilité du canal BSC(p), avec $N = 10000$, $K = 100$, poids des équations $d = 3$, $\mu_b \approx 17.2$ équations par bit. 86

3.9 Taux de succès des différents algorithmes B.P. en fonction de la probabilité du canal BSC(p) avec $N = 20000$, $K = 100$, poids des équations $d = 3$, $\mu_b \approx 20.2$ équations par bit. 86
3.10 Description de l'algorithme IDBF. 100
3.11 Probabilité d'erreur obtenue en appliquant l'IDBF seul à une fonction à $n = 4$ entrées (0101010011110100) ; $K = 100$. 102
3.12 Probabilité d'erreur obtenue en appliquant l'IDBF seul à la fonction f_5 à $n = 5$ entrées ; $K = 100$. 103
3.13 Probabilité d'erreur obtenue en appliquant l'IDBF seul à la fonction f_9 à $n = 6$ entrées ; $K = 100$. 103
3.14 Résultats de l'IDBF-Gallager, $K = 100$, poids des équations $d = 3$. 105

4.1 Algorithme du treillis - version forward. 112
4.2 Table des valeurs d'une fonction à $n = 4$ entrées. 113
4.3 Algorithme du treillis généralisé à la recherche de formes linéaires constantes sur l'ensemble des survivants. 118
4.4 Algorithme Forward-Backward (non intersectant). 120
4.5 Algorithme Forward-Backward Intersectant. 122
4.6 Résultats de simulation obtenus sur un registre de mémoire $K = 100$ filtré par différentes fonctions : longueur minimale nécessaire pour identifier K formes linéaires indépendantes. Entre parenthèses figure le nombre moyen de survivants obtenus. 124
4.7 Transitions d'états pour une fonction à $n = 3$ entrées. 126
4.8 Espérance du nombre de survivants : théorie vs expérience. 128
4.9 Longueur minimale nécessaire à l'identification de l'initialisation d'un automate cellulaire 90-150 de longueur 64 en appliquant l'algorithme forward-backward intersectant afin d'identifier des formes quadratiques constantes de l'état initial. Les quantités entre crochets représentent la longueur minimale requise à l'identification de K combinaisons linéairement indépendantes dans le cas d'un LRSR de même longueur. 133
4.10 Application d'un algorithme aller-retour après suppression d'une entrée : algorithme A.T.S.E. 136
4.11 Table de vérité d'une fonction booléenne à $n = 3$ entrées. 137
4.12 Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_8, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$. 138
4.13 Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_{12}, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$. 138
4.14 Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_1, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$. 139
4.15 Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 6$ entrées, f_9, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = \lambda_3 = 1$, $\lambda_4 = K - 5$. 139
4.16 Algorithme du treillis utilisant une équation de parité. 141
4.17 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_1, une seule équation de parité est utilisée. 142
4.18 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_7, une seule équation de parité est utilisée. 143

4.19 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée f_1, deux équations de parité sont utilisées. 143
4.20 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_7, deux équations de parité sont utilisées. 143
4.21 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction f_1, les informations générées sur le premier treillis sont injectées dans le second. 143
4.22 Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction f_7, les informations générées sur le premier treillis sont injectées dans le second. 143

5.1 Algorithme général du SOJA. 152
5.2 Table de vérité d'une fonction à $n = 3$ entrées. 153
5.3 Comparaison de différentes stratégies de cryptanalyse incluant le SOJA. Paramètres du système testé : $K = 40$, équations de poids $d = 5$, fonction à $n = 7$ entrées, 3-résiliente, « plateau ». 172
5.4 Comparaison de différentes stratégies de cryptanalyse incluant le SOJA ; paramètres du système testé : $K = 100$, équations de poids $d = 3$, fonction à $n = 8$ entrées, 2-résiliente, « plateau ». 173
5.5 Algorithme du SOJA itératif. 176
5.6 Construction de la table $\mathcal{T}$ de la fonction f_1 à $n = 5$ entrées, équations de poids $d = 5$. 186
5.7 Algorithme « Probability-matching ». 188
5.8 Construction de la table $\mathcal{T}$ de la fonction f_{13} à $n = 7$ entrées. Poids des équations : $d = 5$. 197
5.9 Résumé des complexités en calculs des algorithmes SOJA et du « Probability-matching ». 199
5.10 Résumé des complexités en mémoire des algorithmes SOJA et du « Probability-matching ». 199
5.11 Algorithme de Viterbi. 218
5.12 Algorithme Forward-Backward. 225

Table des figures

1.1 Registre filtré par une fonction booléenne. 3
1.2 Combinaison linéaire du type $\ell_{\mathbf{u}}(x) = 1 + x^2 + x^3$ appliquée à chaque vecteur d'entrée, produisant la séquence $a^{\ell_{\mathbf{u}}}$ où $\mathbf{u} = (10110)$. 5
1.3 Chaîne de transmission classique. 8
1.4 Capacité d'un canal binaire symétrique sans mémoire en fonction de sa probabilité de transition. 19
1.5 Modélisation du système de chiffrement LFSR - fonction booléenne en un code correcteur bruité par un BSC. 21

2.1 Un registre unique filtré par une fonction booléenne. 28
2.2 Un générateur par combinaison de registres. 28
2.3 BSC(p) vs fonction booléenne. 33

3.1 Modélisation de la chaîne d'émission dans le cadre d'une transmission d'information classique. 62
3.2 Mouvements horizontaux de l'algorithme de Gallager dans la matrice de parité du code lors du calcul des probabilités extrinsèques. 68
3.3 Mouvements verticaux de l'algorithme de Gallager dans la matrice de parité du code lors du calcul des probabilités *a posteriori*. 68
3.4 Propagation ascendante de l'information le long du graphe de Tanner du code. . 70
3.5 Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_3 à $n = 5$ entrées. 83
3.6 Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_4 à $n = 5$ entrées. 84
3.7 Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_{14} à $n = 7$ entrées. 85
3.8 Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 100$, $d = 3$, équations générées par élévations au carré successives. . 88
3.9 Comparaison des taux de convergence des algorithmes de Gallager version δ et Min-Sum sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 100$, $d = 3$. 88
3.10 Comparaison des taux de convergence de de l'algorithme de Gallager version δ sur BSC($p = 0.375$) et sur deux fonctions de même probabilité de transition équivalente, f_2 à $n = 5$ entrées, f_{13} à $n = 7$ entrées ; $K = 100$, $d = 3$. 89
3.11 Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 21$, $d = 3$, équations générées par l'algorithme Canteaut-Trabbia [10]. 90
3.12 Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 21$, $d = 3$. 90

3.13 Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.375$) et sur deux fonctions de même probabilité de transition équivalente, f_2 à $n = 5$ entrées, f_{13} à $n = 7$ entrées ; $K = 21$, $d = 3$. 91
3.14 Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 120$, $d = 5$, équations générées par élévations au carré successives. . 92
3.15 Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 120$, $d = 5$. 93
3.16 Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 40$, $d = 5$, équations générées par l'algorithme Canteaut-Trabbia [10]. 94
3.17 Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 40$, $d = 5$. 95
3.18 Représentation de l'algorithme IDBF sous la forme d'un arbre. 98
3.19 Principe de l'IDBF, décodeur itératif de la fonction booléenne. 104
3.20 Schéma complet de cryptanalyse à partir de l'IDBF. 104
3.21 Décodage IDBF-Gallager. 105

4.1 Évolution du vecteur d'entrée d'une fonction à $n = 4$ entrées entre deux instants consécutifs lorsque les espacements λ_i sont tous égaux à 1. 110
4.2 Treillis d'une fonction à $n = 4$ entrées. 113
4.3 Déroulement de l'algorithme du treillis sur une fonction à $n = 4$ entrées. 114
4.4 Exemple d'un automate cellulaire 90-150 à cinq cellules. 130
4.5 Déroulement de l'algorithme A.T.S.E sur une fonction à $n = 3$ entrées initialement. 138

5.1 Registre filtré par une fonction booléenne. 148
5.2 Mise en évidence d'une équation vectorielle de poids $d = 3$ sur un registre filtré représenté de façon équivalente sous la forme d'un générateur par combinaison. . 150
5.3 Détection du vecteur nul en utilisant le SOJA-1 sur des équations vectorielles de poids $d = 3$ que satisfont les vecteurs d'entrée de la fonction booléenne. 160
5.4 Représentation des différentes stratégies de décodage comparées : algorithme BP appliqué sur une séquence bruitée par un BSC ou par une fonction booléenne et algorithmes du SOJA suivis d'un décodage BP ou d'une détection à seuil. 171
5.5 Différentes étapes du calcul des probabilités à l'itération i dans l'algorithme SOJA-itératif. 177
5.6 Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_1 en utilisant des équations de parité de poids $d = 3$. 178
5.7 Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_9 en utilisant des équations de parité de poids $d = 3$. 179
5.8 Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_{15} en utilisant des équations de parité de poids $d = 3$. 179
5.9 Cycles de commutativité dans le graphe des vecteurs d'entrée lors de l'application du SOJA-itératif. 181
5.10 Variations de la distance de Kullback D_K entre une loi binomiale d'espérance $P_1 = \frac{1}{2}$ et une loi binomiale dont l'espérance P_2 varie. 193
5.11 Illustration de la première stratégie de cryptanalyse de la fonction f_1 en utilisant l'algorithme du « Probability-matching ». 193
5.12 Probabilités d'erreur de l'algorithme « Probability-matching » en fonction du nombre d'équations vectorielles par vecteur, obtenues sur la fonction f_1 à $n = 5$ entrées. 197

5.13 Probabilités d'erreur de l'algorithme « Probability-matching » en fonction du nombre d'équations vectorielles par vecteur, obtenues sur la fonction f_{13} à $n = 7$ entrées. 198
5.14 Code convolutif (7,5), mémoire $B = 3$. 215
5.15 Structure d'un turbo-code parallèle. 220
5.16 Structure d'un turbo-code série. 220
5.17 Passage de la séquence (x) à travers un canal sans mémoire. 224

Chapitre 1

Présentation du système

Ce document traite de la « cryptanalyse d'un registre à décalage filtré par une fonction booléenne ». À toute personne non initiée, ce vocabulaire paraîtra sans doute quelque peu abscons. Nous nous proposons, dans ce chapitre, d'éclairer le lecteur en répondant aux questions suivantes : comment ce système se situe-t-il dans un contexte cryptographique ? Quel est-il ? Quels sont les éléments qui le constituent et à quelles théories renvoient-ils ?

Le système du registre à décalage filtré par une fonction booléenne est couramment modélisé dans la littérature en un codeur dont la sortie est bruitée par un canal binaire symétrique. Nous avons donc choisi d'exposer la structure de ce système de chiffrement de la façon suivante : après avoir situé le système du registre filtré dans un contexte cryptographique plus large, nous présentons le registre à décalage, étudions quelques-unes de ses propriétés et abordons ensuite les bases de la théorie des codes correcteurs. Suit un paragraphe traitant des fonctions booléennes scalaires ; sa vocation est de souligner les caractéristiques des fonctions booléennes qui conditionnent le succès des cryptanalyses existantes ou présentées ultérieurement dans ce document. Le modèle du canal de transmission d'information renvoie à la théorie de l'information, dont nous présenterons brièvement les bases dans un nouveau paragraphe. Nous détaillerons ensuite la modélisation dont le registre filtré fait l'objet, et soulignerons les propriétés essentielles qu'il se doit de vérifier. Les notations utilisées dans les chapitres suivants seront finalement exposées dans le dernier paragraphe.

1.1 Introduction à la cryptographie

La cryptographie [83, 99] permet l'échange confidentiel de messages. L'émetteur, A, envoie un message à B ; s'il craint que des espions n'interceptent le message, il doit s'assurer que ceux-ci ne pourront en extraire aucune information. La cryptographie intervient à ce niveau : A peut à l'aide de certaines techniques de chiffrement produire un cryptogramme, que seul B, muni de la clé de chiffrement, pourra déchiffrer.

Les algorithmes de chiffrement peuvent être classés en deux catégories : algorithmes à clé publique, algorithmes à clé secrète.

Dans le cadre d'une transmission de données utilisant un algorithme à clé publique, chaque personne reçoit un couple de clés, l'une publique, l'autre secrète et propre à chaque utilisateur. La clé publique est liée mathématiquement à la clé privée. Un exemple de tel algorithme est le RSA. À l'inverse, les algorithmes à clé secrète ne disposent que d'une seule clé dont la connais-

sance est partagée entre émetteur et récepteur(s). Il existe deux sous-catégories d'algorithmes à clé secrète : les algorithmes de chiffrement par bloc, comme le DES, l'AES, ou les algorithmes de chiffrement à flot continu, « stream cipher », auxquels nous nous intéressons dans toute la suite de ce document. Les algorithmes de chiffrement par blocs segmentent les données à chiffrer en blocs de taille donnée ; ces derniers subissent une suite d'opérations de chiffrement qui se déroulent sur plusieurs étages, appelés rondes (rounds en anglais).
À l'inverse, les systèmes de chiffrement à flot chiffrent le message bit à bit, en ajoutant à chaque bit un bit d'une séquence de clé. À la réception, le message est déchiffré en ajoutant au cryptogramme la même séquence de clé, et le message clair est ainsi identifié. Les principaux avantages que présentent les systèmes de chiffrement à flot sont les suivants : les erreurs de transmission ne sont pas propagées puisque le message est chiffré bit à bit. Ces systèmes sont par ailleurs relativement simples et consomment peu d'énergie, comparativement aux systèmes de chiffrement par blocs qui opèrent sur plusieurs rondes.
Le système originel de chiffrement agissant bit à bit est le « one-time pad », qui consiste à ajouter bit à bit une séquence de clé tirée aléatoirement. La séquence de clé est alors de taille identique à celle du message, ce qui rend l'algorithme inconditionnellement sûr mais peu pratique. Ainsi, l'idée naturelle fut d'opter pour des générateurs produisant des séquences pseudo-aléatoires de période la plus élevée possible : les automates linéaires tels le LFSR furent, à cet égard, de bons candidats. Ces générateurs, seuls, ne sont cependant pas des objets cryptographiques : leur transition est linéaire et l'observation d'un nombre relativement faible de bits permet de déterminer entièrement la séquence pseudo-aléatoire produite. Il fallait donc ajouter à ces générateurs d'aléa une composante non-linéaire : la fonction booléenne, dont la sortie constitue la séquence de clé. Les propriétés cryptographiques du système peuvent alors être très bonnes, pour une taille de fonction relativement faible (le nombre d'entrées est de l'ordre de la dizaine).
Le système du registre à décalage filtré non-linéairement par une fonction est toutefois un « cas d'école » au sens où il n'est généralement pas utilisé seul mais plutôt comme brique de base d'un système cryptographique plus complexe. Il peut aussi servir de modèle simplifié d'un système plus élaboré.

À l'opposé de ces procédés de chiffrement se trouve la cryptanalyse, qui consiste à endosser le rôle d'un espion, tenter de déchiffrer le message sans avoir la connaissance complète des paramètres qui ont permis de le chiffrer. Il existe principalement trois méthodes de cryptanalyse :

- cryptanalyse à chiffré seul : en plus de la connaissance sur le système plus ou moins complète dont il dispose, le cryptanalyste ne possède qu'un chiffré ;
- cryptanalyse à clair connu : le cryptanalyste possède, dans ce cas, un couple clair chiffré ;
- cryptanalyse à clair choisi : le cryptanalyste a alors la liberté de chiffrer des messages qu'il choisit, et d'observer le cryptogramme correspondant.

Dans ce document, nous nous sommes intéressés à la **cryptanalyse à clair connu d'un unique registre à décalage linéaire filtré par une fonction booléenne** : nous supposons avoir réussi à identifier une certaine quantité de bits de la séquence (z) en sortie de la fonction, et cherchons à identifier **la clé secrète** du système qui n'est autre que **l'initialisation du registre à décalage** (la séquence (z) étant la séquence de bits de clé, elle peut être obtenue en ajoutant bit à bit le message clair et la séquence chiffrée correspondante qui serait interceptée).

Dans les paragraphes qui suivent, nous avons choisi de présenter chacune des constituantes du système qui est schématisé sur la figure 1.1 : le LFSR qui fait appel à la théorie des codes correcteurs, puis la fonction booléenne qui, elle, suggère l'introduction de quelques notions de

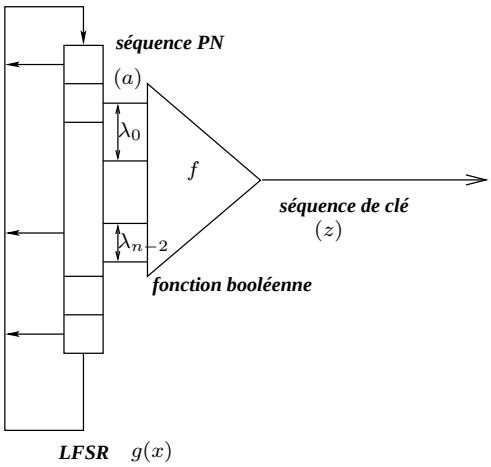


Fig. 1.1 – Registre filtré par une fonction booléenne.

théorie de l'information.

Enfin, nous serons amenés à introduire l'ensemble des notations qui seront utilisées dans la suite de ce document.

1.2 Récurrence linéaire - Séquence PN

Le lecteur peut consulter les références suivantes pour plus de détails [43, 59, 80].
Définition : Une **séquence PN** est une suite engendrée par un registre linéaire ayant K bascules, de période maximale : $T = 2^K - 1$. Les séquences PN sont également appelées ***m*-séquences**.

Une séquence PN est donc associée à une récurrence linéaire du type :

$$\forall t > K,\ a_t = \sum_{i=1}^{K-1} g_i a_{t-i}\ . \tag{1.1}$$

À la suite ainsi générée est associé le **polynôme de rétroaction**, $g(x)$, de la suite :

$$g(x) = \sum_{i=0}^{K} g_i x^i\ , \tag{1.2}$$

où la multiplication par x correspond à un **retard temporel** d'une unité. L'action d'un polynôme p sur la suite (a) sera notée : $p.(a)$. Dans le cas particulier du polynôme de rétroaction, on a $g.(a) = 0$.

Exemple 1 *La suite* (a) *ayant comme polynôme de rétroaction*

$$g(x) = 1 + x + x^3$$

satisfait la récurrence :

$$g.(a) = 0 \Longrightarrow \forall t \geq 3,\ a_t = a_{t-1} + a_{t-3}\ .$$

Supposons que $(a_0, a_1, a_2) = (1, 0, 0)$ *alors la séquence produite par le générateur est périodique de période* $T = 2^3 - 1 = 7$ *et on a :*

$$(a) = (1, 0, 0, 1, 1, 1, 0, 1, 0, 0 \ldots)$$

Il possède la propriété suivante :

Proposition 1 *La suite produite par une récurrence de polynôme de rétroaction* $g(x)$ *est une séquence PN si et seulement si* $g(x)$ *est un polynôme primitif.*

De par la récurrence 1.1, l'état initial du générateur, *i.e.* les K premiers bits de la séquence déterminent entièrement la séquence de longueur $2^K - 1$.

Dans la suite, nous supposerons que le polynôme associé au LFSR est choisi primitif. Citons alors quelques propriétés satisfaites par la séquence :

Proposition 2 *Considérons un LFSR. Alors,*
- *deux états initiaux différents produisent deux suites qui sont en fait des versions décalées (cycliquement) l'une de l'autre. Ainsi, l'observation de* K *bits consécutifs de la séquence PN, permet de retrouver l'état initial de ce dernier ;*
- *toute combinaison linéaire de décalées d'une séquence PN est encore une séquence PN qui satisfait la même récurrence.*

Notons que toute combinaison linéaire des bits de la suite (a) a un correspondant polynomial, au même titre que le polynôme caractéristique précédemment cité.

Exemple 2 *La figure 1.2 donne l'exemple de la combinaison linéaire :* $\ell_{\mathbf{u}}(x) = 1 + x^2 + x^3$ *qui se traduit en une somme de décalées de la séquence* (a).
Soit $(a^{\ell_{\mathbf{u}}})$ *la séquence obtenue en appliquant* $\ell_{\mathbf{u}}$ *à* (a).
$(a^{\ell_{\mathbf{u}}})$ *satisfait la récurrence :*

$$a_t^{\ell_{\mathbf{u}}} = a_t + a_{t+2} + a_{t+3}.$$

Théorème 1 *Si* $g(x)$ *est un polynôme irréductible sur* $GF(2)$ *de degré* K*, alors l'ensemble des polynômes en* x *de degré strictement inférieur à* K *et à coefficients dans* $GF(2)$*, assorti d'opérations effectuées modulo* $(g(x))$ *forme un corps d'ordre* 2^K.

Se référer à [61] par exemple pour plus de détails.

Ces propriétés nous seront très utiles dans toute la suite de ce document. Nous en déduirons en effet les propriétés suivantes :
- toute séquence résultant d'une **combinaison linéaire des bits de la séquence initiale** (a) satisfait les **mêmes relations de récurrence** que la séquence dont elle est issue ;
- décoder une **combinaison linéaire** de la séquence initiale nous permet **sans aucune difficulté** de remonter à la **séquence initiale** (a), puisqu'elle-même s'exprime comme une combinaison linéaire de la séquence décodée de par le théorème 1 ;
- tout bit de la séquence (a) s'exprime comme une **combinaison linéaire des** K **premiers bits** $a_0, \ldots, a_{K-1}$. Ainsi, décoder K **bits quelconques linéairement indépendants** nous permet d'identifier $a_0, \ldots, a_{K-1}$ en inversant un système linéaire $(K \times K)$.

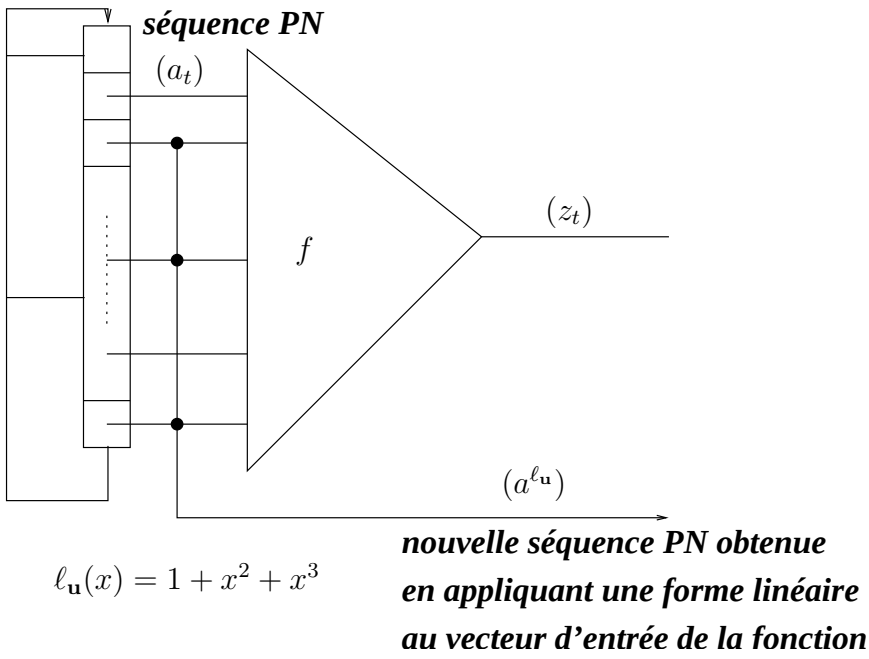


FIG. 1.2 – Combinaison linéaire du type $\ell_{\mathbf{u}}(x) = 1+x^2+x^3$ appliquée à chaque vecteur d'entrée, produisant la séquence $a^{\ell_{\mathbf{u}}}$ où $\mathbf{u} = (10110)$.

L'état interne du LFSR peut être représenté sous la forme d'un vecteur

$$\mathbf{a_t} = (a_t, a_{t+1}, \ldots, a_{t+K-1}).$$

L'évolution de cet état est régie par la matrice suivante, dite **matrice de transition** associée au LFSR :

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & \ldots & 0 & g_K \\ 1 & 0 & \ldots & 0 & g_{K-1} \\ 0 & 1 & & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & g_1 \end{pmatrix}.$$

À chaque instant t, on a la relation suivante :

$$\mathbf{a_t} = \mathbf{a_{t-1}}\ \mathbf{G}\ = \mathbf{a_0}\ \mathbf{G}^t\ .$$

Supposons maintenant que nous ayons observé une suite PN, sans connaître le générateur qui l'a produite. La complexité linéaire, abordée ci-dessous, permet de caractériser cette suite en évaluant la longueur minimale d'un registre qui l'aurait générée.

Définition : La **complexité linéaire** d'une suite est la mémoire minimale nécessaire à la reproduction de la suite par un automate linéaire. Il s'agit également de la dimension de l'espace vectoriel engendré par la suite et l'ensemble de ses décalées.

Considérons le système du registre filtré décrit sur la figure 1.1. Il est constitué de deux éléments : un registre à décalage linéaire et une fonction booléenne. Certaines « cases mémoire »

du registre sont prélevées pour former les bits d'entrée de la fonction booléenne. La sortie de cette fonction est la séquence de bits de clé, lesquels sont ajoutés à la séquence claire pour produire le message chiffré. Quiconque parvient à générer cette séquence de clé (à partir de l'observation de quelques-uns de ses bits par exemple) pourra entièrement décrypter le message transmis.
Ainsi, il est souhaitable que la complexité linéaire en sortie de la fonction soit la plus élevée possible. Donnons quelques propriétés sur la complexité linéaire, notée κ.

Proposition 3 *Si (a) et (b) sont deux suites de complexité linéaire respective $\kappa(a)$ et $\kappa(b)$, alors*

$$\kappa(a+b) \leq \kappa(a) + \kappa(b) \tag{1.3}$$
$$\kappa(ab) \leq \kappa(a)\kappa(b) \tag{1.4}$$

Preuve :
Afin de souligner l'intérêt de considérer la complexité linéaire d'une suite comme la dimension de l'espace vectoriel engendré par cette suite et ses décalées, nous démontrons l'inégalité (1.4). Cette démonstration est due à E. Garrido [37].

Soit (a), (b) deux suites.
Notons D l'opérateur décalage. $D^i.(a)$ est la i-ème décalée de (a) : $D^i.(a) = (a_{t+i}, a_{t+1+i}, \ldots)$.
Alors, d'après la définition de la complexité linéaire,

$$\begin{aligned} \kappa(a) &= \dim\Big(Vect\Big((a), D.(a), D^2.(a), \ldots\Big)\Big) \\ \kappa(b) &= \dim\Big(Vect\Big((b), D.(b), D^2.(b), \ldots\Big)\Big) \end{aligned}$$

Soit $(A_1, \ldots, A_{\kappa(a)})$, $(B_1, \ldots, B_{\kappa(b)})$ les bases de $\Big(Vect\Big((a), D.(a), D^2.(a), \ldots\Big)\Big)$ et $\Big(Vect\Big((b), D.(b), D^2.(b), \ldots\Big)\Big)$.
Alors $\forall i$,

$$\begin{aligned} D^{(i)}(a.b) &= (D^{(i)}(a).D^{(i)}(b)) \\ &= \Big(\sum_{i=1}^{\kappa(a)} \alpha_i A_i\Big)\Big(\sum_{j=1}^{\kappa(b)} \beta_j B_j\Big) \\ &= \sum_{i=1}^{\kappa(a)} \sum_{j=1}^{\kappa(b)} \alpha_i \beta_j A_i B_j \end{aligned}$$

Ainsi, toute suite obtenue en décalant la suite $(a.b)$ s'exprime comme une combinaison linéaire de $A_i B_j$. La famille $(A_i B_j)_{i=1,j=1}^{\kappa(a),\kappa(b)}$ est donc une famille génératrice de l'espace vectoriel engendré par la suite $(a.b)$ et ses décalées. La dimension de cet espace vectoriel est inférieure ou égale à la dimension de la famille génératrice, et donc $\kappa(ab) \leq \kappa(a)\kappa(b)$.

∎

Dans notre cas, les complexités linéaires de chaque suite en entrée de la fonction sont identiques, égales à K (puisqu'elles sont simplement des versions décalées les unes des autres).

Proposition 4 *Borne de Key*
Soit un générateur PN de mémoire K, filtré par une fonction à n entrées.

Alors, la complexité de la séquence (z) en sortie de la fonction vérifie l'inégalité suivante

$$\kappa(z) \leq \sum_{i=1}^{n} C_K^i \ . \tag{1.5}$$

Proposition 5 *Soit $d < K$ tel que $\forall i < d$, i est premier avec K.*
Alors une fonction booléenne de degré d tirée au hasard engendrera une suite de complexité linéaire maximale avec la probabilité :

$$P = \left(1 - \frac{1}{2^K}\right)^{\frac{B_d}{n}} \tag{1.6}$$

avec $B_d = \sum_{i=0}^{d} C_K^d$.

Le cas où les espacements entre les entrées de la fonction sont réguliers permet d'énoncer quelques propriétés sur la complexité linéaire.

Proposition 6 *Soit $\delta > 0$ tel que $pgcd(\delta, 2^K - 1) = 1$, soit (a) une séquence PN. Soit (z) la suite obtenue en faisant le produit de n termes décimés de δ : $z_t = a_t a_{t+\delta} \dots a_{t+(n-1)\delta}$, alors la complexité linéaire de cette suite est bornée par :*

$$\kappa(z) \geq C_K^n.$$

Proposition 7 *Soit $\delta > 0$ tel que $pgcd(\delta, 2^K - 1) = 1$, (a) une séquence PN.*
Soit (z) la séquence résultant de la combinaison linéaire de produits d'ordre m du type : $a_t a_{t+\delta} a_{t+m\delta}$

$$z = \sum_{t=0}^{n} c_t a_t a_{t+\delta} \dots a_{t+m\delta}.$$

La complexité linéaire de (z) satisfait l'inégalité suivante :

$$\kappa(z) \geq C_K^m - (n-1).$$

Les preuves des propositions 4, 5, 6 et 7 sont données dans [80].
Ces propriétés de complexité linéaire nous permettront ultérieurement d'en déduire des critères de conception des fonctions booléennes.

La modélisation du système du registre filtré qui est couramment faite est celle d'un mot de code bruité par un canal de transmission. Afin de mieux cerner ce modèle, nous nous intéressons aux codes correcteurs dans la partie qui suit.

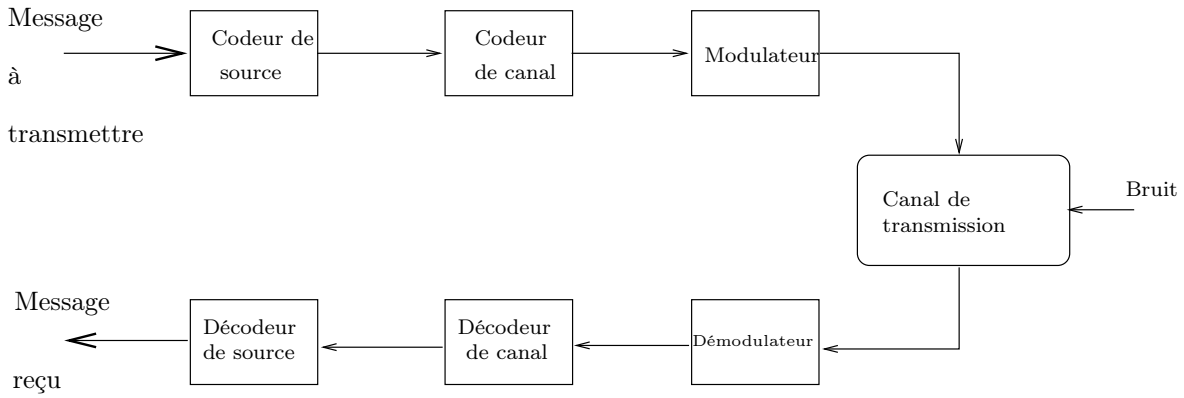


Fig. 1.3 – Chaîne de transmission classique.

1.3 Introduction aux codes correcteurs

Les codes correcteurs [61] sont utilisés dans le cadre d'une transmission d'information au cours de laquelle le message émis peut être altéré par du bruit. Nous avons schématisé une chaîne de transmission classique sur la figure 1.3.

La première étape de la transmission est celle du codage de source, qui comprime le message, supprime la redondance inutile afin d'optimiser la taille de la version codée de ce dernier. En prévision de possibles distorsions induites par le canal de transmission, le codeur de canal rajoute quant à lui certains bits, de telle sorte que la correction d'un nombre restreint d'erreurs est rendue possible en réception. Le modulateur transforme le mot de code en une forme d'onde émissible sur le canal.

Il existe deux types de codes correcteurs : les codes en blocs auxquels nous nous intéressons dans ce chapitre et les codes en treillis, dont un exemple est donné en annexe (voir annexe C sur les codes convolutifs).

Nous nous intéressons ici aux codes binaires mais ces définitions sont généralisables dans le cas où l'alphabet possède plus de deux symboles.

Définitions et propriétés essentielles des codes linéaires :

Définition : Un **code en bloc binaire** de taille M est un ensemble de mots de longueur n.

Généralement M est de la forme 2^k et le code est désigné par le couple (n, k).

À tout code, on associe le **rendement** R :

$$R = \frac{k}{n} \tag{1.7}$$

Définition : La **distance minimale**, d_{min}, d'un code $\mathcal{C}$ est la plus petite distance entre deux mots distincts du code :

$$d_{min} = \min_{\mathbf{x},\mathbf{y}\in\mathcal{C},\ \mathbf{x}\neq\mathbf{y}} d(\mathbf{x}, \mathbf{y}) \tag{1.8}$$

Intuitivement, on sent que plus la distance minimale est élevée plus la confusion entre deux mots de code est rendue difficile, et donc plus le code résiste aux erreurs. En contrepartie, la

taille du code est d'autant plus faible que les mots sont éloignés les uns des autres. Un compromis entre ces deux paramètres est donc à trouver.

Un code est dit **linéaire** si la somme de deux éléments quelconques du code est aussi un élément du code. Un code linéaire de longueur n est donc un **sous-espace vectoriel** de $\{0,1\}^n$. Notons $\mathcal{C}[n,k]$ un code de longueur n, de dimension k et linéaire.
Si le code est linéaire, la distance minimale du code est le poids minimal des mots non nuls du code.
La distance minimale d'un code en bloc linéaire satisfait la relation :

$$d_{min} \leq n - k + 1 \tag{1.9}$$

Par ailleurs les paramètres d'un code $\mathcal{C}[n,k]$ et de capacité de correction t (où t est le nombre de bits que peut corriger le code $\mathcal{C}$) ne sont pas quelconques ; ils satisfont l'égalité :

$$\sum_{i=0}^{t} C_n^i \leq 2^{n-k} \ ,$$

ainsi que la borne suivante :

Théorème 2 *Borne de Plotkin*
Soit $\mathcal{C}[n,k]$ un code linéaire binaire. Alors

$$d_{min} \leq \frac{n2^{k-1}}{2^k - 1}. \tag{1.10}$$

Tout code linéaire admet une représentation matricielle : la matrice génératrice du code est constituée d'une base du code, *i.e.* de k mots de code de longueur n et linéairement indépendants. Une matrice génératrice est donc de taille $(k \times n)$. Cette matrice est de rang k puisqu'elle engendre un espace vectoriel de dimension k. Notons d'ores et déjà que plusieurs matrices différentes peuvent définir un même code si les bases choisies sont différentes. Par ailleurs, en faisant des opérations de permutations des colonnes ou d'additions des lignes de la matrice dont on dispose, le code obtenu aura des propriétés identiques au code de départ, puisqu'il ne différera de ce dernier que par une simple permutation des composantes : deux tels codes sont dits **équivalents**.
La matrice génératrice est usuellement notée $\mathbf{G}$; puisque l'espace vectoriel qu'elle engendre est de dimension k, il est possible en faisant des opérations sur les lignes et les colonnes de cette matrice de faire apparaître la matrice identité et d'obtenir ainsi un code équivalent au code de départ, dit code **systématique** ; la matrice génératrice d'un code est dite **systématique**, si les k premières colonnes forment la matrice identité, $\mathbf{I_k}$.

$$\mathbf{G_{syst}} = (\mathbf{I_k} \ \ \mathbf{P}) \ .$$

Un code systématique fait donc correspondre à un mot d'information $\mathbf{u}$ de k bits, un mot de code $\mathbf{c}$ qui sera formé par la concaténation de $\mathbf{u}$ et des $n-k$ bits de redondance, appelés **bits de parité** : $\mathbf{c} = (\mathbf{u}| * **)$.
L'espace vectoriel de dimension k qu'est le code $\mathcal{C}$ peut également être défini à partir de son orthogonal, qui est, lui, de dimension $(n-k)$. La matrice génératrice du code orthogonal à $\mathcal{C}$ est appelée **matrice de parité de** $\mathcal{C}$. Elle est de dimension $\Big((n-k) \times n\Big)$.
La matrice de parité d'un code le caractérise de la façon suivante :

Proposition 8 *Soit $\mathcal{C}[n,k]$, $\mathbf{H}$ sa matrice de parité.*

$$\mathbf{c} \in \mathcal{C} \iff \mathbf{c}.\mathbf{H}^t = 0$$

La notion de distance entre deux mots de code est directement liée à la notion de décodage. Le décodage optimal des codes linéaires est le décodage à maximum de vraisemblance (maximum likelihood, ML) : il consiste à générer tous les mots du code et à choisir celui qui est à distance minimale du mot de code reçu. La distance entre deux mots de code sera évaluée en fonction de la nature du bruit et donc de celle du canal de transmission : par exemple, si le canal est AWGN (Additive White Gaussian Noise), la distance entre deux mots sera évaluée en utilisant la distance Euclidienne ; s'il s'agit d'un canal binaire symétrique, la distance sera calculée en utilisant la distance de Hamming. Outre la complexité associée au calcul de la distance, nous constatons que ce type de décodage requiert la génération de 2^k mots de code, sauf s'il existe des propriétés algébriques permettant de réduire cette complexité. La complexité du décodage optimal est en $\mathcal{O}(2^k)$.

Parmi les codes linéaires, il existe une famille particulière de codes dits **cycliques**.
Définition : Un code est dit cyclique si pour tout mot de code $\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}) \in \mathcal{C}$, le code $\mathcal{C}$ contient sa décalée cyclique $\mathbf{c}' = (c_{n-1}, c_0, \ldots, c_{n-2})$.

Représentation polynomiale des codes cycliques :

Lorsque l'on travaille avec des codes cycliques, on utilise souvent une représentation polynomiale des mots de code :
au mot

$$\mathbf{c} = (c_0, c_1, \ldots, c_{n-1}),$$

on associe le polynôme en x

$$c(x) = c_0 + c_1 x + c_2 x^2 + c_{n-1} x^{n-1}.$$

Comment obtenir la décalée d'un tel mot de code ? Si nous multiplions $c(x)$ par x, il nous faut transformer x^n en 1. Ceci s'obtient grâce à l'opération « modulo $x^n + 1$ » puisque $x^n = 1.(x^n + 1) + 1$.

L'ensemble des polynômes à coefficients dans $GF(2)$ et de degré inférieur ou égal à $n-1$ possède une structure d'espace vectoriel sur $GF(2)$. Cependant, lorsque les opérations sont effectuées modulo $x^n + 1$, il possède une structure d'anneau notée $GF(2)[x]/x^n + 1$.

Un tel code peut être engendré par un polynôme dit **polynôme générateur** : il s'agit du polynôme non nul de plus petit degré de $GF(2)[x]/x^n + 1$ usuellement noté $g(x)$, de degré noté $n-k$. Tout mot de code s'écrit comme le produit de $g(x)$ et d'un polynôme de degré inférieur ou égal à $k-1$.

Théorème 3 *Il existe un code de longueur n et de polynôme générateur $g(x)$ si et seulement si $g(x)$ divise $x^n + 1$.*

Par analogie avec la matrice de parité, on peut définir un **polynôme de parité** : soit $\mathcal{C}$ un code cyclique de polynôme générateur $g(x)$. Alors son dual a pour polynôme générateur le plus petit polynôme non nul tel que

$$g(x)h(x) = 0 \bmod (x^n + 1) \ .$$

Supposons que n s'écrive sous la forme $n = 2^m - 1$; le code de longueur n est alors dit **code primitif**.
D'après le théorème 3, $g(x)$ divise $x^{2^m-1} + 1$. L'ensemble des racines du polynôme $x^{2^m-1} + 1$ sont d'ordre $2^m - 1$ et sont donc des éléments de $GF(2^m)$.
Soit β_j un élément de $GF(2^m)$. Alors le plus petit polynôme à coefficients dans $GF(2)$ dont β_j soit une racine dans $GF(2^m)$ est dit **polynôme minimal** de β_j. On a donc l'égalité

$$x^{2^m-1} + 1 = \prod_{\beta_j \neq 0} (x - \beta_j) \ .$$

Théorème 4 *Si f est le polynôme minimal de $\beta \in GF(2^m)$ alors f est aussi le polynôme minimal de β^{2^i}.*

Deux éléments ayant même polynôme minimal sont dits **conjugués**.
Soit r le plus petit entier tel que $\beta^{2^r} = \beta$. Alors le polynôme minimal de ces éléments conjugués est $f_\beta = \prod_{i=1}^{r-1}(x - \beta^{2^i})$.
Intéressons-nous à une famille de codes particulières : les codes «BCH» (Bose-Chaudhuri-Hockenghem). Ces codes sont de longueur primitive et sont construits avec une capacité de correction t. Soit b un entier positif, β un élément primitif de $GF(2^n)$, *i.e.* d'ordre $2^n - 1$. Alors, un code BCH de longueur n et de capacité de correction t est défini comme le plus petit commun multiple des polynômes minimaux de $2t$ puissances successives de β.

$$g(x) = \text{PPCM}[f_{\beta^b}(x), f_{\beta^{b+1}}(x), \ldots, f_{\beta^{b+2t-1}}(x)].$$

En d'autres termes, g est le plus petit polynôme à coefficients dans $GF(2)$ ayant $\beta^b, \beta^{b+1}, \ldots, \beta^{b+2t-1}$ comme racines. Les codes BCH admettent des algorithmes de décodage qui utilisent la structure algébrique de ces codes (par exemple l'algorithme de Berlekamp-Massey, ou l'algorithme de Peterson-Gorenstein-Zierler). Nous verrons dans la suite (voir chapitre 3) un algorithme de décodage itératif, qui raisonne non par mot mais par bit.

Intéressons-nous maintenant à la distribution de poids des mots d'un code donné.
Définition : Le **polynôme énumérateur** de poids $A(x)$ d'un code est donné par :

$$A(x) = \sum_{i=0}^{n} a_i x^i, \tag{1.11}$$

où a_i est le nombre de mots de code de poids i.
Étant donné qu'un code détermine son dual, le polynôme énumérateur de poids d'un code détermine le polynôme énumérateur de poids du code dual, selon la relation suivante :

Proposition 9 *Identité de Mac-Williams.*
Soit A le polynôme énumérateur de poids d'un code, B le polynôme énumérateur de son code dual. Alors, on a l'égalité suivante :

$$B(x) = \frac{1}{2^k}(1 + x)^n A\Big(\frac{1 - x}{1 + x}\Big). \tag{1.12}$$

Un résultat asymptotique sur les codes :
Définition : Une famille de code est dite asymptotiquement bonne, si elle contient une suite infinie de codes tels que $R_i = \frac{k_i}{n_i}$ et $\frac{d_i}{n_i}$ ne tendent pas vers 0 lorsque i et n tendent vers l'infini.

$\mathbf{x}$	$f(\mathbf{x})$	$f_\chi(\mathbf{x})$
000	0	1
001	0	1
010	1	-1
011	0	1
100	0	1
101	1	-1
110	1	-1
111	1	-1

Tab. 1.1 – Table des valeurs d'une fonction à $n = 3$ entrées.

Théorème 5 *Borne de Gilbert-Varshamov*
Soit $0 \leq \delta < \frac{1}{2}$.
Il existe une suite infinie de codes linéaires binaires (n,k,d) de distance minimale normalisée $\frac{d}{n} \leq \delta$ *et de rendement* $R = \frac{k}{n}$ *telle que :*

$$R \geq 1 - H_2(\delta)$$

où $H_2(x) = -x\log_2(x) - (1-x)\log_2(1-x)$ est la fonction d'entropie binaire (nous reviendrons sur cette notion d'entropie dans le paragraphe 1.5).

1.4 Quelques notions sur les fonctions booléennes scalaires

Dans ce paragraphe, nous nous intéressons aux fonctions booléennes scalaires : nous présentons les définitions et propriétés qui nous paraissent utiles à la compréhension du système étudié. Le lecteur pourra néanmoins se reporter à [6, 7, 8, 12, 24, 27, 44, 75, 81, 82, 84, 85, 86, 87, 93, 94, 98, 100, 101] pour plus de détails.

Définitions et propriétés essentielles des fonctions booléennes :

Définition : Une **fonction booléenne scalaire** à n entrées est une fonction à valeurs dans $\{0, 1\}$.
Définition : Quelques définitions supplémentaires :

- on appelle **table des valeurs de la fonction**, la table qui recense pour chacun des 2^n vecteurs d'entrée possibles, les valeurs prises par la fonction ;
- une fonction booléenne est dite **équilibrée** si elle prend autant de fois la valeur 0 que la valeur 1 ;
- à toute fonction booléenne f, est associée une **fonction signe**, f_χ, qui s'exprime de la façon suivante : $f_\chi = (-1)^f$.

Exemple 3 *La fonction représentée dans la table 1.1 est une fonction booléenne scalaire à* $n = 3$ *entrées : le nombre de vecteurs d'entrée est* $2^n = 8$ *; cette fonction est équilibrée puisque quatre vecteurs d'entrée appartiennent à l'image réciproque de 0, quatre vecteurs appartiennent à l'image réciproque de 1.*

Définition : La **transformée de Fourier (ou de Walsh)** d'une fonction booléenne f, notée $\widehat{f}$, est telle que :

$$\forall\ \mathbf{u}\ \in \{0,1\}^n,\ \widehat{f}(\mathbf{u}) = \sum_{\mathbf{x}\in\{0,1\}^n} f(\mathbf{x})(-1)^{\mathbf{u}.\mathbf{x}}\ . \tag{1.13}$$

Cette définition est valable pour toute fonction à valeurs réelles et est un cas particulier de la transformée de Fourier sur un corps de caractéristique p (ici $p = 2$). Ainsi, la transformée de Fourier de la fonction signe est

$$\forall \mathbf{u},\ \widehat{f_\chi}(\mathbf{u}) = \sum_{\mathbf{x}\in\{0,1\}^n} (-1)^{f(\mathbf{x})+\mathbf{u}.\mathbf{x}}\ . \tag{1.14}$$

Cette transformation porte également le nom de « transformée de Walsh ». Le parallèle avec les matrices de Walsh-Hadamard peut notamment être établi de la façon suivante : rappelons que les matrices de Walsh-Hadamard sont construites par récurrence, avec

$$\mathbf{H_0} = 1$$

$$\mathbf{H_{2n}} = \begin{pmatrix} \mathbf{H_n} & \mathbf{H_n} \\ \mathbf{H_n} & \overline{\mathbf{H}}_\mathbf{n} \end{pmatrix}$$

avec $\overline{\mathbf{H}}_\mathbf{n} = -\mathbf{H_n}$.

Associons à chaque vecteur binaire de taille n, sa valeur décimale dans l'intervalle $[0; 2^{n-1}]$. Alors le vecteur $\Big(\widehat{f}(\mathbf{0}), \widehat{f}(\mathbf{1}), \ldots, \widehat{f}(\mathbf{2^n - 1})\Big)$ est obtenu en faisant le produit de H_n par le vecteur $\Big(f(\mathbf{0}), f(\mathbf{1}), \ldots, f(\mathbf{2^n - 1})\Big)$, appelé « **vecteur des valeurs** » de f.

Ces transformées doivent être évaluées en 2^n points, et chaque calcul fait intervenir une somme de 2^n termes : à première vue, le calcul du spectre d'une fonction semble requérir une complexité en calcul de l'ordre de 2^{2n} opérations. Il existe cependant un algorithme de calcul rapide de la transformée de Fourier qui est à relier directement à la construction récurrente des matrices de Walsh-Hadamard. Les différents calculs du spectre de la fonction font en effet intervenir des termes identiques (au signe près), et ce, de façon récursive.
L'algorithme de calcul rapide de la transformée de Fourier (Fast Fourier Transform) utilise cette propriété et permet ainsi un calcul ayant une complexité en

$$\mathcal{O}(n2^n). \tag{1.15}$$

La transformée de Fourier de la fonction signe $\widehat{f_\chi}$ compare les valeurs prises par f et celles prises par une forme linéaire du type $\mathbf{x} \longmapsto \mathbf{u}.\mathbf{x}$:

$$\widehat{f_\chi}(\mathbf{u}) = \sum_{\mathbf{x}} (-1)^{\mathbf{u}.\mathbf{x}+f(\mathbf{x})} \tag{1.16}$$

$$= \#\{\ \mathbf{x} \mid \mathbf{u}.\mathbf{x} = f(\mathbf{x})\ \} - \#\{\ \mathbf{x} \mid \mathbf{u}.\mathbf{x} \neq f(\mathbf{x})\ \}\ . \tag{1.17}$$

Définition : Soit $\mathbf{u} \in \{0,1\}^n$.
On appelle **fonction de Wash** $\chi_\mathbf{u}$ la fonction ;

$$\begin{array}{rcl} \chi_\mathbf{u} : \{0,1\}^n & \longrightarrow & \mathbb{R} \\ \mathbf{x} & \longmapsto & (-1)^{\mathbf{u}.\mathbf{x}}\ . \end{array}$$

Proposition 10 *Les fonctions $\{\chi_{\mathbf{u}}\}$ forment une base orthogonale de l'ensemble des fonctions à valeurs réelles, et sont de norme 2^n.*

Proposition 11 *Formule de Parseval :*
$\forall f : \{0,1\}^n \longrightarrow \{0,1\}$

$$\sum_u \widehat{f_\chi}^2(\mathbf{u}) = 2^{2n}. \tag{1.18}$$

Toute fonction booléenne s'exprime en fonction de sa transformée de Fourier par l'intermédiaire de la relation suivante :

Proposition 12 *Soit f une fonction booléenne, $\widehat{f}$ sa transformée de Fourier. Alors,*

$$\forall \mathbf{x} \in \{0,1\}^n,\ f(\mathbf{x}) = \frac{1}{2^n} \sum_{\mathbf{u}} \widehat{f}(\mathbf{u})(-1)^{\mathbf{u}.\mathbf{x}}. \tag{1.19}$$

La transformée de Fourier est donc une opération quasi involutive :

$$\widehat{(\widehat{f})} = 2^{2n} f\ . \tag{1.20}$$

La transformée de Walsh inverse correspond à l'expression de f dans la base des fonctions de Walsh : $f(\mathbf{x}) = \frac{1}{2^n} < \widehat{f}, \chi_{\mathbf{x}} >$, et $\widehat{f}$ est la projection de f sur les fonctions de Walsh : $\widehat{f}(\mathbf{u}) =< f, \chi_{\mathbf{u}} >$, où $< \mathbf{x}, \mathbf{y} >$ désigne le produit scalaire entre les vecteurs $\mathbf{x}$ et $\mathbf{y}$. La transformée d'une fonction f et la transformée de Fourier de sa fonction signe se déduisent l'une de l'autre par la relation qui suit.

Proposition 13 *Soit f une fonction booléenne.*
Alors $\forall \mathbf{u} \in \{0,1\}^n$

$$\widehat{f_\chi}(\mathbf{u}) = 2^n \mathbb{1}_{\{\mathbf{0_n}\}}(\mathbf{u}) - 2\widehat{f}(\mathbf{u}). \tag{1.21}$$

Définition : La **forme algébrique normale (F.A.N.)** d'une fonction booléenne f, est l'écriture (unique) de cette dernière comme une combinaison linéaire de monômes dont les coefficients sont dans $GF(2)$.

$$f(\mathbf{x}) = \sum_{\mathbf{u} \in \{0,1\}^n} a_{\mathbf{u}} \mathbf{x}^{\mathbf{u}} \tag{1.22}$$

où $\mathbf{x}^{\mathbf{u}} = \prod_{i=1}^n x_i^{u_i}$ avec $\mathbf{0^0} = 1$.

Le **degré** de la fonction f, également appelé ordre de non linéarité (noté $deg(f)$ ou $ord(f)$), est le degré de sa forme algébrique normale. La forme algébrique normale est calculée à partir de la table des valeurs de la fonction f de la façon suivante :

Proposition 14 *Étant donnée la table des valeurs de la fonction f, les coefficients a_u de la forme algébrique normale vérifient*

$$\forall \mathbf{u},\ a_{\mathbf{u}} = \sum_{\mathbf{v} \preceq \mathbf{u}} f(\mathbf{v}) \pmod 2 \tag{1.23}$$

*où la relation d'ordre $\preceq$, dite **de Lucas**, est définie par*

$$\forall\ \mathbf{x}, \mathbf{y} \in \{0,1\}^n,\quad \mathbf{x} \preceq \mathbf{y} \Longleftrightarrow \quad \forall\ 1 \leq i \leq n,\quad x_i \leq y_i\ . \tag{1.24}$$

Définition : Une fonction booléenne f est dite **linéaire** si

$$\exists \mathbf{u} \in \{0,1\}^n \text{ tel que } \forall \mathbf{x} \in \{0,1\}^n, \quad f(\mathbf{x}) = \mathbf{u}.\mathbf{x} \ .$$

Ces fonctions sont donc de degré 1, la fonction nulle exceptée.

Les fonctions dites **affines** consistent en l'ensemble des fonctions linéaires et leurs complémentaires.

Distance, corrélation, fonction « plateau » :

Définition : La **distance** entre deux vecteurs est le nombre de composantes sur lesquelles ces vecteurs diffèrent. Par extension, la distance entre deux fonctions booléennes est le nombre de vecteurs d'entrée sur lesquels ces fonctions diffèrent, *i.e.* la distance entre les deux vecteurs des valeurs des fonctions : soit f, g deux fonctions booléennes. Alors

$$d(f,g) = \#\{ \ \mathbf{x} \mid f(\mathbf{x}) \neq g(\mathbf{x}) \ \}. \tag{1.25}$$

D'après la remarque effectuée précédemment, la transformée de Fourier d'une fonction booléenne f et la distance de cette fonction à une fonction linéaire $\mathbf{u} \longmapsto \mathbf{u}.\mathbf{x}$ sont donc directement liées.

Proposition 15 *Soit f une fonction booléenne. Notons $\ell_{\mathbf{u}}$ la forme linéaire $\ell_{\mathbf{u}} : \mathbf{x} \longmapsto \mathbf{u}.\mathbf{x}$. Alors $\forall \mathbf{u}$,*

$$\begin{aligned} d(f, \ell_{\mathbf{u}}) &= 2^{n-1} - \frac{1}{2}\widehat{f_\chi}(\mathbf{u}) && (1.26)\\ &= 2^{n-1}\mathbb{1}_{\{\mathbf{u} \neq \mathbf{0_n}\}}(\mathbf{u}) + \widehat{f}(\mathbf{u}). && (1.27) \end{aligned}$$

Donc, plus $\widehat{f_\chi}(\mathbf{u})$ est faible en valeur absolue, plus la distance de f à la forme linéaire $\ell_{\mathbf{u}}$ (ou à sa fonction complémentaire) est élevée. Par ailleurs, d'après l'égalité de Parseval, la somme de ces quantités élevées au carré est constante : il apparaît donc légitime de se demander s'il existe des fonctions telles que leur distance à l'ensemble des fonctions affines soit la plus élevée possible.

Il existe une classe de fonctions qui sont à distance maximale des formes linéaires : il s'agit de la classe des fonctions courbes.

Définition : Soit f une fonction booléenne. Alors

$$\begin{aligned} f \text{ est } \mathbf{courbe} &\iff f \text{ est à distance maximale des formes linéaires} && (1.28)\\ &\iff \forall \mathbf{u}, \ \widehat{f_\chi}(\mathbf{u}) = 2^{\frac{n}{2}} \ . && (1.29) \end{aligned}$$

De telles fonctions n'existent que lorsque n est pair.

Proposition 16 *Soit f une fonction booléenne.*
Alors

$$f \text{ est courbe} \Longleftrightarrow \forall \mathbf{a} \neq \mathbf{0_n}, \ \sum_{\mathbf{x}} (-1)^{f(\mathbf{x}) + f(\mathbf{x}+\mathbf{a})} = 0 \ . \tag{1.30}$$

Proposition 17 *Une fonction booléenne courbe n'est pas équilibrée.*

Définition : La **corrélation** entre deux fonctions booléennes f et g est définie par

$$c(f,g) = \frac{1}{2^n}\Big(\#\{\ \mathbf{x} \mid f(\mathbf{x}) = g(\mathbf{x})\ \} - \#\{\ \mathbf{x} \mid f(\mathbf{x}) \neq g(\mathbf{x})\ \}\Big)\ . \tag{1.31}$$

En particulier, lorsque g est une fonction linéaire, $c(f,\ell_{\mathbf{u}}) = \frac{1}{2^n}\widehat{f_\chi}(\mathbf{u})$ ou, écrit en terme de probabilité,

$$P\Big[\ f(\mathbf{x}) = \ell_{\mathbf{u}}(\mathbf{x})\ \Big] = \frac{1}{2}\Big(1 + c(f,\ell_{\mathbf{u}})\Big) \tag{1.32}$$

$$= \frac{1}{2} + \frac{\widehat{f_\chi}(\mathbf{u})}{2^{n+1}}\ , \tag{1.33}$$

et

$$P\Big[\ f(\mathbf{x}) \neq \ell_{\mathbf{u}}(\mathbf{x})\ \Big] = \frac{1}{2}\Big(1 - c(f,\ell_{\mathbf{u}})\Big) \tag{1.34}$$

$$= \frac{1}{2} - \frac{\widehat{f_\chi}(\mathbf{u})}{2^{n+1}}\ . \tag{1.35}$$

Les égalités ci-dessus signifient que, si une fonction est telle que sa sortie est corrélée à certaines formes linéaires de son vecteur d'entrée, l'observation de la sortie de la fonction nous renseigne sur la valeur de cette forme linéaire.

Il existe cependant des fonctions qui, dans une certaine mesure, ne présentent pas de telles faiblesses : les fonctions immunes aux corrélations et les fonctions résilientes.

Définition : Le **poids de Hamming**, w_H, d'un vecteur est le nombre de composantes non nulles de ce dernier :

$$\forall \mathbf{x} \in \{0,1\}^n,\quad w_H = \#\{x_i \text{ tel que } x_i \neq 0\}\ .$$

Dans le cas particulier de $GF(2)^n$,

$$\forall \mathbf{x} \in \{0,1\}^n,\quad w_H = \sum_i x_i\ .$$

Définition : Soit f une fonction booléenne.
f est dite **immune aux corrélations à l'ordre** t si et seulement si f n'a aucune corrélation avec les formes linéaires de poids inférieur ou égal à t, *i.e.*

$$\forall \mathbf{u} \text{ tel que } 0 < w_H(\mathbf{u}) \leq t,\ \widehat{f_\chi}(\mathbf{u}) = 0\ .$$

f est dite **résiliente d'ordre** t ou t**-résiliente** si et seulement si f est immune aux corrélations d'ordre t et f est équilibrée *i.e.*

$$\forall \mathbf{u} \text{ tel que } 0 \leq w_H(\mathbf{u}) \leq t,\ \widehat{f_\chi}(\mathbf{u}) = 0\ .$$

Proposition 18 *Soit f une fonction booléenne à n entrées. Alors*

1. *Si f est immune aux corrélations à l'ordre t,*

$$deg(f) \leq n - t\ .$$

2. Si f est t-résiliente

$$deg(f) \leq n - t - 1 \ .$$

La notion de fonction « plateau » a été introduite par Y. Zheng et X.M. Zhang dans [100].
Définition : Une fonction est dite **plateau d'ordre r** si et seulement si le spectre de la fonction possède 2^r valeurs non nulles, toutes égales en valeur absolue.
Remarquons que d'après l'égalité de Parseval (cf. équation (1.18)),

$$f \text{ est } r\text{-plateau} \Longrightarrow \widehat{f_\chi}^2(\mathbf{u}) = 0 \text{ ou } \widehat{f_\chi}^2(\mathbf{u}) = 2^{2n-r} \ .$$

Définition : La **fonction d'auto-corrélation**, r_f, d'une fonction booléenne f est donnée par

$$\forall \mathbf{a} \in \{0,1\}^n, \ r_f(\mathbf{a}) = \sum_{\mathbf{x}} (-1)^{f(\mathbf{x})+f(\mathbf{x}+\mathbf{a})} \ . \tag{1.36}$$

Définition : Le **produit de convolution** de deux fonctions booléennes f et g est noté $f * g$, et s'exprime de la façon suivante :

$$\forall \mathbf{a} \ \in \ \{0,1\}^n, \ (f * g)(\mathbf{a}) = \sum_{\mathbf{x}} (-1)^{f(\mathbf{x})+g(\mathbf{x}+a)} \ . \tag{1.37}$$

À ce titre, la fonction d'auto-corrélation apparaît comme le produit de convolution de f avec elle-même.

Proposition 19 *Soit f, g deux fonctions booléennes.*
Alors

$$\widehat{f * g} = \widehat{f} \times \widehat{g} \ . \tag{1.38}$$

Corollaire 1 *Soit f une fonction booléenne.*

$$\begin{aligned} &\forall \mathbf{u} \in \{0,1\}^n \\ &\widehat{r_f}(\mathbf{u}) = (\widehat{f_\chi})^2(\mathbf{u}) \ . \end{aligned} \tag{1.39}$$

Définition : La fonction dite **translation de vecteur v** est la fonction

$$\begin{aligned} \tau_{\mathbf{v}} : \{0,1\}^n &\longrightarrow \{0,1\}^n \\ \mathbf{x} &\longmapsto \mathbf{x} + \mathbf{v} \ . \end{aligned} \tag{1.40}$$

1.5 Quelques notions de théorie de l'information

Nous renvoyons le lecteur aux ouvrages donnés en référence [19, 89] pour une description beaucoup plus complète que celle que nous donnons.

Soit X une variable aléatoire à valeurs discrètes sur un alphabet $\mathcal{A}$, prenant la valeur x avec la probabilité $P(x) = P(X = x)$.

Alors on définit la notion d'**entropie** de cette variable aléatoire de la façon suivante :
Définition :

$$H(X) = -\sum_{x\in\mathcal{A}} P(x)\log_2 P(x)\ . \tag{1.41}$$

L'expression de cette entropie en base 2 implique que l'on va parler d'entropie binaire. L'entropie binaire correspond à l'espérance de la variable aléatoire $\log_2 \frac{1}{P(X)}$. Elle peut également être perçue comme le nombre de questions binaires à poser pour déterminer la valeur de X : plus précisément, on peut montrer que ce nombre de questions est compris entre $H(X)$ et $H(X)+1$. L'entropie d'une variable aléatoire est donc une **mesure d'incertitude**, puisqu'elle donne la quantité d'information à se procurer pour pouvoir la connaître.

Supposons maintenant que nous disposions de deux variables aléatoires définies sur un même alphabet. La distance de Kullback nous permet d'évaluer l'erreur commise en représentant la première variable aléatoire par la seconde :
Définition : La **distance de Kullback** (également appelée entropie relative) entre deux lois de probabilité P et Q est donnée par l'expression suivante :

$$\begin{aligned} D(P||Q) &= \sum_{x\in\mathcal{A}} P(x)\log_2\frac{P(x)}{Q(x)} &(1.42)\\ &= E_P[\log_2\frac{P(x)}{Q(x)}]\ . &(1.43)\end{aligned}$$

Il peut être intéressant maintenant de mesurer l'écart entre la distribution conjointe de (X,Y) et celle obtenue en faisant le produit des deux distributions de probabilités (comme si les variables étaient stochastiquement indépendantes). Cet écart est mesuré par l'information mutuelle $I(X;Y)$ définie comme suit.

Définition : L'information mutuelle $I(X,Y)$ de deux variables aléatoire X, Y ayant pour distribution respective $P(x)$, $P(y)$ et comme distribution conjointe $P(x,y)$ est donnée par l'expression suivante :

$$\begin{aligned} I(X;Y) &= \sum_{x\in\mathcal{A}}\sum_{y\in\mathcal{B}} P(x,y)\log_2\frac{P(x,y)}{P(x)P(y)} &(1.44)\\ &= D\Big(P(x,y)||P(x)P(y)\Big)\ . &(1.45)\end{aligned}$$

Considérons maintenant un système de transmission constitué d'un canal. Soit X la variable aléatoire en entrée de ce canal ; la sortie Y du canal est conditionnée par la valeur de X et les probabilités du canal, et la distribution de probabilité de Y par celle de X et les probabilités du canal.

Un canal est dit **discret** s'il accepte en entrée un alphabet discret fini $\mathcal{A}$ et un alphabet discret fini $\mathcal{B}$ en sortie. Il est caractérisé par l'ensemble de ses probabilités de transition, $\Big(P(y|x)\Big)_{y\in\mathcal{B},x\in\mathcal{A}}$.
Citons d'ores et déjà l'exemple du **canal binaire symétrique**, en entrée duquel deux valeurs sont acceptées, 0 ou 1, tout comme en sortie. Il se caractérise par une unique probabilité de transition p, puisqu'il est symétrique $P(1 \mid 0) = P(0 \mid 1) = p$. Puisque $P(1 \mid 0) + P(0 \mid 0) = 1$, on en déduit également que $P(0 \mid 0) = P(1 \mid 1) = 1 - p$.
De plus, le canal est dit **sans mémoire** si la distribution de probabilité de la sortie du canal à l'instant t ne dépend que de l'entrée à l'instant t et pas des entrées ou sorties précédentes.

Considérons un instant t : le canal transmet de l'information ; mais la quantité d'information qu'il peut transmettre est-elle infinie ?
La définition qui suit laisse présager le contraire :

Définition : La **capacité de canal** d'un canal discret sans mémoire correspond au maximum d'information mutuelle sur les distributions de probabilité d'entrée :

$$C = \max_{P(x)} I(X;Y) \ . \tag{1.46}$$

La capacité de canal d'un canal binaire symétrique de probabilité de transition p est donnée par la formule suivante :

$$\begin{aligned} C &= 1 - H_2(p) && (1.47)\\ &= 1 - \Big(- p \log_2(p) - (1-p) \log_2(1-p) \Big) \ . && (1.48) \end{aligned}$$

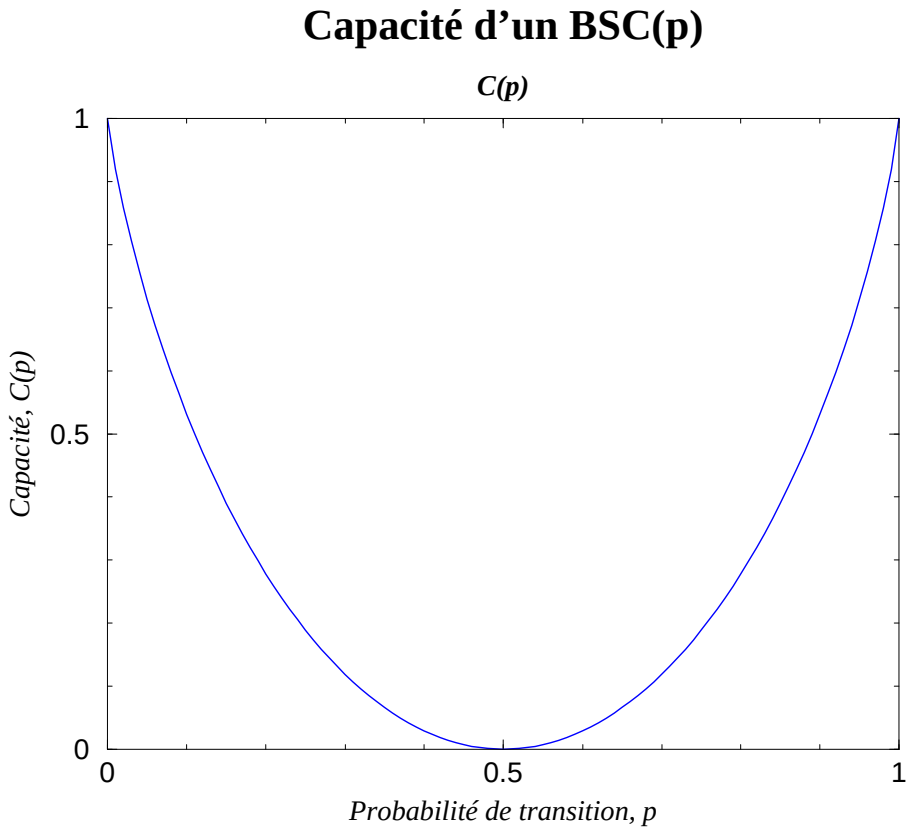


Fig. 1.4 – Capacité d'un canal binaire symétrique sans mémoire en fonction de sa probabilité de transition.

La figure 1.4 représente la capacité d'un canal binaire symétrique en fonction de sa probabilité de transition.
La capacité de canal constitue une limite très importante dans la théorie du codage, puisqu'elle constitue le seuil au-delà duquel le décodage d'un mot de code bruité ne pourra avoir lieu sans erreur.

Théorème 6 *Théorème de codage*
Soit un canal de transmission de capacité C. Tous les taux de codage inférieurs à C peuvent être atteints : quel que soit $R < C$, il existe une séquence de codes de paramètre n, $k = 2^{nR}$ dont la probabilité d'erreur après décodage tend vers 0 lorsque n tend vers l'infini.
Toute séquence de codes $(n, 2^{nR})$ dont la probabilité d'erreur tend vers 0 avec n vérifie $R \leq C$.

La preuve de ce théorème utilise des codes aléatoires et un décodage à maximum de vraisemblance. La notion d'aléa introduite dans cette preuve est à mettre en relation avec la construction des LDPC suggérée par Gallager (voir chapitre 3, paragraphe 3.1.1) et l'entrelaceur des turbo-codes (voir annexe E).
Remarquons par ailleurs que la probabilité d'erreur tend vers 0 lorsque **la longueur du code tend vers l'infini**, ce qui, en pratique, est irréalisable.

Nous verrons par la suite des algorithmes qui, utilisant non pas la fonction booléenne comme un canal binaire symétrique « simple » mais la structure plus complexe du registre filtré par la fonction, permettent de cryptanalyser des systèmes avec un nombre de bits parfois moins élevé que celui prévu par la théorie.

1.6 Lien entre la cryptanalyse du registre filtré et le décodage d'erreurs dans un mot de code bruité par un canal

Établissons maintenant le lien entre les notions décrites précédemment, le système du LFSR et les méthodes de cryptanalyse qui sont présentées dans ce document.
Rappelons que le système étudié est constitué d'un registre à décalage et d'une fonction booléenne (cf. figure 1.1). Le registre à décalage produit une séquence PN dont certaines phases sont prélevées afin de constituer le vecteur d'entrée de la fonction booléenne. La sortie de cette fonction constitue la séquence de clé qui est ajoutée bit à bit au message clair afin de produire le message chiffré.
Les méthodes de cryptanalyse présentées dans ce document sont des cryptanalyses à clair connu : nous supposons avoir intercepté une longueur N de bits de la séquence de bits de clé (z) (en ajoutant par exemple le message clair connu à la séquence de chiffré interceptée), et l'objectif est alors d'en déduire l'initialisation du registre à décalage.

La plupart des méthodes de cryptanalyse publiées dans la littérature considère non pas le système tel que nous venons de le décrire mais un système de codage :

1. Les K bits d'initialisation de la séquence PN sont vus comme les K bits d'information d'un code (N, K), code simplexe (linéaire) tronqué : tout mot de code du code simplexe est en effet de taille $2^K - 1$, qui sera très largement supérieure à la longueur N de séquence observée. Le dual du code simplexe est un code BCH (plus précisément un code de Hamming), il est cyclique. Ainsi, toute équation de parité satisfaite par les bits de la séquence PN sera également vérifiée par décalage.
   Par ailleurs, si l'on considère le code simplexe total, le nombre d'équation de parité disponibles et leur poids peuvent être évalués en utilisant l'identité de Mac-Williams et le polynôme énumérateur de poids $A(x)$ du code simplexe. D'après ce que l'on a vu, ce code contient un mot tout à zéro et $2^K - 1$ mots de code de poids $2^{K-1} + 1$. Ainsi,

$$A(x) = 1 + (2^K - 1)x^{2^{K-1}+1}.$$

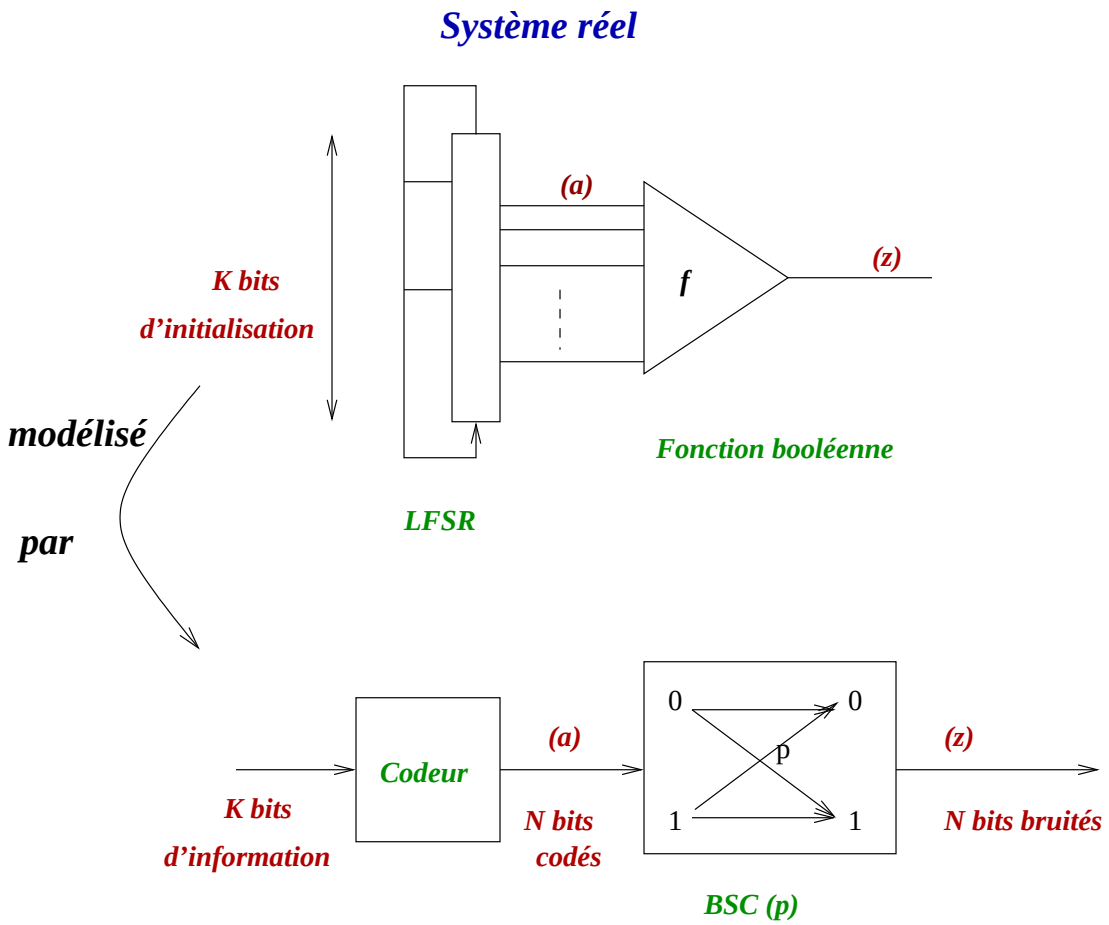


Fig. 1.5 – Modélisation du système de chiffrement LFSR - fonction booléenne en un code correcteur bruité par un BSC.

2. La fonction booléenne est supposée se comporter comme un canal binaire symétrique, dont la probabilité de transition est évaluée en fonction du spectre de la fonction. Nous revenons en détail ci-après sur l'évaluation de cette probabilité.
3. La séquence (z) en sortie de la fonction est alors perçue comme une version bruitée du mot de code émis, la probabilité d'erreur sur chacun des bits de cette séquence étant la probabilité de transition du canal.

La modélisation du système cryptographique en un mot de code bruité par un canal est illustrée sur la figure 1.5.

Comme nous l'avons vu dans le paragraphe 1.2, le polynôme de rétroaction du registre sera choisi primitif et de mémoire importante, empêchant ainsi toute attaque exhaustive sur son état initial.

Il faudra également que la séquence résultant du filtrage de la séquence PN ait de bonnes propriétés. La fonction est donc au coeur de la sécurité du système : si la perturbation qu'elle

engendre sur la séquence (a) est faible, le cryptanalyste pourra aisément identifier la clé secrète du système et regénérer sans difficulté toute la séquence de bits de clé.

Comme nous l'avons déjà souligné dans le paragraphe 1.2, toute séquence déduite d'une combinaison linéaire des bits de la séquence (a) est en fait une décalée de cette dernière séquence. En particulier, et nous utiliserons et développerons cette idée plus avant dans la suite de ce document, le décodage de la séquence (a) peut se faire par l'intermédiaire d'une séquence déduite d'une combinaison linéaire de cette séquence. L'intérêt est que, si nous considérons le système de la figure 1.1, et en utilisant les propriétés de la fonction booléenne décrites au paragraphe 1.4, la fonction booléenne peut avoir des corrélations assez élevées avec certaines combinaisons linéaires de son vecteur d'entrée, ce qui se traduit, en terme de codage, par une probabilité d'erreur plus faible sur la séquence reçue.
Quels sont alors les **critères** qui dicteront le **choix d'une fonction booléenne** ?

D'après ce qui précède, le **spectre** de la fonction ne doit pas présenter de pics. Par ailleurs, elle devrait être **équilibrée**, afin que l'observation brutale de la séquence (z) ne nous renseigne pas sur la séquence (a). Le modèle du canal binaire symétrique, qui correspond à une probabilité de transition symétrique est par ailleurs faux lorsque la fonction booléenne n'est pas équilibrée. En effet, soit $\mathbf{u}$ le vecteur correspondant à la forme linéaire que l'on choisit de décoder. Détaillons le calcul de la probabilité de transition du canal binaire symétrique équivalent.
$\forall\ z_t,\ \exists\ \mathbf{X}(t)$ tel que $z_t = f\Big(\mathbf{X}(t)\Big)$. Nous désignerons $\mathbf{X}(t)$ par $\mathbf{X}$ pour simplifier les notations. En appliquant la définition de la probabilité conditionnelle, on obtient

$$P[f(\mathbf{X}) = 1 \mid \mathbf{u}.\mathbf{X} = 0] = \frac{P[f(\mathbf{X}) = 1,\ \mathbf{u}.\mathbf{X} = 0]}{P[\mathbf{u}.\mathbf{X} = 0]} \tag{1.49}$$

$$= 2 \times \frac{1}{2^n} \sum_{\mathbf{X}} f(\mathbf{X}) \frac{1 + (-1)^{\mathbf{u}.\mathbf{X}}}{2} \tag{1.50}$$

et donc,

$$P[f(\mathbf{X}) = 1 \mid \mathbf{u}.\mathbf{X} = 0] = \frac{1}{2^n}[\widehat{f}(\mathbf{0_n}) + \widehat{f}(\mathbf{u})] \tag{1.51}$$

$$P[f(\mathbf{X}) = 1 \mid \mathbf{u}.\mathbf{X} = 1] = \frac{1}{2^n}[2^n - \widehat{f}(\mathbf{0_n}) + \widehat{f}(\mathbf{u})] \tag{1.52}$$

où

$$\widehat{f}(\mathbf{0_n}) = \sum_{\mathbf{x}} f(\mathbf{x}) = \#\{\mathbf{x} \text{ tel que } f(\mathbf{x}) = 1\}$$

et donc si la fonction n'est pas équilibrée, le canal équivalent à la fonction n'est plus symétrique.

Considérons les deux remarques précédentes. Une fonction idéale du point de vue des corrélations serait une fonction dont toutes les corrélations sont faibles afin que toutes les formes linéaires soient éloignées de la sortie de la fonction. Comme nous l'avons vu au paragraphe 1.4, ces fonctions sont les fonctions courbes qui n'existent que pour une dimension n paire. Cependant, le fait même que leur spectre soit plat implique que ces fonctions ne sont pas équilibrées. D'après l'égalité de Parseval, si f est courbe alors $\widehat{f_\chi}(\mathbf{u}) = 2^{\frac{n}{2}}$. En particulier, $\widehat{f_\chi}(\mathbf{0_n}) = \sum_{\mathbf{x}} (-1)^{f(\mathbf{x})} = 2^{\frac{n}{2}}$ et la fonction n'est donc pas équilibrée.

Enfin, d'après l'étude de complexité linéaire qui a été faite au paragraphe 1.2, il sera souhaitable que la fonction ait un **ordre de non-linéarité élevé** afin d'assurer une complexité linéaire élevée à la suite qu'elle produit.

L'intérêt du **caractère résilient** (cf. paragraphe 1.4) d'une fonction booléenne dans le cas d'un unique registre filtré n'est, à première vue, pas évident ; en revanche, dans le cas de générateurs par combinaison de registres (dont la structure est détaillée plus avant au chapitre 2, cf. figure 2.2), leur utilisation empêche la mise en oeuvre d'attaque de type « divide and conquer ». Nous montrerons cependant dans ce document, notamment aux chapitres 3 et 5, que ces fonctions présentent une meilleure résistance face aux attaques que nous avons développées.

Il nous faut toutefois modérer cette affirmation.

En effet, A. Canteaut (cf. [13]) nous a fait remarquer que nous n'étions pas obligés de considérer le système tel quel : nous pouvons considérer que la séquence (z) résulte non plus du filtrage de la séquence (a) par la fonction f mais du filtrage d'une séquence (a^ℓ) par une fonction h^ℓ où

- (a^ℓ) est obtenue par combinaison linéaire des bits de la séquence (a) ;
- h^ℓ est calculée à partir de f et de l'inverse de la forme linéaire ℓ pour produire la même séquence (z) en sortie.

Cela est possible puisque toute combinaison linéaire de la séquence (a) satisfait la récurrence associée au polynôme de rétroaction, et que, en vertu du théorème 1 donné au chapitre 1 (cf. page 4) et du fait que le polynôme de rétroaction est primitif donc irréductible, toute combinaison linéaire est inversible.
Donc si $\exists \ell$ tel que $(a^\ell) = \ell.(a)$, alors $\exists \ell^{-1}$ tel que

$$\begin{aligned} \ell(x).\ell^{-1}(x) &= 1 \bmod g(x) \\ &\text{et} \\ (a) &= \ell^{-1}(a^\ell)\ . \end{aligned}$$

La fonction h^ℓ ne sera pas nécessairement résiliente et les algorithmes de cryptanalyse exploitant les corrélations d'ordre 1 seront applicables et permettront d'identifier (a^ℓ).

Notons cependant que, en toute généralité, l'inverse du polynôme $\ell(x)$ peut être de degré et de poids élevé, ce qui se répercute sur la fonction h^ℓ en un nombre élevé de variables d'entrées (majoré par K). Le spectre de la fonction h^ℓ pourra toutefois être évalué aisément à partir des valeurs non nulles de la fonction f de départ.
Ainsi, si la fonction f est résiliente et si l'on arrive à exhiber une forme linéaire ℓ telle que la fonction h^ℓ correspondante a peu de variables et n'est plus résiliente, le caractère résilient de la fonction de départ n'est plus pertinent.

Ceci amène à définir un nouveau critère de conception : la fonction f, les espacements (λ_i) entre ses entrées et le polynôme de rétroaction g devront être tels qu'il n'existe aucune forme linéaire ℓ de la séquence d'entrée telle que la la fonction h^ℓ qui lui correspond et permet de produire la séquence (z) vérifie **conjointement** les conditions suivantes :

- h^ℓ est telle qu'elle entre dans les conditions d'application de nos algorithmes : par exemple, on peut exiger de h^ℓ qu'elle ait un nombre d'entrées restreint, *i.e.* l'inverse de $\ell \bmod g(x)$ est de poids faible, et/ou les espacements λ_i seront tels que le nombre d'entrées de h^ℓ restera « praticable » pour nos algorithmes (même si son spectre est déduit de celui de f, il peut faire intervenir un nombre prohibitif de valeurs) ;
- la fonction h^ℓ n'est plus résiliente.

Si ces deux conditions ne peuvent être conjointement satisfaites, l'utilisation de fonctions résilientes demeure pertinente.

Cette remarque nous servira tout au long de ce document, puisque nous avons très souvent souligné l'effet des fonctions résilientes sur nos algorithmes, d'autant plus qu'elle peut se généraliser à des propriétés différentes de la résilience : de façon plus générale, s'il existe une forme linéaire ℓ de la séquence d'entrée telle que la fonction h^ℓ qui lui correspond a un nombre d'entrées relativement faible et que ses propriétés sont meilleures que celles de la fonction d'origine f, il sera plus intéressant de décoder (a^ℓ).

Tous les résultats que nous présentons dans ce document sont obtenus sur f directement, sans avoir préalablement cherché de système équivalent plus favorable à la cryptanalyse, *i.e.* **en supposant qu'il n'existe aucune fonction déduite de f qui pourrait être plus facilement cryptanalysée**. Mais le lecteur doit garder à l'esprit que le fonctionnement de nos algorithmes pourrait être éventuellement amélioré, en recherchant une forme linéaire ℓ induisant sur la fonction h^ℓ des propriétés favorables au fonctionnement de l'algorithme considéré.

Essayons enfin de voir les limites imposées par la théorie des codes correcteurs introduite plus haut. La théorie développée au paragraphe 1.5 nous permet par exemple d'évaluer la longueur de séquence minimale en deçà de laquelle le cryptanalyste ne pourra jamais espérer décoder sans erreur les K premiers bits de la séquence : supposons, par exemple que $K = 100$, et que la probabilité de transition équivalente du système soit de $p = 0.30$. Alors d'après la figure 1.4, la capacité du canal est $C \approx 1 - 0.88 = 0.12$. Le code sur lequel nous travaillons est de rendement $R = \frac{K}{N}$ où N est la longueur d'observation. D'après le théorème de codage, on doit avoir $R \leq C$ ce qui implique $\frac{K}{N} \leq C(p) \Longrightarrow N \geq \frac{K}{C(p)}$
Dans notre exemple, il faudra donc que N vérifie

$$N \geq \frac{100}{0.12} \approx 833 \ ,$$

i.e. qu'il soit environ huit fois plus élevé que le nombre de bits de clé.
Si la probabilité de transition est de $p = 0.45$, alors N devra vérifier :

$$N \geq \frac{100}{7.10^{-3}} \approx 14285 \ ,$$

i.e. N devra être environ cent cinquante fois supérieur au nombre de bits de clé.

1.7 Notations utilisées dans ce document

Nous exposons succinctement les principales notations qui seront utilisées dans la suite de ce document. Nous serons ponctuellement amenés à introduire de nouvelles quantités mais les notations communes à tous les chapitres de ce document sont les suivantes :

- le **degré** du **polynôme de rétroaction** $g(x) = \sum_i g_i x^i$ est K ;
- la **séquence PN** produite par le générateur d'aléa est notée (a). En particulier, **l'état initial** du registre à décalage est $\mathbf{a_0} = (a_0, \ldots, a_{K-1})$.
  Le générateur étant à rétroaction linéaire, les bits de la séquence (a) vérifient la récurrence linéaire :
  $$\forall n \geq K, a_t = \sum_{i=1}^{K} g_i a_{t-i} \ ;$$
- le nombre d'entrées de la fonction booléenne sera noté n ;

– la séquence déduite de l'application d'une forme linéaire $\ell_{\mathbf{u}}$ à la séquence initiale (a) sera notée $(a^{\ell_{\mathbf{u}}})$: si $\ell_{\mathbf{u}} = \sum_i \ell_i x^i$ alors

$$a_t^{\ell_{\mathbf{u}}} = \sum_i \ell_i a_{t+i}.$$

$\mathbf{u}$ est le vecteur d'entrée de la fonction (donc à n composantes) qui correspond à cette forme linéaire ;

– la séquence observée en sortie de la fonction, séquence de bits de clé, de longueur N, est notée (z) ;

– la **matrice de transition** $(K \times K)$ associée au registre à décalage est notée

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & \dots & 0 & g_K \\ 1 & 0 & \dots & 0 & g_{K-1} \\ 0 & 1 & & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & g_1 \end{pmatrix}.$$

Ainsi, à chaque instant t, l'état du générateur PN, s'exprime de la façon suivante :

$$\mathbf{a_t} = \mathbf{a_{t-1}}\ \mathbf{G}\ = \mathbf{a_0}\ \mathbf{G}^t$$

où $\mathbf{a_t} = (a_t, a_{t+1}, \dots, a_{t+K-1})$.
On note $\mathbf{G_i}^p$ la i-ème colonne de la matrice $\mathbf{G}^p$.
La **matrice génératrice globale** s'écrit :

$$\mathbf{C_{glob}} = \left(\mathbf{I_K}\ \ \mathbf{G_K}^1 \dots \mathbf{G_K}^m \dots \mathbf{G_K}^{N-K}\right)$$

qui, en posant $\mathbf{G_K}^m = (c_1^m, \dots, c_K^m)^\dagger$ s'écrit :

$$\mathbf{C_{glob}} = \begin{pmatrix} 1 & 0 & \dots & 0 & c_1^1 & \dots & c_1^m & \dots & c_1^{N-K} \\ 0 & 1 & \dots & 0 & c_2^1 & \dots & c_2^m & \dots & c_2^{N-K} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & \dots & 1 & c_K^1 & \dots & c_K^m & \dots & c_K^{N-K} \end{pmatrix}$$

On a

$$(a_0, \dots, a_{N-1}) = (a_0, a_1, \dots, a_{K-1})\mathbf{C_{glob}}\ ;$$

– la **fonction booléenne** de filtrage est notée f ;

– à toute fonction booléenne (réelle dans le cas général), f, on peut associer sa **transformée de Fourier** :

$$\forall u \in \{0,1\}^n, \widehat{f}(u) = \sum_{x \in \{0,1\}^n} f(x)(-1)^{u.x}$$

où $u.x = \sum_i u_i x_i$ désigne le produit scalaire usuel ;

– lorsque le modèle du canal binaire symétrique est utilisé, le **vecteur d'erreurs** qui est associé au passage de la séquence (a) dans le canal est appelé $(e) = (e_0, \dots, e_{N-1})$. On a alors la relation $\forall i,\ z_i = a_i + e_i$;

– la **probabilité de transition** du canal binaire symétrique est notée $p = 1/2 + \epsilon$. On a donc $\forall i,\ P(e_i = 1) = p$;

– les **espacements** entre les différentes entrées de la fonction, relativement au générateur de pseudo-aléa sont notés $(\lambda_i)_{i=0}^{n-2}$;

– la **mémoire globale du système** est notée Λ ; elle correspond au nombre minimal de décalages (« clock ») de même amplitude à effectuer pour qu'un bit de la séquence d'entrée (a), initialement à la première entrée de la fonction, sorte de la dernière entrée.
Dans le cas d'un registre à décalage,

$$\Lambda = 1 + \frac{\sum_{i=0}^{n-2} \lambda_i}{\mathrm{pgcd}\,(\lambda_0, \lambda_1, \ldots, \lambda_{n-2})} \ . \tag{1.53}$$

On voit alors que, si les espacements sont réguliers, $\Lambda = n$, *i.e* il suffit de décimer la séquence de sortie pour se ramener au cas où les espacements sont tous égaux à 1.
Le système constitué d'une fonction à n entrées, prélevées dans le générateur de telle sorte que la mémoire est Λ, est équivalent au système constitué d'une fonction f' à Λ entrées, obtenue en dégénérant f sur $n-\Lambda$ positions. Pour simplifier les notations, nous assimilerons f' à f, *i.e* la fonction f considérée aura Λ entrées dont $\Lambda - n$ sans influence sur la sortie. Λ est donc le nombre minimal d'entrées d'une fonction f' déduite de f, qui a des entrées régulièrement espacées et qui produit la suite (z) en filtrant la suite (a) ;
– enfin, les vecteurs ou matrices seront notés en gras afin de mieux les différencier des quantités scalaires.

Dans toute la suite de ce document, nous supposons que le cryptanalyste connaît parfaitement la structure du registre à décalage, de la fonction booléenne ainsi que l'espacement existant entre chacune de ses entrées. Parmi les paramètres présentés ci-dessus, seule la séquence (a) (ainsi que les séquences qui en dérivent et les vecteurs d'état du registre qui lui sont associés) est inconnue.

Le document sera organisé de la façon suivante : le chapitre 2 présente quelques-unes des principales attaques publiées dans la littérature en distinguant deux types d'attaques : les attaques que nous avons appelées déterministes et les attaques probabilistes.
Dans le chapitre 3, nous nous sommes intéressés à l'application de l'algorithme de Gallager au système du registre filtré, et avons souhaité évaluer les différences de performances de ces derniers selon les paramètres du système et le type de canal utilisé. Cet algorithme (et les versions approchées qui en dérivent) utilisent des contraintes linéaires qui agissent sur les bits : nous avons étendu son principe à des contraintes très différentes.
Le chapitre 4 présente des algorithmes de cryptanalyse qui utilisent un treillis. L'application de ces derniers semble, à première vue, restreinte à des structures de registre filtré très particulières, dans lesquelles les espacements entre les entrées de la fonction sont réguliers. Nous avons étendu ces algorithmes à des espacements quelconques.

Enfin, dans le dernier chapitre, nous présentons des algorithmes de cryptanalyse que nous avons appelés « SOJA » (Soft Output Joint Algorithm) qui utilisent conjointement des contraintes linéaires sur les vecteurs d'entrée et la connaissance de la fonction booléenne pour inférer les bits de clé. Une version itérative du SOJA est par ailleurs étudiée. Nous présentons en dernier lieu un algorithme inspiré à la fois du SOJA et de la cryptanalyse linéaire et étudions ses performances.

Chapitre 2

État de l'art sur la cryptanalyse des registres filtrés

Un regain d'intérêt pour la cryptanalyse des registres filtrés est apparu en 1988 lors de la parution de l'article de W. Meier et O. Staffelbach [66], qui, les premiers, ont modélisé la cryptanalyse du registre filtré comme un problème de décodage d'un mot de code bruité et ont adapté un algorithme de décodage de canal à un tel système. De nombreux articles fondés ou non sur ce principe de « codes correcteurs », ont depuis lors vu le jour : [1, 10, 11, 16, 17, 25, 26, 34, 39, 40, 41, 42, 47, 48, 49, 53, 50, 51, 67, 68, 69, 70, 71, 72, 73, 74, 76].
L'objet de ce chapitre est de présenter quelques-unes des nombreuses méthodes de cryptanalyse d'un unique registre filtré non linéairement par une fonction booléenne (cf. figure 2.1, reproduction de la figure 1.1 du chapitre 1) qui sont parues dans la littérature ces dernières années.

Nous allons distinguer deux grands types d'attaques : les attaques déterministes et les attaques probabilistes. Une attaque sera qualifiée de « **déterministe** » si, exploitant la structure du registre filtré, le caractère déterministe de la fonction booléenne et la mémoire induite par l'unique registre à décalage notamment, elle permet de récupérer la clé du système avec une probabilité d'erreur (généralement) nulle. À l'inverse, les attaques dites « **probabilistes** » sont issues des techniques de codage correcteur d'erreurs : la modélisation de la fonction en un canal introduisant des erreurs sur la séquence de pseudo-aléa permet de se ramener à un problème de décodage de canal. Ce type d'attaques est en fait dédié aux générateurs par combinaison de registres, dont un exemple est donné sur la figure 2.2 : un générateur à combinaison est constitué de plusieurs registres indépendants filtrés par une fonction. Dès lors qu'une corrélation non nulle entre une des entrées de la fonction et sa sortie peut être mise en évidence, ce type d'attaques est applicable. Le cas d'un unique registre filtré par une fonction peut être vu comme un cas particulier du générateur par combinaison de registres ; on peut s'y ramener en considérant n registres identiques en entrée de la fonction, initialisés spécifiquement. L'initialisation de chacun d'entre eux est en effet choisie de telle sorte que les vecteurs en entrée de la fonction soient identiques à ceux obtenus dans le cas d'un registre unique : cela est possible car chaque composante du vecteur d'entrée de la fonction est une décalée de la séquence PN produite par le générateur, et peut donc être générée par un registre dont l'initialisation est une décalée de l'état initial du registre originel.

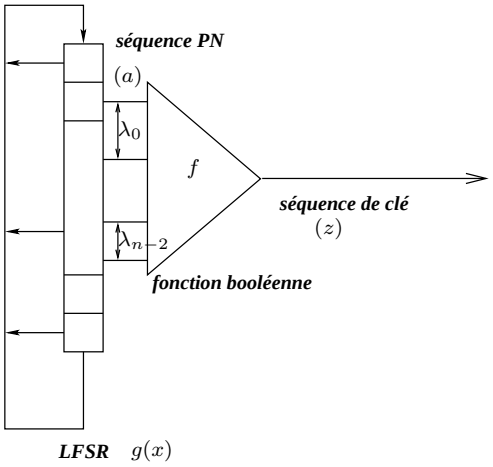


FIG. 2.1 – Un registre unique filtré par une fonction booléenne.

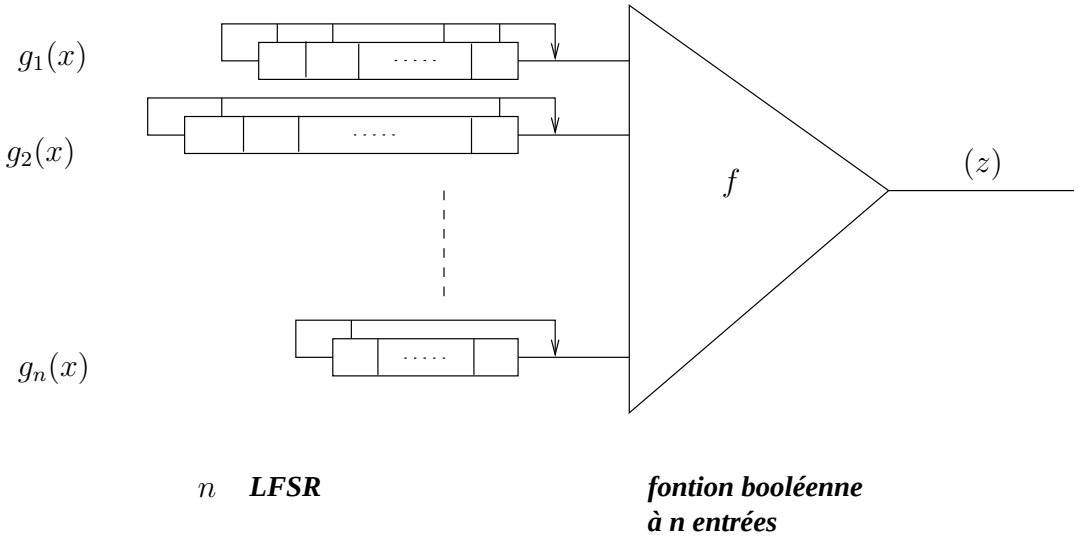


FIG. 2.2 – Un générateur par combinaison de registres.

2.1 Quelques attaques déterministes

Nous présentons ci-après trois attaques déterministes, dédiées par nature au système du registre filtré. Les attaques déterministes permettent (généralement) de retrouver la clé sans probabilité d'erreur. Dans ce paragraphe, nous n'aborderons pas les attaques par résolution polynomiale : tout bit en sortie de la fonction booléenne est en effet un polynôme des K variables de mémoire interne du registre, qui se calcule à partir de la forme algébrique normale de la fonction et en utilisant le décalage opéré par le générateur. Une attaque algébrique consiste à résoudre un tel système polynomial dont la complexité devient, à première vue, rapidement prohibitive lorsque le degré de la fonction et la mémoire du registre croissent. Toutefois, certaines techniques récentes de résolution de systèmes polynomiaux, parues dans la littérature, s'avèrent très efficaces pour la cryptanalyse de nos systèmes. Nous renvoyons à ce propos le lecteur aux articles [20, 21, 2, 23].

Comme nous l'avons précisé auparavant, les algorithmes présentés dans cette partie utilisent la connaissance de la fonction booléenne, ses propriétés, ainsi que la mémoire induite par le décalage qu'opère le LFSR. Leur complexité est donc essentiellement liée au nombre d'entrées de la fonction et au choix des espacements (λ_i), comme nous allons le constater ci-après.

2.1.1 Recherche de corrélations par bloc, R. J. Anderson [1]

• ***Description de l'algorithme :***
On rappelle que Λ est la mémoire du système induite par la fonction booléenne et les connexions de ses entrées dans le générateur d'aléa. On définit la « fonction augmentée » :

$$\begin{array}{rccl} \mathcal{F} \ : & \{0,1\}^{2\Lambda-1} & \longrightarrow & \{0,1\}^{\Lambda} \\ & (x_1,\ldots,x_{2\Lambda-1}) & \longmapsto & \Big(f(x_1,\ldots,x_{\Lambda}), f(x_2,\ldots,x_{\Lambda+1}),\ldots,f(x_{\Lambda},\ldots,x_{2\Lambda-1})\Big)\ . \end{array}$$

Cette fonction tient compte de l'influence entière du bit x_Λ et de ses voisins sur les bits de sortie. La première phase de l'algorithme est une phase de pré-calculs, qui consiste à lister l'ensemble des antécédents de chaque vecteur de l'image de $\mathcal{F}$ afin de construire une table $\mathcal{T}$ telle que :

$$\begin{array}{l} \forall (y_1,\ldots,y_\Lambda) \in \{0,1\}^{\Lambda}, \\ \mathcal{T}(y_1,\ldots,y_\Lambda) = \Big\{ \ (x_1,\ldots,x_{2\Lambda-1}) \ \mid \ \mathcal{F}(x_1,\ldots,x_{2\Lambda-1}) = (y_1,\ldots,y_\Lambda) \Big\}\ . \end{array}$$

À l'aide de cette table, on cherche les vecteurs $(y_1,\ldots,y_\Lambda)$ dont l'image réciproque est constituée de vecteurs ayant une composante identique, *i.e.*, on cherche $(y_1,\ldots,y_\Lambda)$ tel que :

$$\begin{array}{c} \exists\ i\ \in [1; 2\Lambda-1],\ \exists\ b \in \{0,1\} \text{ tel que} \\ \forall (x_1,\ldots,x_{2\Lambda-1}) \in \mathcal{T}(y_1,\ldots,y_\Lambda),\quad x_i = b\ . \end{array}$$

Les vecteurs $(y_1,\ldots,y_\Lambda)$ vérifiant une telle propriété sont particulièrement intéressants puisque l'observation du motif $(y_1,\ldots,y_\Lambda)$ en sortie de la fonction nous renseigne sur la valeur d'un bit de la séquence d'entrée. L'attaque consiste à chercher de tels motifs dans la séquence reçue ; comme nous l'avons souligné au paragraphe 1.6 du chapitre 1, une fois que l'on a pu identifier K bits linéairement indépendants, il ne reste qu'à inverser un système linéaire $(K \times K)$ pour retrouver l'état initial.

Remarque : La recherche de bits constants sur les vecteurs de l'image réciproque d'un motif $(y_1,\ldots,y_\Lambda)$ peut être étendue à la recherche de formes linéaires constantes sur ces vecteurs.

Algorithme de recherche de corrélations par bloc

1. Chercher les motifs de taille Λ tels que leur image réciproque ait un bit ou une forme linéaire de bits constant(e). S'il n'en existe pas, l'algorithme se termine sans avoir trouvé la clé.
2. Scanner la séquence (z) à la recherche des motifs d'intérêt précédemment déterminés, jusqu'à avoir obtenu K formes linéaires des bits d'entrée indépendantes.
3. Inverser le système linéaire pour obtenir la clé secrète.

Tab. 2.1 – Algorithme de recherche de corrélations par bloc de R.J. Anderson, FSE 94 [1].

L'algorithme complet est alors décrit dans le cadre 2.1.

• ***Complexité :***
La complexité de l'algorithme réside essentiellement dans la phase de construction de la table, qui revient à énumérer tous les vecteurs d'entrée de $\mathcal{F}$. La complexité est donc en $\mathcal{O}(2^{2\Lambda-1})$.

2.1.2 Attaque par inversion dans le cas d'une fonction linéaire par rapport à sa première ou sa dernière variable, J. Golic [41]

• ***Description de l'algorithme :***
Cette attaque ne s'applique que lorsque la fonction de filtrage f est linéaire par rapport à sa première ou sa dernière variable. Le principe de l'attaque est d'utiliser la linéarité de f pour inverser le système et reconstituer la séquence de départ.
Supposons par exemple que f soit linéaire par rapport à sa première variable, *i.e.*

$$\exists h : \{0,1\}^{\Lambda-1} \longrightarrow \{0,1\} \text{ telle que } \forall (x_1, \ldots, x_\Lambda) \in \{0,1\}^\Lambda$$

$$f(x_1, \ldots, x_\Lambda) = x_1 + h(x_2, \ldots, x_\Lambda) \ .$$

Cette relation se traduit alors sur les séquences d'entrée et de sortie de la façon suivante :

$$\forall t, \quad z_t = a_t + h(a_{t-\lambda_0}, a_{t-(\lambda_0+\lambda_1)}, \ldots, a_{t-(\Lambda-1)}) \ . \tag{2.1}$$

L'algorithme est donné dans le cadre 2.2.

• ***Complexité :***
La complexité est essentiellement liée au nombre de tirages qu'il faut effectuer. L'espérance du nombre de tirages est $\mathcal{O}(2^{\Lambda-2})$.

Attaque déterministe sur une fonction linéaire par rapport à sa première ou sa dernière variable

1. Tirer aléatoirement $\Lambda-1$ nouveaux bits $a_0, \ldots, a_{\Lambda-2}$, et en utilisant la séquence reçue (z) et la relation (2.1), en déduire $a_0, \ldots, a_{K-1}$.
2. En utilisant la récurrence linéaire induite par le générateur de pseudo-aléa et les K premiers bits $a_0, \ldots, a_{K-1}$ calculés à l'étape 1, générer $(a_i)_{i=0}^{N-1}$.
3. Filtrer cette séquence, et comparer le résultat à la séquence (z) réellement obtenue. Si les deux séquences sont identiques, la séquence initiale a été trouvée. Sinon retourner en 1.

Tab. 2.2 – Attaque déterministe sur une fonction linéaire par rapport à sa première ou sa dernière variable, J. Golic, FSE 96 [41].

2.1.3 Cryptanalyse via une recherche arborescente, J. Golic *et al.* [42]

• ***Description de l'algorithme :***
L'idée de cette attaque est de construire un arbre, fonction de la suite observée (z) et de la table de la fonction. Chaque noeud dans l'arbre a deux fils : ce noeud décalé vers la droite, précédé d'un 0 ou d'un 1. Les noeuds sont de taille $\Lambda-1$: les fils de $(x_0, \ldots, x_{\Lambda-2})$ sont donc $(0, x_0, \ldots, x_{\Lambda-3})$ et $(1, x_0, \ldots, x_{\Lambda-3})$. L'objectif est de construire un arbre de profondeur $(K-\Lambda+1)$ compatible avec l'observation $(z_0, \ldots, z_{K-1})$: K bits consécutifs sont alors identifiés et la séquence entière (a) peut être générée.
L'algorithme est détaillé plus avant dans le cadre 2.3.

• ***Complexité :***
La complexité est ici aussi essentiellement liée au nombre moyen de racines à étudier, à la profondeur et à la largeur des arbres construits avant de trouver le bon. L'auteur montre que la complexité globale est en $\mathcal{O}(K2^{\Lambda})$.

2.2 Attaques probabilistes

Comme nous l'avons précisé dans l'introduction, nous avons regroupé sous la dénomination « probabilistes » les attaques qui modélisent la fonction booléenne comme un canal binaire symétrique sans mémoire (cf. figure 2.3).
Comme nous l'avons vu au chapitre 1, la probabilité de transition, *i.e.* la probabilité de recevoir un '1' alors qu'un '0' a été émis (ou inversement), qui est donc la probabilité qu'une erreur ait lieu, est évaluée à l'aide de la connaissance de la fonction booléenne :

$$p = \frac{1}{2^n} \times max_{u \in \{0,1\}^n} \left| \widehat{f}(u) \right| + 1/2 \ .$$

Attaque déterministe via une recherche arborescente

Entrées de l'algorithme : (z), f, Λ.

1. À $t = 0$, tirer aléatoirement une racine de l'arbre, de longueur $\Lambda - 1$, qui n'a pas été précédemment examinée.
2. À l'instant $t > 0$, examiner les fils de chaque noeud obtenu.
   Soit $(x_t, \ldots, x_{t+\Lambda-2})$ le père.
   $\forall \xi \in \{0, 1\}$
   – si $f(\xi, x_t, \ldots, x_{t+\Lambda-3}) = z_t$, alors $(\xi, x_t, \ldots, x_{t+\Lambda-3})$ est un noeud valide,
   – sinon, il est éliminé.
   Si l'ensemble des noeuds est éliminé, retourner en (1).
   Sinon
   – $t \leftarrow t + 1$,
   – si $t < K - \Lambda + 1$, retourner en (2),
   – sinon, aller en (3).
3. Si un arbre de profondeur $K - \Lambda + 1$ a pu être construit, alors l'état initial a été trouvé. (On peut éventuellement vérifier sur le reste des bits de la séquence que la suite générée à partir de cet état initial et filtrée par la fonction, correspond à la suite observée).

Tab. 2.3 – Attaque déterministe via une recherche arborescente, J. Golic *et. al.*, IEEE Transactions on Computers [42].

lorsque la fonction est équilibrée. $\widehat{f}$ désigne la transformée de Fourier de f ; sa définition a été donnée dans le chapitre 1.

Le problème de cryptanalyse est maintenant transposé en un problème de décodage. Quelle que soit l'attaque considérée, le code pris en compte est un code (N, K) : dans le cas le plus simple, il s'agit du code qui, aux K premiers bits de la séquence (a), associe les N bits qui sont ensuite bruités par le canal binaire symétrique. Dans certains cas, nous allons chercher à décoder non pas (a), mais une forme linéaire appliquée à (a), que nous allons noter (a^{ℓ_u}) : dans ce cas, le mot de code considéré est celui qui, aux K premiers bits de la séquence (a^{ℓ_u}), associe les N bits de (a^{ℓ_u}).

Comment caractériser ce code ?

Afin de décoder le mot de code reçu, il nous faut exploiter certaines propriétés, caractéristiques du code considéré. Les caractéristiques utilisées dans les cryptanalyses probabilistes du registre filtré sont essentiellement :

– les contraintes vérifiées par le mot de code émis ;
– le caractère cyclique du code dual, ce qui se traduit notamment par le fait que les équations sont vraies par décalage, comme nous l'avons déjà souligné au paragraphe 1.6 du chapitre 1.

La complexité des attaques déterministes étudiées au paragraphe 2.1 résidait dans la mémoire

du système Λ, et dépendait très peu du degré ou du poids du polynôme associé au générateur PN. À l'inverse, dans les attaques « probabilistes », le poids et le degré du polynôme générateur sont d'une importance capitale, puisqu'ils conditionnent le nombre de relations vérifiées par la séquence initiale (a) ainsi que leur poids.
La résistance du système s'est donc, au moins partiellement, « **déplacée** » de la suite des espacements (λ_i) vers le générateur PN.

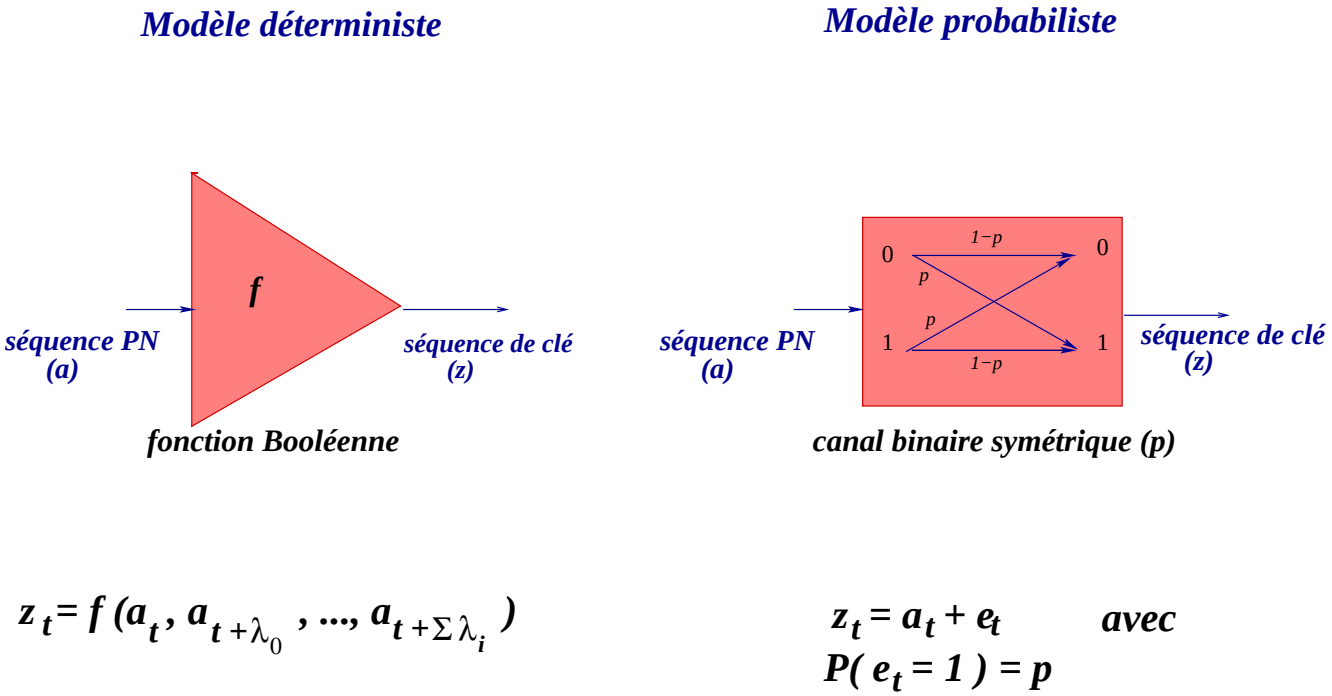


Fig. 2.3 – BSC(p) vs fonction booléenne.

Plusieurs techniques ont été utilisées pour exhiber des contraintes et faire apparaître une structure dans le code de la séquence. Nous avons choisi de classer les différentes attaques probabilistes publiées sur le système en fonction de la génération des contraintes utilisées.

2.2.1 Recherche de multiples creux du polynôme générateur

Nous présentons dans ce paragraphe deux algorithmes de génération d'équations de parité, parmi tous ceux qui ont été publiés. Le lecteur pourra néanmoins consulter les références suivantes afin de trouver d'autres algorithmes : [76] ainsi que [9, 55, 90] qui consistent à chercher des mots de poids creux dans un code.

Génération des équations de parité par élévations au carré successives, W. Meier et O. Staffelbach

Une première façon de générer des équations de parité est suggérée dans l'article de Meier et Staffelbach [66].

Comme nous l'avons vu dans le chapitre 1, la séquence produite par le LFSR est associée au polynôme de rétroaction $g(x) = \sum_{i=0}^{K} g_i x^i$ à coefficients dans $GF(2)$ par l'intermédiaire de la récurrence :

$$\forall n > K,\ a_n = \sum_{i=1}^{K} g_i a_{n-i}$$

c'est-à-dire que

$$g.(a) = 0\ .$$

Le polynôme g étant à coefficients dans $GF(2)$, il vérifie la propriété :

$$g(X^2) = g^2(X),$$

ce qui se traduit sur la séquence (a) par

$$\begin{aligned} g.(a) = 0 \implies \forall h \quad & g^{2^h}.(a) = 0 \\ \implies \quad & a_n = \sum_{i=1}^{K} g_i a_{n-2^h i}\ . \end{aligned} \tag{2.2}$$

Ainsi, à partir du polynôme de rétroaction, il est possible d'obtenir de nouvelles équations de parité à décaler.

- **Quand s'arrête-t-on ?**
  Si la longueur d'observation est N, il nous faut vérifier

  $$2^h K < N\ .$$

- **Quel est le poids des équations obtenues ?**
  D'après l'équation (2.2), nous constatons que le poids des équations obtenues est identique à celui du polynôme générateur.
- **Quel nombre moyen d'équations par bit obtient-on ?**
  Notons T le nombre de coefficients non nuls du polynôme $g(X)$:

  $$T = \text{card}\{i \in [0\ ;\ K] \text{ tel que } g_i \neq 0\}\ .$$

  Comme nous l'avons décrit précédemment, le principe de génération d'équations est d'élever à une puissance de deux le polynôme de rétroaction, ce qui est possible tant que $2^h K < N$. La dernière équation obtenue par élévation à une puissance de deux vérifie donc

  $$h_{max} = \lfloor \log_2 \left(\frac{N}{K} \right) \rfloor\ .$$

  Alors le nombre total d'équations est donné par :

$$\begin{aligned} N_{tot} &= \sum_{i=0}^{h_{max}} \left(N - 2^i K \right) \\ &= (h_{max}+1)N - K(2^{h_{max}+1} - 1)\ . \end{aligned}$$

Chacune de ces équations fait intervenir T termes.
Le nombre moyen d'équations par bit est donc de

$$N_{moy} = \frac{T}{N}\Big((h_{max}+1)N - K(2^{h_{max}+1}-1)\Big) \quad (2.3)$$

$$\approx T\lfloor \log_2(\frac{N}{K})\rfloor\ , \quad (2.4)$$

en approchant $2^{\lfloor \log_2(\frac{N}{K})\rfloor}$ par $\frac{N}{K}$.

Décrivons à présent l'algorithme de cryptanalyse proposé par Meier et Staffelbach.

L'algorithme est fondé sur le principe suivant : un bit de la séquence en réception a d'autant plus de chances d'être correct qu'il satisfait un grand nombre d'équations de parité.
Par ailleurs, la probabilité qu'il satisfasse une équation de parité est directement liée à celle des bits qui interviennent dans ces mêmes équations de parité.

Cet algorithme est à mi-chemin entre l'algorithme BP décrit au chapitre 3 et un algorithme de type « dur », comme ceux décrits dans ce chapitre (cf. [72] page 40 et [17] page 53) : d'une part, il évalue des probabilités sur les bits de la séquence, utilisant les équations de parité, la connaissance du canal de transmission et les probabilités calculées à l'itération précédente. D'autre part, il procède par complémentation : si un bit correspond à une probabilité insuffisamment élevée (ce qui résulte du fait qu'il ne vérifie pas suffisamment d'équations), il est complémenté.

Une valeur seuil intervient dans le déroulement de l'algorithme ; il s'agit de l'espérance du nombre de bits à complémenter, N_{seuil} : si le nombre de bits ayant une corrélation inférieure à $c_{seuil} = 0$ est inférieur à N_{seuil}, corriger la séquence obtenue risque de ralentir la convergence de l'algorithme, le nombre de bits complémentés étant très inférieur à la valeur prévue. Il est plus profitable d'itérer le calcul des probabilités. N_{seuil} est déterminé de telle sorte que le nombre de bits complémentés à juste titre soit optimal, *i.e.* en maximisant l'espérance du nombre de bits correctement corrigés.
Posons N_c le nombre de bits à complémenter, $c_n^i = 2p_n^i - 1$ la corrélation du n-ième bit à l'itération i et s le nombre moyen de relations satisfaites.
L'algorithme de Meier et Staffelbach est alors donné dans le cadre 2.4.

Génération systématique des équations de parité, A. Canteaut et M. Trabbia

Les performances de l'algorithme de Meier et Staffelbach utilisant des équations générées par élévation au carré se révèlent insuffisantes pour attaquer des systèmes dont la probabilité de transition est supérieure à une valeur limite située aux alentours de $p \approx 0.30$, notamment lorsque le polynôme de rétroaction est dense (voir [66]).

A. Canteaut et M. Trabbia ont proposé un algorithme de génération d'équations de parité de poids d donné, qui permet d'obtenir davantage d'équations par bit. Ils s'attaquent ainsi à des systèmes dont la probabilité de transition est de l'ordre de $p \gtrsim 0.4$, quel que soit le poids du polynôme de rétroaction.
Notons d le poids des équations recherchées. L'algorithme de recherche des équations de parité de poids d d'une séquence générée par $g(x)$ est alors donné dans le tableau 2.5.

Algorithme de Meier et Staffelbach

1. Calculs préliminaires : calculer s, N_{seuil}.
2. Initialisation : $\forall\ n\ c_n = c = 2p-1$, $N_c = 0$, $i = 0$.
3. Coeur de l'algorithme :
   (a) Tant que $N_c < N_s$ ou que $i < nbiter_{max}$, itérer le calcul des $c_n^{i\ app}$ en fonction des $c_n^{i-1\ app}$.
   (b) Complémenter les z_n qui, après itération du calcul des corrélations, sont tels que $c_n^{i\ app} < 0$. L'observation est alors (z'_n).
   (c) Vérifier que $\forall n,\ z'_n$ vérifie les relations de parité
   – si toutes les relations sont vérifiées, l'algorithme est terminé,
   – sinon, $i \leftarrow i+1$, $\forall n,\ z_n \leftarrow z'_n$ et retourner en 3a.

Tab. 2.4 – Algorithme de W. Meier et O. Staffelbach, Journal of Cryptology 89, [66].

Le nombre d'équations que l'on peut espérer obtenir est

$$m(d) = \frac{N^{d-1}}{2^K (d-1)!} \tag{2.5}$$

équations à décaler.

L'équation 2.5 montre que le poids des équations est choisi par le cryptanalyste dans une certaine mesure : le degré du polynôme de rétroaction K étant fixé, la longueur d'observation nécessaire à la génération d'équations de parité sera d'autant plus élevée que le poids des équations est faible.
Évaluons alors le nombre moyen d'équations par bit : faisons l'hypothèse que les degrés des polynômes multiples de $g(x)$ sont uniformément répartis entre K et N.
Alors, l'espérance du nombre d'équations par bit est :

$$\begin{aligned} & \frac{T}{N} \sum_{i=K}^{N} \frac{m(d)}{N} \times (N-i) \\ = \ & \frac{Tm(d)}{N} \times \left[(N-K) - \frac{1}{2N} \Big(N(N+1) - K(K-1) \Big) \right] \\ \approx \ & \frac{Tm(d)}{2N} \ . \end{aligned} \tag{2.6}$$

Une fois les équations générées, l'algorithme de décodage appliqué est la version γ de l'algorithme Belief Propagation : c'est un algorithme qui travaille sur les APP totales et qui calcule l'information extrinsèque de façon approchée (voir chapitre 3, paragraphe 3.1.4, page 74).

2.2.2 Opérations sur la matrice génératrice

Une autre façon de générer des équations de parité est de considérer la matrice génératrice associée à la séquence PN et d'effectuer des opérations sur cette matrice : rappelons que la

Algorithme de recherche des multiples de poids d d'un polynôme

1. $\forall i$, calculer le résidu de X^i modulo $g(X)$, $X^i = Res_i \bmod g(X)$, et le ranger dans une table $Table$ telle que :

$$Table(Res_i) = \{\ j \ \ tq\ X^j \ = \ Res_i \ \bmod g(X)\ \}\ .$$

2. $\forall\ i_1 < i_2 < ... < i_{d-2}$ évaluer la somme

$$S = 1 + X^{i_1} + ... + X^{i_{d-2}} \ \bmod g(X)\ .$$

3. Si $Table(S) \neq \emptyset$ alors $\forall j \in Table(S)$

$$1 + X^{i_1} + ... + X^{i_{d-2}} + X^j$$

est un multiple de g de poids d et

$$a_n = a_{n-i_1} + \ldots + a_{n-i_{d-2}} + a_{n-j}$$

est une nouvelle équation de parité vérifiée par tout bit a_n $(n > \max(i_{d-2}, j))$ de la séquence (a).

Tab. 2.5 – Algorithme de recherche par table des multiples de poids d d'un polynôme, A. Canteaut et M. Trabbia, Eurocrypt 2000, [10].

matrice de transition $(K \times K)$ associée au registre à décalage est notée

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & \ldots & 0 & g_K \\ 1 & 0 & \ldots & 0 & g_{K-1} \\ 0 & 1 & & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & g_1 \end{pmatrix}\ .$$

Ainsi, à chaque instant t, l'état interne du LFSR s'exprime de la façon suivante :

$$\mathbf{a_t} = \mathbf{a_{t-1}}\ \mathbf{G}\ = \mathbf{a_0}\ \mathbf{G}^t\ ,$$

où $\mathbf{a_t} = (a_t, a_{t+1}, \ldots, a_{t+K-1})$.
On note $\mathbf{G_i}^p$ la i-ème colonne de la matrice $\mathbf{G}^p$.
La matrice génératrice globale s'écrit :

$$\mathbf{C_{glob}} = \left(\mathbf{I_K}\ \ \mathbf{G_K}^1 \ldots \mathbf{G_K}^m \ldots \mathbf{G_K}^{N-K}\right)$$

qui, en posant $\mathbf{G_K}^m = (c_1^m, \ldots, c_K^m)^\dagger$ s'écrit :

$$\mathbf{C_{glob}} = \begin{pmatrix} 1 & 0 & \dots & 0 & c_1^1 & \dots & c_1^m & \dots & c_1^{N-K} \\ 0 & 1 & \dots & 0 & c_2^1 & \dots & c_2^m & \dots & c_2^{N-K} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & \dots & 1 & c_K^1 & \dots & c_K^m & \dots & c_K^{N-K} \end{pmatrix} .$$

On a ainsi

$$(a_0, \dots, a_{N-1}) = (a_0, a_1, \dots, a_{K-1})\mathbf{C_{glob}} .$$

Les algorithmes décrits ci-après utilisent tous une fenêtre de $(K-B)$ bits ($B < K$) sur laquelle ils recherchent des collisions : le principe est de trouver un ensemble de t colonnes (t le plus petit possible) telles que leur somme s'annule sur les $(K-B)$ dernières coordonnées. Ces collisions permettent :

- soit de générer artificiellement des équations de parité de poids faibles, par l'intermédiaire d'un test exhaustif sur les B premiers bits [74], [72], [17] ;
- soit d'exhiber la structure d'un code dont le décodage est performant et de complexité raisonnable [47], [48], [16] ;
- soit de se ramener aux hypothèses de test d'un algorithme de reconstruction polynomiale [49].

Supposons que $w_1, \dots, w_t$ soient les indices de t colonnes qui coïncident sur leur $(K-B)$ dernières coordonnées. L'équation qui leur correspond est alors de la forme :

$$\sum_{j=0}^{B-1} \sum_{h=0}^{t} c_j^{w_h} a_j = \sum_{h=0}^{t} a_{w_h} .$$

Or la seule connaissance dont nous disposions ensuite sur la séquence (a) est sa version bruitée (z) : ainsi, dans les algorithmes que nous décrivons ci-après, la somme $\sum_{h=0}^{t} a_{w_h}$ est approchée par $\sum_{h=0}^{t} z_{w_h}$. Si t variables bruitées avec une probabilité p sont sommées, la probabilité que cette somme soit égale à 1 est donnée par :

$$P\left(\sum_{h=0}^{t} z_{w_h} = 1\right) = \frac{1-(1-2p)^t}{2} \tag{2.7}$$

(voir lemme de Gallager dans le chapitre 3, page 64). Ainsi, nous constatons d'ores et déjà que les performances des algorithmes que nous allons présenter seront d'autant meilleures que le nombre t de colonnes sommées sera faible (à p fixée).

Décodage par liste, M.J. Mihaljevič *et al.* [74]

L'idée générale de cet algorithme est

- d'une part d'obtenir des équations de parité de poids faible en transformant des équations de poids élevé en relations creuses à l'aide d'un test exhaustif sur une partie de la séquence d'entrée ;
- d'autre part, à l'aide de ces équations et de la séquence reçue, d'isoler deux ensembles de bits globalement plus fiables que les autres, et de vérifier leur pertinence.

Décrivons cette procédure plus en détails. Nous commençons par décrire la recherche des équations de parité, puis nous détaillerons le fonctionnement complet de l'algorithme.

• ***Recherche des équations de parité :***
Un premier paramètre, B, intervient dans le fonctionnement de l'algorithme : il correspond à une fenêtre de bits supposés connus. En pratique, la connaissance de ces B bits sera obtenue via un test exhaustif. On cherche alors des équations de parité de poids $d = 3$ en dehors de cette fenêtre : par le biais d'un test exhaustif, on se ramène au cas d'équations de poids faibles.
Un second paramètre, $D > K - B$, définit le nombre de bits que l'on cherche à décoder. Ce paramètre devrait, à première vue, être égal à $K - B$ puisqu'il s'agit du nombre de bits qu'il nous reste à décoder, les B premiers bits faisant l'objet d'un test exhaustif. En pratique, on choisit $D > K - B$ pour éviter les effets de bords (bits sur lesquels l'algorithme ne parvient pas à prendre de décision fiable), et pour assurer l'indépendance des bits décodés.

Comme nous l'avons rappelé précédemment,

$$\mathbf{C_{glob}} = \left(\mathbf{I_K} \;\; \mathbf{G_K}^1 \dots \mathbf{G_K}^m \dots \mathbf{G_K}^{N-K}\right) \; .$$

Deux sortes d'équations sont générées :
– $i \in [B\,; K-1]$:
On cherche m et w tels que

$$\mathbf{G_K}^m + \mathbf{G_K}^w$$

ait la forme spéciale d'être quelconque sur les B premières coordonnées, de valoir 1 sur la i-ème coordonnée, 0 partout ailleurs. On obtient alors une équation de la forme

$$\sum_{j=0}^{B-1} (c_j^m + c_j^w).a_j + a_i = a_m + a_w \; ;$$

– $i \in [K\,; D-1]$:
On cherche m, w tels que :

$$\mathbf{G_K}^i + \mathbf{G_K}^m + \mathbf{G_K}^w$$

ait des valeurs quelconques sur les B premières coordonnées, et soit nulle partout ailleurs. L'équation obtenue est alors :

$$\sum_{j=0}^{B-1} (c_j^i + c_j^m + c_j^w).a_j = a_i + a_m + a_w \; .$$

Le nombre d'équations que l'on espère obtenir est

$$\frac{1}{2^{K-B}} C_N^2 \; . \tag{2.8}$$

En effet, on dispose d'un ensemble de N éléments (les N colonnes) qui sont par hypothèse uniformément distribués sur l'intervalle des entiers compris entre 0 et $\left(2^{K-B}-1\right)$. La probabilité que deux éléments soient égaux est de $1/2^{K-B}$ et le nombre de couples de colonnes est de C_N^2. D'où l'espérance du nombre de couples de colonnes qui coïncident sur leurs $(K-B)$ dernières

coordonnées.

• ***Description de l'algorithme :***
L'algorithme se décline en trois phases principales : l'étape de pré-calculs, suivie d'une première phase qui consiste, au moyen d'un test exhaustif sur B bits d'entrée, à exhiber les deux candidats les plus fiables ; enfin, on vérifie à l'aide d'un test par corrélation l'éventuelle validité de l'un d'entre eux. Les paramètres régissant cette étape de vérification sont issus de l'article [88]. Le seuil $\mathcal{T}$ notamment, est évalué de la façon suivante : il s'agit du plus petit entier tel que la relation

$$\frac{p}{1-p}\left(\frac{1+(1-2p)^2}{1-(1-2p)^2}\right)^{|\Omega_i|-2\mathcal{T}(i)} \leq 1$$

soit vérifiée, $|\Omega_i|$ désignant le nombre d'équations de parité issues des méthodes (1) et (2) auxquelles le bit i appartient.[1]

L'algorithme complet est développé dans le cadre 2.6.

• ***Complexité :***
On peut montrer (se référer à l'article pour plus de détails) que :
- la complexité de la phase de pré-calculs peut être ramenée à $D(N-K)$ en terme de calculs, et qu'elle est proportionnelle à $N-K$ en terme de mémoire ;
- la complexité du noyau de l'algorithme, en notant $|\Omega|$ le nombre moyen d'équations de parité par bits, et T le poids du polynôme de rétroaction, est en $\mathcal{O}\Big(2^B\left[\ |\Omega|\,(D-B)+T\,(M-K)\ \right]\Big)$ additions modulo 2.

Décodage itératif « dur », M.J. Mihaljevič *et. al.* [72]

Cet algorithme utilise quasiment le même ensemble d'équations de parité que l'algorithme de « list-decoding » précédemment décrit ([74]), et effectue un décodage itératif « dur », au sens où les algorithmes itératifs proposés travaillent sur les bits (ou leurs complémentaires), non sur des probabilités. Ce principe de décodage, ainsi que celui décrit ultérieurement, [17] (cf. page 53) n'est pas sans rappeler la version A de l'algorithme de Gallager décrit dans [35] et dont nous rappelons brièvement le principe dans le chapitre 3 (voir paragraphe 3.1, page 60).

• ***Génération des équations de parité :***
L'algorithme est fondé sur l'utilisation d'équations de poids creux, en dehors d'une fenêtre de bits sur laquelle on fait des tests exhaustifs. Cette procédure permet, comme nous l'avons vu dans le paragraphe précédent, d'obtenir des relations creuses à partir d'équations de poids élevé.

Soit B la taille de la fenêtre sur laquelle on fait des tests exhaustifs.
Trois méthodes permettent de générer les équations :

[1] Les notations Ω_i pour désigner l'ensemble des équations de parité satisfaites par le i-ième bit, et $|\Omega_i|$ pour en désigner le cardinal sont locales. Dans les autres chapitres, elles seront remplacées respectivement par $\mathcal{E}^b(i)$ et μ_b.

Algorithme du « list-decoding »

- **Paramètres d'entrée :**
  la séquence (z), $g(X)$, la probabilité de transition p du canal binaire symétrique modélisant la fonction booléenne, la probabilité de rater l'événement P_m, la probabilité de fausse alarme, P_f, les paramètres B et D ;
- **Phase de pré-calculs :**
  - Les probabilités p, P_m, P_f étant données, évaluer le nombre M de bits requis pour la phase de vérification par corrélation, ainsi que le seuil qui lui est associé, noté $\mathcal{T}$ ([88]),
  - déterminer l'ensemble des équations de parité selon la méthode décrite au paragraphe précédent. L'ensemble des équations dans lesquelles le bit i intervient est noté Ω_i ;
- **Coeur de l'algorithme :**
  1. Choisir un nouveau B-uplet correspondant aux B premiers bits de la séquence, à partir de $(z_0, \ldots, z_{B-1})$ et d'un nouveau motif d'erreur qui n'a pas été considéré précédemment. S'il n'en existe pas, l'algorithme se termine et l'initialisation du registre n'a pas été trouvée.
  2. $\forall i \in [B\ ; D-1]$, calculer le nombre d'équations de Ω_i qui sont vérifiées. Sélectionner alors :
     - les $(K-B)$ bits qui vérifient le plus d'équations de parité, et en supposant qu'ils sont corrects (et sous réserve d'indépendance), réévaluer les bits a_i pour $i \in [B; K-1]$;
     - les $(K-B)$ bits qui vérifient le moins d'équations et les complémenter. En supposant que l'on a ainsi obtenu $(K-B)$ bits corrects et indépendants, générer les bits a_i pour $i \in [B; K-1]$.
  3. Tester la validité des deux séquences obtenues. Pour cela, les corréler avec la séquence de sortie : pour chacune des deux suites, évaluer $S = \sum_{i=0}^{M-1} a_i + z_i$ en prenant pour $(a_0, \ldots, a_{B-1})$ le B-uplet sélectionné à l'étape 1, et pour $(a_B, \ldots, a_{K-1})$ les $(K-B)$-uplets générés à l'étape précédente.
     Si $S \leq \mathcal{T}$ l'algorithme se termine. Sinon, retourner à l'étape 1.

Tab. 2.6 – Algorithme du « list-decoding », M.J. Mihaljevič *et. al.*, FSE 2001, [74].

1. On cherche une colonne ${\mathbf{G_K}}^m$ ayant des valeurs quelconques sur ses B premières positions, et au plus deux valeurs non nulles sur les $K-B$ positions restantes.
   On obtient alors une équation du type :

$$\sum_{j=0}^{B-1} c_j^m . a_j + a_m + a_{i_1} + a_{i_2} = 0 \ ,$$

   avec i_1, i_2 inférieurs à K, supérieurs à $K-B$.

2. On cherche m et w tels que
$$\mathbf{G_K}^m + \mathbf{G_K}^w$$
ait la forme spéciale d'être quelconque sur les B premières coordonnées, de valoir 1 sur la i-ème coordonnée, 0 partout ailleurs. On obtient alors une équation de la forme
$$\sum_{j=0}^{B-1}(c_j^m + c_j^w).a_j + a_i = a_m + a_w \ .$$

3. $i \in [K\ ; D-1]$ On cherche m, w tels que :
$$\mathbf{G_K}^i + \mathbf{G_K}^m + \mathbf{G_K}^w$$
ait des valeurs quelconques sur les B premières coordonnées, et soit nul partout ailleurs. L'équation obtenue est alors :
$$\sum_{j=0}^{B-1}(c_j^i + c_j^m + c_j^w).a_j = a_i + a_m + a_w \ .$$
On peut généraliser cette procédure à la somme de t colonnes.

Remarquons que les deux premiers types d'équations ne concernent que des bits d'information, alors que seuls des bits de parité interviennent dans le dernier type d'équation (hormis les bits de la fenêtre de taille B supposés connus).

• ***Description de l'algorithme :***
Deux algorithmes sont proposés en fonction du type d'équations de parité générées, équations impliquant des bits d'information ou non.
Les entrées de l'algorithme sont (z), N, K, B, et un seuil $\mathcal{T}$, les équations de parité générées à l'aide des méthodes (1), (2) dans le cas où l'on applique l'OSDA, de la méthode (3) dans le cas où l'IDA est utilisé.
L'algorithme général est alors donné dans le cadre 2.7

Description du cœur de l'algorithme : l'OSDA, l'IDA
Pour le cœur de l'algorithme, l'algorithme choisi est soit l'OSDA, soit l'IDA.
- **OSDA** : One Step Decoding Algorithm.
  L'OSDA utilise les équations de parité générées par les méthodes (1) et (2) (*i.e.* équations concernant aussi des bits d'information).
  Cet algorithme comporte deux étapes :
  - $\forall i \in [B; K-1]$, en prenant pour initialisation la séquence $z_B, \ldots, z_{K-1}$, évaluer le nombre d'équations de parité vérifiées par le bit i ;
  - si ce nombre d'équations vérifiées est inférieur au seuil $\mathcal{T}(i)$, complémenter le bit, sinon le laisser inchangé.

  Le seuil $\mathcal{T}$ est déterminé à partir de l'article [88], comme nous l'avons précisé plus haut (cf. page 40).
- **IDA** : Iterative Decoding Algorithm
  On utilise ici les équations générées par la méthode (3), équations qui concernent les bits de parité et on cherche à décoder un sous-ensemble des bits de parité de taille $N^* \leq N$.

Algorithme de décodage itératif dur

- **Paramètres d'entrée** :
  (z), $g(X)$, p, T, B.
- **Phase de pré-calculs :**
  - évaluer le seuil $\mathcal{T}(i)$ pour tout bit ;
  - calculer les équations de parité en utilisant la méthode décrite ci-dessus.
- **Coeur de l'algorithme :**
  1. Choisir un nouveau B-uplet correspondant aux B premiers bits de la séquence.
  2. Appliquer l'algorithme OSDA ou l'algorithme IDA (ces algorithmes sont décrits ci-après).
  3. Vérifier par un calcul de corrélation que l'estimation des bits $a_0, \ldots, a_{K-1}$ est correcte : une fois la séquence entière régénérée, calculer $S = \sum_{i=0}^{N-1}(a_i + z_i) \bmod 2$. Si $S \leq \mathcal{T}$, le vecteur $a_0, \ldots, a_{K-1}$ est considéré comme l'initialisation correcte du générateur PN. Sinon, retourner en (1).

Tab. 2.7 – Algorithme de décodage itératif dur, M.J. Mihaljevič *et. al.*, FSE 2000, [72].

Cet algorithme fait intervenir l'équivalent d'une information extrinsèque (couramment utilisée dans les algorithmes de décodage itératif classiques, voir chapitre 3), mais cette information extrinsèque est ici « hard » : pour un bit dans une équation donnée, on tient compte de l'information apportée par les autres équations auxquelles il appartient, cette information se traduisant en nombre d'équations vérifiées.
On introduit quelques notations supplémentaires :
- $a_{i,m}$ est la valeur potentielle du bit a_i dans l'équation m ;
- $\sigma_i(m)$ est la valeur de la m-ième équation de parité du bit i, évaluée à l'aide des $a_{j,m}$
- $\forall i \in [K+1; N^*]$, Ω_i^* désigne l'ensemble d'équations de parité du bit i obtenues par la méthode (3) décrite précédemment, *i.e.* des équations qui contiennent au plus B bits d'information (les B premiers) et trois bits de parité.

L'algorithme s'écrit alors :

1. Initialiser $a_i = z_i$, $a_{i,m} = z_i$.
2. – Si le nombre d'itérations est supérieur au nombre maximal d'itérations autorisées (trente), l'algorithme s'arrête sans avoir trouvé (a) ;
   – sinon, $\forall i, m$, évaluer $\sigma_m(i)$: s'ils sont tous nuls, l'algorithme se termine et la sortie de l'algorithme est constituée des a_i. Sinon, aller à l'étape suivante.
3. $\forall i$:
   – si $\sum_{m \in \Omega_i^*} \sigma_i(m) \geq |\Omega_i^*|/2$, complémenter le bit a_i ;
   – $\forall j \in \Omega_i^*$, si $\sum_{m \in \Omega_i^* \setminus j} \sigma_i(m) \geq |\Omega_i^* \setminus j|/2$, complémenter $a_{i,m}$.
   Si aucune des deux étapes ci-dessus n'a impliqué de complémentation, l'algorithme se termine sans que les a_i n'aient été identifiés.

Sinon, retourner à l'étape 2.

• ***Complexités :***
Dans cette partie, nous présentons les complexités de l'algorithme de décodage itératif hard (sans la phase de pré-calculs), en fonction de l'algorithme interne (OSDA ou IDA) utilisé.
Lorsque l'OSDA est utilisé, le décodage ne requiert pas de mémoire ; la mémoire nécessaire lors de l'application de l'IDA est de l'ordre de N^*.
En termes de calculs, on peut montrer que les complexités prennent les valeurs suivantes :

– en notant $|\Omega|$ le nombre moyen d'équations de parité par bit générées par les méthodes 1 et 2, t le nombre moyen de bits dans une équation, T le poids du polynôme générateur, l'OSDA requiert une complexité de calculs de l'ordre de

$$2^B\Big[(K-B)|\Omega|t+(N-K)T\Big]$$

additions modulo 2 ;

– en notant $|\Omega^*|$ le nombre moyen d'équations de parité par bit générées par la méthode 3, t^* le nombre moyen de bits dans une équation, T le poids du polynôme générateur, I le nombre d'itérations, l'IDA requiert une complexité de calculs de l'ordre de

$$2^B\Big[I(N^*-K)|\Omega^*|(|\Omega^*|-1)t^*+(N-K)T\Big]$$

additions modulo 2.

Attaques « turbo » de Johansson et Jönsson, EUROCYPT'99 et CRYPTO'99 [47], [48]

Les opérations sur la matrice génératrice ont également donné lieu à des attaques de type « turbo » : la première attaque [47] de Johansson et Jönsson consiste à exhiber un code convolutif et à appliquer un décodage de Viterbi sur ce code. L'attaque a ensuite été améliorée dans [48], où, au moyen de permutations, un turbo-décodage peut être appliqué.

Voyons cette attaque plus en détail ; la structure d'un code convolutif est rappelée dans l'annexe C.

• ***Extraction d'un code convolutif :***
L'objet de l'article [47] est de faire apparaître un code convolutif de mémoire B dans le code déduit du LFSR et dont la matrice génératrice s'écrit :

$$\mathbf{C_{glob}} = \begin{pmatrix} 1 & 0 & \dots & 0 & c_1^1 & \dots & c_1^m & \dots & c_1^{N-K} \\ 0 & 1 & \dots & 0 & c_2^1 & \dots & c_2^m & \dots & c_2^{N-K} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 0 & 0 & \dots & 1 & c_K^1 & \dots & c_K^m & \dots & c_K^{N-K} \end{pmatrix}.$$

Pour ce faire, $\mathbf{C_{glob}}$ est réécrite sous la forme :

$$\mathbf{C_{glob}} = \begin{pmatrix} \mathbf{I_{B+1}} & \mathbf{C_{B+1}} \\ \mathbf{0_{K-B-1}} & \mathbf{C_{K-B-1}} \end{pmatrix}$$

où

- $\mathbf{I_{B+1}}$ est la matrice identité de taille $(B+1) \times (B+1)$;
- $\mathbf{0_{K-B-1}}$ est la matrice nulle de taille $(K-B-1) \times (B+1)$;
- $\mathbf{C_{B+1}}$ est de taille $(B+1) \times (N-B-1)$;
- $\mathbf{C_{K-B-1}}$ est de taille $(K-B-1) \times (N-B-1)$.

Nous allons chercher des collisions sur les colonnes de la matrice $\mathbf{C_{K-B-1}}$: si l'on réussit à trouver t colonnes de $\mathbf{C_{K-B-1}}$ dont la somme est nulle et telles que la somme des colonnes de même indice de la matrice $\mathbf{C_{B+1}}$ ne soit pas nulle sur sa $(B+1)$-ième coordonnée, alors on obtient une équation de la forme :

$$a_{B+1} + \sum_{j=1}^{B} \sum_{w=1}^{t} c_j^{i_w} \; a_j = \sum_{l=1}^{t} a_{i_w} \; .$$

Réécrivons cette équation

$$a_{B+1} + \sum_{j=1}^{B} \sum_{w=1}^{t} c_j^{'\, i_w} \; a_{B+1-j} = \sum_{l=1}^{t} a_{i_w} \; ,$$

avec $c'_j = c_{B+1-j}$.
Cette équation est valable par décalage.
Avant d'écrire sa forme décalée, supposons que nous ayons obtenu m équations de ce type, indexons chacune d'entre elles par un indice h, et écrivons la h-ième équation sous la forme :

$$a_n + \sum_{i=1}^{B} \gamma_i^h a_{n-i} = \sum_{j=1}^{t} a_{i_j^h} \; .$$

Chaque bit a_n appartient à m équations de ce type. Nous construisons alors une nouvelle séquence (r) qui fera office de nouvelle séquence reçue.

$$r = r_0^0, r_0^1, r_0^2, \ldots, r_0^m, \; r_1^0, \ldots, r_1^m, \ldots$$

avec

$$\begin{aligned} r_0^0 &= z_n \\ \forall \; h \; \in [1; m], \; r_0^h &= \sum_{j=1}^{t} z_{i_j^h} \end{aligned}$$

qui représente donc une version bruitée d'une nouvelle séquence (b) telle que

$$\begin{aligned} b_n^0 = &\; a_n \\ \forall \; h \; \in [1 \; ; m], \; b_0^h &= \sum_{j=1}^{t} a_{i_j^h} \; . \end{aligned}$$

Il apparaît ainsi un code convolutif de rendement $\frac{1}{m+1}$, telle que $b_n = a_n + \sum_{i=1}^{B} \gamma_i^h a_{n-i}$, dont on observe une version bruitée en sortie de la fonction :

$$\begin{aligned} P(r_n^0 = b_n^0) &= 1 - p \\ P(r_n^h = b_n^h) &= \frac{1}{2} + 2^{t-1} (\frac{1}{2} - p)^t \; . \end{aligned}$$

• ***Algorithme de décodage*** :
Une fois que le code convolutif est exhibé, on peut appliquer l'algorithme de Viterbi (voir annexe D). L'algorithme complet est donné dans le cadre 2.8. Comme l'état initial du treillis n'est pas connu, l'algorithme de Viterbi est appliqué sur une fenêtre de taille $10K$; les symboles décodés sont ceux qui sont situés au milieu de la fenêtre.

Cryptanalyse du registre filtré utilisant un code convolutif

– **Entrées de l'algorithme :**
t, $g(X)$, p, (z).
– **Pré-calculs :**
Exhiber le code convolutif en cherchant des collisions sur t colonnes (voir plus haut). Soit (b) le code convolutif, (r) sa version bruitée.
Tracer le treillis du code convolutif (b).
– **Coeur de l'algorithme :**

1. Initialisation :

$$\begin{aligned} P(r_n^0 = b_n^0) &= 1-p \\ \forall h \in [1;m], \quad P(r_n^h = b_n^h) &= \frac{1}{2} + 2^{t-1}(\frac{1}{2}-p)^t \end{aligned}$$

2. Appliquer l'algorithme de Viterbi sur une fenêtre de taille $K + 10B$.
3. Décoder $b^0_{5B+1}, \ldots b^0_{5B+K}$.
4. En déduire la clé secrète.

Tab. 2.8 – Extraction d'un code convolutif, et attaque via l'algorithme de Viterbi, Johansson et Jönsson, Eurocrypt 99, [47].

• ***Extension de la recherche de code convolutif à celle de turbo-code, Johansson et Jönsson, CRYPTO'99, [48]***
Au lieu d'utiliser un décodeur de Viterbi, les auteurs ont suggéré d'appliquer l'algorithme BCJR (voir annexe F) qui décrit le treillis dans les deux sens, et ce, de façon « conjointe ».
Par ailleurs, une autre amélioration potentielle consiste à utiliser des méthodes de turbo-décodage (voir annexe E), en faisant apparaître de nouveaux codes convolutifs. Pour ce faire, les auteurs suggèrent de permuter les colonnes de la matrice génératrice $\mathbf{C_{glob}}$ en se restreignant aux colonnes dont l'indice est compris dans une fenêtre $[B; B+10K]$ sur laquelle le décodage est effectué. Dans un deuxième temps, ils cherchent des équations du type de celles trouvées lors de l'extraction du code convolutif, impliquant chaque bit (a_n) pour tout $n \in [B; B + 10K]$. La matrice ayant subi une permutation, les équations ne sont plus systématiquement vraies par simple décalage ; il leur faut donc chercher un ensemble d'équations pour chaque bit dont l'indice est compris entre B et

$B + 10K$ (fenêtre sur laquelle les codes convolutifs sont décodés). Ensuite, il suffit d'appliquer le principe du décodage turbo comme il est décrit en annexe.

Les deux algorithmes qui suivent ne sont ni de type turbo, ni de type itératif, mais utilisent tous deux des opérations sur la matrice génératrice : le premier consiste à faire un décodage à maximum de vraisemblance sur un sous-code déduit du code de départ, le second utilise l'algorithme de Sudan de reconstruction polynomiale.

Décodage à maximum de vraisemblance d'un sous-code, V.V. Chepyzhov *et. al.*, FSE 2000, [16]

Le principe de l'algorithme qui est décrit ci-dessous est de faire apparaître dans le code issu de la séquence PN un plus petit code, qui sera décodé par un décodage à maximum de vraisemblance.

• ***Principe de l'extraction du sous-code :***
Afin de simplifier l'écriture de la recherche d'équations, nous supposerons que le nombre de colonnes sommées est $t = 2$. Pour faire apparaître le sous-code, nous cherchons des paires de colonnes de $\mathbf{C_{glob}}$ qui pour un $B < K$ donné, coïncident sur leurs $(K - B)$ dernières coordonnées. Si cela est impossible ou que le nombre de paires de colonnes trouvées est très faible, nous chercherons des t-uplets, $t > 2$ de colonnes dont la somme est nulle sur les $(K - B)$ dernières coordonnées.
Soit ν le nombre de paires de colonnes obtenues, $(i_1, j_1), \ldots, (i_\nu, j_\nu)$ les indices de ces couples de colonnes.

Alors, le sous-code a pour matrice

$$\mathbf{C_{ss-code}} = \begin{pmatrix} c_1^{i_1} + c_1^{j_1} & \ldots & c_1^{i_\nu} + c_1^{j_\nu} \\ \vdots & \ddots & \vdots \\ c_B^{i_1} + c_B^{j_1} & \ldots & c_B^{i_\nu} + c_B^{j_\nu} \end{pmatrix} .$$

Si l'on appelle $(A_1, \ldots, A_\nu) = (a_{i_1} + a_{j_1}, \ldots, a_{i_\nu} + a_{j_\nu})$, alors on peut écrire :

$$(A_1, \ldots, A_\nu) = (a_0, \ldots, a_{K-1})\mathbf{C_{ss-code}} .$$

Le sous-code que l'on a exhibé est de taille $(K \times \nu)$, $(a_0, \ldots, a_{K-1})$ sont les bits d'information qui correspondent aux bits de parité $(A_1, \ldots, A_\nu)$.
La séquence correspondant aux « bits » $(A_1, \ldots, A_\nu)$ bruités par le canal s'écrit :

$$(Z_1, \ldots, Z_\nu) = (z_{i_1} + z_{j_1}, \ldots, z_{i_\nu} + z_{j_\nu}) ,$$

ce qui correspond à une séquence d'erreur :

$$(e_{i_1} + e_{j_1}, \ldots, e_{i_\nu} + e_{j_\nu}) .$$

La probabilité de bruit équivalente est alors plus élevée, puisqu'elle correspond à la somme de deux bits d'erreur :

$$p_{\text{équiv}} = 2p(1 - p) .$$

• ***Description de l'algorithme :***
L'algorithme complet est décrit dans le cadre 2.9.

Cryptanalyse du registre filtré via l'algorithme ML.

- **Paramètres d'entrée :**
  K, B, (z), t.
- **Phase de pré-calculs :**
  Faire apparaître le sous-code de dimension $(K \times \nu)$ en cherchant t colonnes dont la somme s'annule sur les $(K - B)$ dernières coordonnées.
- **Coeur de l'algorithme :**
  1. Calculer $(Z_1, \ldots, Z_\nu)$.
  2. Décoder le sous-code en faisant un décodage à maximum de vraisemblance : on génère tous les 2^B mots du sous-code, et on choisit celui qui est à distance minimale du mot de code reçu.

Tab. 2.9 – Décodage à maximum de vraisemblance d'un sous-code, Chepyzhov *et. al.*, FSE 2000, [16].

Remarque : après application de l'algorithme, nous avons décodé B bits sur les K bits d'état initial. Nous pouvons alors itérer l'application de l'algorithme en considérant maintenant que le code de départ est non plus un code $(K \times N)$, mais $\Big((K - B) \times N\Big)$.

• ***Performances, complexité :***

- l'espérance de la longueur du sous-code est

$$E(\nu) = \frac{N(N-1)}{2} 2^{K-B} \ .$$

  La justification est identique à celle donnée page 39 où l'on évaluait l'espérance du nombre d'équations obtenues ;
- la longueur de séquence nécessaire au succès de l'algorithme est

$$N \approx 1/4(2B\ t!\ ln2)^{1/t}\ \epsilon^{-2}\ 2^{\frac{B-K}{t}} \ ,$$

  où t est le nombre de colonnes impliquées dans la recherche de chacune des collisions ;
- la complexité de l'algorithme est

$$2^B\ B\ \frac{2\log_2 2}{(2\epsilon)^{2t}} \ .$$

Deux extensions de l'algorithme précédent ont été publiées : Jönsson et Johansson l'ont étendu à la cryptanalyse du système LILI-128 dans [52] ; Canteaut et Filiol l'ont appliqué à la cryptanalyse du registre filtré [11].

Détaillons plus avant ces deux attaques : le principe de ces deux attaques est de trouver un sous-code décodable par un décodage à maximum de vraisemblance. La différence avec l'algorithme présenté ci-dessus est que ces deux algorithmes tiennent compte des propriétés de la fonction booléenne de filtrage : ils utilisent en effet plusieurs combinaisons linéaires de la séquence initiale ayant une corrélation non nulle avec la sortie pour former un code plus grand, associé à une matrice génératrice dans laquelle les collisions seront recherchées.

Le lecteur pourra trouver une description du système LILI-128 dans [22]. Rappelons toutefois que ce générateur de clé est constitué de deux sous-systèmes : un premier générateur de mémoire égale à $K_1 = 39$, dont la valeur de sortie est comprise entre 1 et 4, et qui contrôle le décalage du second générateur. Ce second générateur est un LFSR filtré par une fonction booléenne à $n = 10$ entrées (donnée dans l'annexe A), dont la sortie constitue la séquence de clé. La différence essentielle avec un registre filtré classique réside donc dans le décalage potentiellement irrégulier du second registre, décalage dont la valeur est dictée par un premier registre. Dans notre cas, nous supposons que nous connaissons la sortie du premier registre par le biais d'un test exhaustif. Considérons également pour l'instant que sa sortie est constante, égale à 1.

Un examen du spectre de la fonction f révèle qu'il existe deux cent quarante combinaisons linéaires de la séquence d'entrée qui sont à distance minimale de la séquence de sortie. La probabilité d'erreur qui leur est associée est de $p = 0.46875$. Chacune des séquences déduites d'une combinaison linéaire de la séquence (a) est une fonction des K premiers bits $(a_0, \ldots, a_{K-1})$. Soit N leur longueur. Chacune d'entre elles est donc associée à une matrice $(K \times N)$ telle que :

$$(a_0^{\ell_i}, \ldots, a_{N-1}^{\ell_i}) = (a_0, \ldots, a_{K-1})\mathbf{C}_{\mathrm{glob}}^{\ell_i} \ ,$$

où nous notons (a^{ℓ_i}) la i-ième séquence issue d'une combinaison linéaire de la séquence initiale, et $\mathbf{C}_{\mathrm{glob}}{}^{\ell_i}$ la matrice génératrice qui lui est associée.

Les auteurs ont alors considéré une « super » matrice obtenue en concaténant les deux cent quarante matrices du type $\mathbf{C}_{\mathrm{glob}}^{\ell_i}$ de taille $\Big(K \times (240.N)\Big)$.

$$\mathbf{C}_{\mathrm{super}} = (\mathbf{C}_{\mathrm{glob}}^{\ell_1}, \ldots, \mathbf{C}_{\mathrm{glob}}^{\ell_{240}}).$$

Ils suggèrent ensuite de se ramener à l'algorithme précédent en cherchant des triplets de colonnes dont la somme s'annule sur les $(K - B)$ dernières coordonnées. Le sous-code obtenu est alors décodé en utilisant un décodage à maximum de vraisemblance.

Dans l'article [11], Canteaut et Filiol ont appliqué le même principe d'algorithme à un registre filtré par une fonction booléenne. Cependant, au lieu de ne tenir compte que des pics de corrélation, ils ont considéré tous les codes issus de combinaisons linéaires de la séquence d'entrée, associées à une corrélation non nulle avec la sortie. Les résultats obtenus montrent que le nombre d'entrées de la fonction n'a que peu d'influence sur la longueur de séquence d'observation nécessaire au succès de l'algorithme.

Algorithme de reconstruction polynomiale [49]

L'idée générale de cet algorithme est de modéliser chaque bit de la séquence obtenue en sortie de la fonction booléenne comme la valeur prise par un polynôme linéaire à plusieurs variables en

différents points, et bruité par un canal binaire symétrique.

On cherche alors à appliquer les algorithmes de recherche de polynômes multi-variables à notre cas d'étude.

• ***Modélisation du problème :***
On appelle polynôme d'état initial :

$$A(x) = a_0x_0 + a_1x_1 + \ldots + a_{K-1}x_{K-1} \ .$$

Chaque bit de la séquence initiale (a) s'exprime linéairement en fonction des K premiers bits $a_0, \ldots, a_{K-1}$, ce qui revient à évaluer A en un certain point. Dans notre système de notations, cela correspond à

$$a_i = A(\mathbf{c^i})$$

où $\mathbf{c^i} = (c_1^i, \ldots, c_{K-1}^i)$.
La séquence bruitée s'écrit alors : $\forall i, z_i = A(\mathbf{c^i}) + e_i$ avec $P(e_i = 1) = p$.

• ***Principe et description des algorithmes :***
Les algorithmes de cryptanalyse proposés dans cette partie sont inspirés d'un algorithme de reconstruction polynomiale développé par Goldreich, Rubinfeld, et Sudan [91] : en supposant que l'on puisse accéder, à l'aide d'un oracle, à la sortie bruitée d'une fonction $f : \{0,1\}^K \longrightarrow \{0,1\}$, l'algorithme fournit une liste de toutes les fonctions linéaires à K variables qui coïncident avec f sur une fraction d'au moins δ valeurs. Nous décrivons brièvement cet algorithme.

Considérons le polynôme linéaire : $Q(\mathbf{x}) = \sum_{i=1}^{K} q_i x_i$. L'algorithme utilise la notion de i-préfixe : on appelle i-préfixe du polynôme Q, le polynôme à i variables $Q(x_1, \ldots, x_i, 0, \ldots, 0) = \sum_{j=1}^{i} q_j x_j$.
On procède alors en K étapes : à l'étape i, on « agrandit » chacun des $(i-1)$ préfixes sélectionnés précédemment en ajoutant un 1 ou un 0 pour former deux fois plus de i-préfixes potentiels.
On applique ensuite une opération de discrimination de la façon suivante : soit $(q_1, \ldots, q_i, 0, \ldots, 0)$ un candidat au préfixe. On tire $\nu = poly(K/|p-0.5|)$ séquences $(s_{i+1}, \ldots, s_K)$ de façon uniforme. Pour $\xi = 0$ ou 1, on estime

$$P(\xi) = Prob_{r_1,\ldots,r_i}\left[f(r_1, \ldots, r_i, s_{i+1}, \ldots, s_K) = \sum_{j=1}^{i} p_j r_j + \xi \right]$$

Un candidat au préfixe passe le test, si pour un ξ donné, il existe au moins une séquence $(s_{i+1}, \ldots, s_K)$ telle que $P(\xi)$ soit supérieure à $1/2 + \epsilon/3$ $(\epsilon = p - 1/2)$.
Cette procédure de sélection des i-préfixes laisse passer le bon polynôme avec une probabilité élevée, et permet d'éliminer un grand nombre de mauvais candidats.

Dans ce qui précède, il est à noter que le polynôme Q est évalué en une succession de points ayant une forme particulière, du type $(* * *, s_{i+1}, \ldots, s_K)$ afin de déterminer la quantité $P(\xi)$. Dans notre cas, n'ayant pas d'oracle, nous ne pouvons pas sélectionner les points sur lesquels le polynôme est évalué. Nous ne pouvons qu'utiliser les points dont nous disposons et au mieux faire des opérations sur ces points pour que l'algorithme soit applicable.
Le polynôme étant linéaire, $\sum_{j=0}^{t} A(\mathbf{c}^{w_j}) = A(\sum_{j=0}^{t} \mathbf{c}^{w_j})$, et la probabilité de bruit correspondant est alors $P\left[\sum_{j=0}^{t} z_{w_j} = A(\ \sum_{j=0}^{t} \mathbf{c}^{w_j}\ )\right] = 1/2 + 2^{t-1}\epsilon^t$.
Ainsi, en faisant des sommes de t termes, on cherche à obtenir des vecteurs de la forme souhaitée. Un compromis est néanmoins à trouver car plus t est élevé, plus le nombre de vecteurs

obtenus sera important, mais plus la probabilité d'erreur associée au nouveau vecteur sera elle aussi élevée, comme nous l'avons déjà observé précédemment (cf. page 38, équation 2.7).

Deux algorithmes de cryptanalyse inspirés de ce qui précède sont donnés dans les tableaux 2.10 et 2.11 : l'algorithme figuré dans le tableau 2.10 contient une version « basique » de l'algorithme de reconstruction, qui n'utilise pas la notion de i-préfixe. Remarquons que, en fixant les paramètres $\nu = 1$ et $\mathbf{s_1} = \mathbf{0_{K-B}}$, on se ramène à l'algorithme ML décrit dans le tableau 2.9.

Algorithme de reconstruction polynomiale I

- **Entrées de l'algorithme :**
  z, $\mathbf{c}^1, \ldots, \mathbf{c}^{N-K}$, t, B, ν.
- **Pré-calculs :**
  Choisir ν vecteurs $\mathbf{s_1}, \ldots, \mathbf{s}_\nu$ de taille $(K - B)$. Pour chaque $\mathbf{s_i}$, chercher les combinaisons linéaires de $\mathbf{c}^i$ de la forme

$$\hat{\mathbf{c}}(i) = \sum_{j=0}^{t} \mathbf{c}^{w_j} = (\hat{c_1}, \ldots, \hat{c_B}, \mathbf{s_i})$$

  Pour tout i, mémoriser les $\hat{\mathbf{c}}(i) = \sum_{j=0}^{t} \mathbf{c}^{w_j}$, et les $\hat{z}(i) = \sum_{j=0}^{t} z_{w_j}$ associés. S_i désigne le cardinal de cet ensemble de couples.
- **Coeur de l'algorithme :**
  1. Choisir un B-uplet $(a_0, \ldots, a_{B-1})$ qui n'a pas été précédemment testé. Initialiser *dist* à 0. Pour chaque $\mathbf{s_i}$, parcourir l'ensemble des S_i couples $(\hat{\mathbf{c}}i), \hat{z}(i))$ et évaluer le nombre de fois que :

$$\sum_{j=1}^{B} a_j \hat{c}_j = \hat{z}_i$$

     où $\hat{\mathbf{c}}(i) = (\hat{c_1}, \ldots, \hat{c_B}, \mathbf{s_i})$. On appelle *num* cette quantité. Mettre à jour la quantité $dist := dist + (S_i - 2\ num)^2$.
  2. Si *dist* est la plus grande valeur obtenue jusqu'alors, stocker $(a_0, \ldots, a_{B-1})$.
  3. La sortie est le B-uplet $(a_0, \ldots, a_{B-1})$ qui correspond à la valeur de *dist* la plus élevée.

Tab. 2.10 – Algorithme de reconstruction polynomiale n'utilisant pas la notion de i-préfixe, Johansson et Jönsson, CRYPTO 2000, [49].

Le principe de ces algorithmes est fondé sur la distribution de *num* : en effet on montre aisément que *num* suit une loi binomiale dont la variance ne dépend que de la validité du B-uplet supposé. Si l'on possède suffisamment de combinaisons linéaires de la séquence de départ,

on va ainsi pouvoir séparer les deux hypothèses « le B-uplet supposé est correct », « le B-uplet supposé est incorrect ».

On peut également montrer que l'utilisation d'une distance Euclidienne est optimale : statistiquement, elle nous permet d'obtenir le candidat associé à la probabilité la plus élevée. La valeur *dist* évalue ainsi une distance Euclidienne « globale » entre chaque B-uplet supposé et la séquence reçue.

Algorithme séquentiel de reconstruction polynomiale II

- **Entrées de l'algorithme :**
  z, $\mathbf{c}^1, \ldots, \mathbf{c}^{N-K}$, t, $\hat{B}$, ν et $seuil(B)$
- Pour tout $\hat{B} \leq B \leq K$
  - **Pré-calculs :**
    - Pour chaque valeur de B, on détermine $seuil(B)$ qui va servir à discriminer les vecteurs.
    - Choisir ν vecteurs $\mathbf{s_1}, \ldots, \mathbf{s}_\nu$ de taille $(K-B)$. Pour chaque $\mathbf{s_i}$, chercher les combinaisons linéaires de $\mathbf{c}^i$ de la forme

$$\hat{\mathbf{c}}(i) = \sum_{j=0}^{t} \mathbf{c}^{w_j} = (\hat{c_1}, \ldots, \hat{c_B}, \mathbf{s_i})$$

      Pour tout i, on mémorise les $\hat{\mathbf{c}}(i) = \sum_{j=0}^{t} \mathbf{c}^{w_j}$, et les $\hat{z}(i) = \sum_{j=0}^{t} z_{w_j}$ associés, et on appelle S_i le cardinal de cet ensemble de couples.
  - **Coeur de l'algorithme :**
    1. Choisir un B-uplet $a_0, \ldots, a_{B-1}$ qui n'a pas été précédemment testé. Pour chaque $\mathbf{s_i}$, on parcourt l'ensemble des S_i couples $(\hat{\mathbf{c}(i)}, \hat{z}(i))$ et on évalue le nombre de fois que :

$$\sum_{j=1}^{B} a_j \hat{c}_j = \hat{z}_i$$

       avec $\hat{\mathbf{c}}(i) = (\hat{c_1}, \ldots, \hat{c_B}, \mathbf{s_i})$. On appelle num cette quantité.
       On met à jour la quantité $dist := dist + (S_i - 2\ num)^2$
    2. Si $dist > seuil(B)$ $(a_0, \ldots, a_{B-1},\ 0)$, $(a_0, \ldots, a_{B-1},\ 1)$ sont ajoutés à l'ensemble des candidats au $(B+1)$-préfixe. Si cet ensemble possède plus d'un élément, incrémenter B, retourner en 1.
       $dist := 0$
    3. En sortie, on a tous les vecteurs qui ont atteint la valeur K.

Tab. 2.11 – Deuxième algorithme de cryptanalyse utilisant la notion de i-préfixe, Johansson et Jönsson, Crypto 2000, [49].

Améliorations algorithmiques : Chose, Joux, Mitton [17]

L'article de Chose, Joux, Mitton publié dans Eurocrypt 2002, propose des améliorations algorithmiques majeures des algorithmes suscités : la recherche d'équations de parité ainsi que l'évaluation de la proportion d'équations vérifiées par un bit (requise pour les algorithmes [66], [72], [74], [49]) sont nettement optimisées.

L'algorithme de décodage proposé par les auteurs est alors le suivant : supposant avoir trouvé un ensemble d'équations que satisfait la séquence initiale (a), le nombre d'équations réellement satisfaites par chacun des bits de la séquence reçue (z) est évalué.
Considérons le i-ème bit :

- si le nombre d'équations qu'il satisfait est supérieur à un certain seuil, alors le bit est décodé en $a_i = z_i$;
- si le nombre d'équations qu'il satisfait est suffisamment faible, inférieur à un seuil, le bit est décodé en complémentant l'observation : $a_i = z_i + 1 \bmod 2$;
- sinon, le bit n'est pas décodé.

Une fois que suffisamment de bits linéairement indépendants ont pu être identifiés, l'état initial du système est déduit de l'inversion linéaire d'un système $(K \times K)$.

Nous décrivons brièvement l'algorithme de recherche d'équations d'une part, le calcul de la proportion d'équations vérifiées d'autre part.

• ***Recherche d'équations de parité :***
La recherche d'équations de parité est optimisée en fragmentant le problème de recherche d'équations de poids t en sous-calculs et en cherchant des collisions partielles. Soit a_i le bit pour lequel on souhaite trouver une équation. L'algorithme est écrit pour un cas plus général : celui de trouver des équations de la forme $P(a) = a_{i_1} + \ldots + a_{i_{t'}} + \sum_{j=0}^{B-1} c_j^m a_j$, où t' est pour l'instant le poids de l'équation. L'algorithme est alors détaillé dans le tableau 2.12.

Remarques : lorsque t est pair, $t' = t$ et $P(a) = 0$. Sinon $t' = t - 1$ et $P(a) = a_i$ un des bits sur lequel on cherche des équations de parité.
Par ailleurs, cet algorithme peut être utilisé plus généralement pour trouver des collisions entre colonnes sur $K - B$ positions, et il s'avère donc utile pour les algorithmes que nous avons décrits précédemment.

• ***Évaluation de la proportion d'équations vérifiées par un bit :***
L'utilisation d'une transformée de Fourier permet de réduire la complexité de cette étape.
On écrit B sous la forme $B = B_1 + B_2$, B_2 étant choisi de la façon suivante : $B_2 = log_2|\Omega|$. Les B_1 premiers bits sont tirés exhaustivement, et on appelle $\mathbf{X_1}$ ces B_1 premiers bits.

Considérons un bit z_i de la séquence reçue, et écrivons une des équations de parité du bit a_i qui lui correspond, sous la forme :

$$a_i = a_{m_1} + \ldots + a_{m_t} + \sum_{j=0}^{B_1-1} c_j^{m,i} a_j + \sum_{j=B_1}^{B_1+B_2-1} c_j^{m,i} a_j \tag{2.9}$$

Algorithme optimisé de recherche d'équations de parité

1. Ecrire $t' = t_1 + t_2 + t_3 + t_4$, t_i tels que $t_i = \lfloor t/4 \rfloor$ ou $t_i = \lceil t/4 \rceil$, et $t_1 \geq t_2$, $t_3 \geq t_4$
2. Calculer la somme de t_2 termes de la séquence (a) en fonction des K premiers bits de la séquence, et les ranger dans une table $U = \{u\}$
3. De la même façon, créer une table $V = \{v\}$ qui contient les sommes de t_4 bits de la séquence (a) exprimées à partir des K premiers.
4. $\forall \mathbf{s} \in [0; 2^S - 1]$ où S est de l'ordre de $t'/4 \log_2 N$, évaluer la somme $\mathbf{b}$ de t_1 bits avec $P(a)$ en fonction des K premiers bits, et chercher dans la table U un élément $\mathbf{u}$ tel que $(\mathbf{b} + \mathbf{u})_{/\mathcal{S}} = \mathbf{s}$. Mémoriser les collisions dans une table $\mathcal{C}$.
5. Procéder de la même façon en cherchant des collisions partielles sur s bits entre la somme $\mathbf{h}$ de t_3 bits et un élément $\mathbf{v}$ de V : $\mathbf{d} = \mathbf{h} + \mathbf{v}$, $(\mathbf{h} + \mathbf{v})_{/\mathcal{S}} = \mathbf{s}$. Chercher alors dans la table $\mathcal{C}$ précédemment construite un élément $\mathbf{c}$ qui coïncide sur ses $(K - B)$ derniers bits avec $\mathbf{d}$, *i.e.* tel que $(\mathbf{c} + \mathbf{d})_{/\ K-B} = \mathbf{0_{K-B}}$. On obtient alors une équation de la forme souhaitée.

TAB. 2.12 – Algorithme optimisé de recherche d'équations de parité, Chose, Joux et Mitton, Eurocrypt 2002, [17].

Posons

$$\Sigma^1_{m,i} = z_{m_1} + \ldots + z_{m_t} + \sum_{j=0}^{B_1-1} c_j^m a_j$$

et

$$\Sigma^2_{m,i} = \sum_{j=B_1}^{B_1+B_2-1} c_j^m a_j \ .$$

On va chercher à exploiter le fait que les équations du bit a_i peuvent avoir des supports non disjoints : on classe les équations de a_i en fonction de la valeur des B_2 derniers coefficients des équations du type de l'équation (2.9).
Soit

$$M_i(\mathbf{c_2}) = \{m \mid \forall j < B_2 \ \ c_j^m = c_{2,j}\},$$

où $\mathbf{c_2} = (c_{2,0}, \ldots, c_{2,B_2-1})$ est de longueur B_2. On définit la fonction

$$f_i(\mathbf{c_2}) = \sum_{m \in M_i(\mathbf{c_2})} (-1)^{\Sigma^1_{m,i}}$$

dont la transformée de Fourier évalue la différence entre le nombre de bits à 0 et le nombre de

bits à 1 étant donnée la séquence reçue (z) :

$$\begin{aligned}\widehat{f_i}(\mathbf{X_2}) &= \sum_{\mathbf{c_2}} f_i(\mathbf{c_2})(-1)^{\mathbf{c_2}\mathbf{X_2}} \\ &= \sum_{m}(-1)^{\Sigma^1_{m,i}+\Sigma^2_{m,i}}\end{aligned}$$

Or, la complexité du calcul de $\widehat{f_i}$ est en $\mathcal{O}(B_2 2^{B_2})$.

• ***Complexités :***

- la complexité calculatoire de la recherche des équations de parité de poids t est :

$$\mathcal{O}\Big(\min(DN^{\lceil (t-1)/2\rceil} logN, DN^{\lceil t/2\rceil} logN)\Big)$$

  et la complexité en mémoire est $\mathcal{O}(N^{\lfloor t/4\rfloor})$;
- de par l'utilisation de la transformée de Walsh, la complexité de l'évaluation de la proportion d'équations vérifiées est : $\mathcal{O}(D2^B log_2|\Omega|)$ au lieu de $\mathcal{O}(D2^B|\Omega|)$.

2.3 Tables comparatives de complexité

Les tables que nous présentons ci-après sont directement issues de [53]. Elles présentent, selon l'auteur, les résultats théoriques et obtenus par simulations sur un système constitué d'un générateur de pseudo-aléa ayant une mémoire $K = 40$, et associé à un polynôme générateur de la forme suivante : $g(x) = 1 + x + x^3 + x^5 + x^9 + x^{11} + x^{12} + x^{17} + x^{19} + x^{21} + x^{25} + x^{27} + x^{29} + x^{32} + x^{33} + x^{38} + x^{40}$.

Nous présentons essentiellement deux séries de résultats obtenus pour une même longueur de séquence, N, et une même valeur de t. Chaque série présente pour différentes valeurs de B, la probabilité de transition maximale et la complexité associée des algorithmes de Johansson-Jönsson [47] (algo I), de Chepyzhov-Johansson-Smeets [16] (algo II), et de Mihaljevic-Fossorier-Imaï[74] (algo III)

Première série de résultats

Cette première série de résultats a été obtenue pour $N = 400000$ et $t = 2$. Les résultats sont contenus dans le tableau 2.13. On constate que les algorithmes I et III sont nettement plus performants que l'algorithme II puisqu'ils permettent de s'attaquer à des systèmes dont la probabilité d'erreur est supérieure à 40%. Leur complexité est néanmoins plus élevée.

Deuxième série de résultats

Cette série est obtenue pour les paramètres suivants : $N = 40000$, $t = 3$. La « faible » longueur de séquence est donc compensée par l'augmentation du nombre d'équations puisque l'on s'autorise à sommer jusqu'à trois colonnes de la matrice génératrice. La probabilité de bruit associée est cependant plus élevée. Les résultats obtenus sont présentés dans le tableau 2.14.

On constate que dans dans la majeure partie des cas, pour une valeur de B fixée, ces paramètres permettent d'attaquer des système de probabilité de transition plus élevée que celle

B	**J-J Codes convolutifs**		**C-J-S Décodage ML**		**M-F-I Décodage par liste**	
	p	complexité	p	complexité	p	complexité
10	0.30	$2^{21,5}$	0.16	$2^{16,2}$	0.35	$2^{23,2}$
12	0.35	$2^{25,5}$	0.25	$2^{20,2}$	0.40	$2^{27,2}$
14	0.40	$2^{29,5}$	0.32	$2^{22,2}$	0.44	$2^{31,2}$

Tab. 2.13 – Résultats obtenus pour $N = 400000$ et $t = 2$.

B	**J-J Codes convolutifs**		**C-J-S Décodage ML**		**M-F-I Décodage par liste**	
	p	complexité	p	complexité	p	complexité
10	0.38	$2^{28,6}$	0.32	$2^{23,3}$	0.41	$2^{29,3}$
12	0.41	$2^{32,6}$	0.35	$2^{27,3}$	0.43	$2^{33,3}$
14	0.42	$2^{36,6}$	0.38	$2^{31,3}$	0.44	$2^{37,3}$

Tab. 2.14 – Résultats obtenus pour $N = 40000$ et $t = 3$.

simulée précédemment. D'autre part, l'algorithme III apparaît ici moins efficace, puisqu'il est de complexité plus élevée que dans le cas où $N = 400000$, $t = 2$, pour une même probabilité de transition maximale ($p = 0,44$). L'utilisation de cet algorithme sera donc dictée par la longueur de séquence reçue.

Algorithme de reconstruction polynomiale

On présente ici la complexité de l'attaque par reconstruction polynomiale, pour une valeur de probabilité fixée à $p = 0,40$, et pour $N = 400000$ et $t = 2$. Les résultats sont présentés dans le tableau 2.15. Si l'on compare les résultats obtenus avec ceux obtenus pour des paramètres identiques avec les algorithmes I et III, on constate que l'algorithme de reconstruction polynomiale est de complexité comparable à l'algorithme I mais qu'il est plus complexe que l'algorithme III, qui pour $B = 12$ permet d'atteindre une probabilité de $p = 0,44$ avec une complexité de $2^{27,2}$ au lieu de $2^{29,2}$.

Apports de l'algorithme de Chose-Joux-Mitton [17]

L'algorithme de Chose-Joux-Mitton permet de réduire la complexité des algorithmes sus-cités. Cette baisse de complexité se répartit de la façon suivante :

– en ce qui concerne la recherche des collisions sur $K - B$ coordonnées de t colonnes de la

B	**Complexité**
10	$2^{27,2}$
11	$2^{28,2}$
12	$2^{29,2}$

Tab. 2.15 – Reconstruction polynomiale, $p = 0,40$, $N = 400000$, $t = 2$.

matrice génératrice, si t est pair, la complexité est réduite d'un facteur D (où D est le nombre de bits que l'on cherche à décoder, soit $D \approx K$), et la mémoire d'un facteur au moins N si le paramètre t est supérieur ou égal à quatre ;
– en ce qui concerne la complexité d'évaluation du nombre d'équations de parité (cf. algorithme OSDA de Mihaljević-Fossorier-Imai, au paragraphe 2.2.2), la complexité, qui est à l'origine proportionnelle à $|\Omega|$, est maintenant de l'ordre de $log_2(|\Omega|)$ ($|\Omega|$ désignant, rappelons-le, le nombre moyen d'équations de parité par bit).

2.4 Conclusion

Ce chapitre présente plusieurs algorithmes de cryptanalyse, applicables au système du registre filtré. Nous avons tout d'abord évoqué les attaques de type « déterministe », qui tirent essentiellement profit de la mémoire intrinsèque au système et de la connaissance de la fonction. Nous avons observé que la complexité de ces attaques était essentiellement conditionnée par le choix des espacements entre les entrées de la fonction, choix qui détermine directement la mémoire du système total. Nous reviendrons au chapitre 4 sur une autre attaque de ce type.

Nous avons ensuite abordé le cas des attaques probabilistes qui, bien qu'écrites pour le cas de générateurs par combinaison de registres, s'appliquent à notre système qui en est un cas particulier (ou une généralisation selon le point de vue adopté !). Ces attaques modélisent la fonction booléenne comme un canal binaire symétrique sans mémoire et cherchent à se ramener à un problème de décodage. Elles requièrent donc la connaissance de contraintes qui caractérisent le mot de code bruité. Les contraintes utilisées sont des équations de parité vérifiées par les bits de la séquence issue du LFSR et c'est cette phase essentiellement qui conditionne la complexité des algorithmes probabilistes. D'une façon générale, plus le degré du polynôme de rétroaction est élevé, plus les équations de parité sont rares et délicates à trouver et plus les performances des algorithmes probabilistes sont mauvaises. L'autre facteur déterminant est la probabilité de transition du canal équivalent à la fonction, *i.e.* la probabilité d'erreur sur la séquence reçue en sortie de la fonction. La complexité de ces attaques dépend donc de facteurs très différents de ceux qui déterminent la complexité des attaques déterministes.

Le chapitre 3 introduit les algorithmes de décodage de type itératif, fréquemment utilisés. Ce chapitre présente également une extension de ces algorithmes au décodage d'une fonction booléenne. Le chapitre 4 présente une attaque de type déterministe utilisant un treillis, qui peut être étendue à une attaque probabiliste. Enfin, partant du constat que les attaques probabilistes ne prenaient pas en compte l'éventuelle structure d'un unique registre filtré, *i.e.* sa mémoire et la connaissance de la fonction booléenne, nous avons dérivé des attaques qui tirent profit de cette structure : dans le chapitre 5, nous présentons des algorithmes de cryptanalyse applicables au registre filtré, qui utilisent la connaissance intégrale du système pour améliorer les résultats classiquement obtenus.

Chapitre 3

Application du décodage itératif au registre filtré

W. Meier et O. Staffelbach ont, les premiers, cryptanalysé le système du registre filtré en utilisant un algorithme inspiré du décodage itératif de codes correcteurs [66]. Tous les algorithmes de ce type publiés depuis modélisent la fonction booléenne comme un canal binaire symétrique (BSC) sans mémoire, selon le modèle introduit dans les chapitres précédents (cf. chapitres 1 et 2) : la séquence produite par le LFSR est perçue comme un mot de code, régi par des contraintes linéaires qui sont imposées par la structure du générateur. La fonction booléenne se comporte comme une source de bruit introduisant une distorsion sur les bits de la séquence PN, dont résulte la séquence de sortie (z).
Si ce modèle s'avère justifié dans le cas d'un registre à combinaison (plusieurs registres éventuellement différents constituant chacun une entrée de la fonction booléenne, cf. figure 2.2 page 28), il l'est moins dans le cas d'un unique registre filtré qui est intrinsèquement sujet à un effet de mémoire dont le modèle du BSC fait totalement abstraction. Il nous est donc paru naturel de présenter quelques résultats de simulation sur le registre filtré et de comparer les performances d'algorithmes itératifs de cryptanalyse obtenues sur un vrai canal binaire symétrique d'une part, sur une fonction booléenne d'autre part (des comparaisons de natures différentes ont été effectuées dans les articles [18, 68, 71]). Dans un deuxième temps, nous avons appliqué le principe du décodage itératif à la table des valeurs d'une fonction booléenne, générant ainsi des probabilités sur les bits de clé qui permettent la cryptanalyse du système. Ce chapitre ne constitue en aucun cas une documentation exhaustive sur le sujet du décodage itératif : son seul objet est d'en présenter le principe général, d'étudier quelques performances sur notre système et de tenter de l'étendre au décodage d'une fonction booléenne. Le lecteur pourra se référer aux documents suivants pour avoir plus de détails : [3, 4, 29, 32, 60, 15, 35, 36, 45, 54, 62, 63, 78, 79, 92, 96, 97].

Nous allons dans un premier temps exposer l'algorithme de Gallager, ainsi que les codes pour lesquels il fut initialement développé. Dans le cas du registre filtré, cet algorithme présente souvent une complexité prohibitive : nous allons donc en présenter trois versions « simplifiées », ainsi qu'une version « discrète » dite du « Min-Sum » qui, travaillant non sur des probabilités mais sur des distances de Hamming, paraît adaptée à la cryptanalyse du système booléen déterministe qu'est celui du registre filtré. Suivront quelques résultats de simulation : nous étudierons des systèmes régis par un nombre de contraintes plus ou moins grand et comparerons les performances sur canal binaire symétrique à celles obtenues sur canal booléen. Nous présentons en dernier lieu un nouvel algorithme baptisé IDBF (Iterative Decoding of a Boolean Function), qui s'inspire du principe du décodage itératif pour cryptanalyser une fonction booléenne. Nous étudierons son

domaine d'application, l'utilisation qui peut en être faite et des résultats de simulation.

3.1 Algorithme de Gallager et quelques-unes de ses versions approchées

Nous allons décrire différentes méthodes de décodage itératif ; toutes peuvent être directement reliées à l'algorithme de Gallager suggéré par ce dernier en 1962 et initialement destiné au décodage des codes LDPC. Nous commencerons donc par une brève présentation de la structure de ces codes avant de décrire l'algorithme de Gallager. Dans le cas d'un registre filtré, le nombre d'équations de parité est souvent trop élevé pour que cet algorithme soit appliqué tel quel. Nous nous sommes donc intéressés à des versions approchées, moins complexes : nous présenterons trois d'entre elles. Nous exposerons ensuite le principe de l'algorithme Min-Sum, version « discrète » de l'algorithme de Gallager, puis terminerons cette partie par des résultats de simulations obtenus sur des canaux binaires symétriques et sur des fonctions booléennes.

Afin de clarifier la suite de ce chapitre, résumons les différents algorithmes exposés et les appellations que nous avons attribuées à chacun d'entre eux :

1. **L'algorithme de Gallager version** α : il s'agit de l'algorithme de Gallager dans sa version originale, sans approximation. [1]
2. **L'algorithme de Gallager version** β : il s'agit de l'algorithme de Gallager dans lequel le calcul des probabilités extrinsèques est approché par le minimum des probabilités *a priori* partielles.
3. **L'algorithme de Gallager version** γ : il s'agit de l'algorithme de Gallager dans lequel le calcul des extrinsèques est approché et les quantités rebouclées d'une itération à l'autre sont les probabilités *a posteriori* totales.
4. **L'algorithme de Gallager version** δ : il s'agit de l'algorithme de Gallager dans lequel le calcul des extrinsèques est exact, mais les quantités rebouclées d'une itération à l'autre sont les probabilités *a posteriori* totales.
5. **L'algorithme Min-Sum « discret »** : il s'agit d'un algorithme itératif qui calcule le coût associé à chacune des deux valeurs possibles que peut prendre un bit dans une équation et qui minimise le coût total.

Précisons qu'aucun de ces algorithmes n'est nouveau : le lecteur pourra en trouver la description dans la littérature (cf. les articles conseillés dans l'introduction de ce chapitre).

Les notions évoquées ci-dessus de probabilités « extrinsèque » et « *a posteriori* » seront, bien sûr, définies dans la suite de ce document ; cette liste est simplement donnée afin que le lecteur puisse s'y référer quand il le souhaite.

[1] Deux versions de l'algorithme de Gallager, la version A et la version B, sont fréquemment mentionnées dans la littérature ; l'algorithme B de Gallager coïncide en fait avec la version que nous avons qualifiée de version « α ». L'algorithme A de Gallager est quant à lui différent de toutes les versions présentées dans ce chapitre : cet algorithme ne s'applique que dans le cas du canal binaire symétrique. Pour chacun des bits de la séquence reçue, il évalue le nombre d'équations satisfaites ; si ce nombre est inférieur à un certain seuil, alors le bit est complémenté, et le processus est itéré. Cet algorithme est redécrit dans le chapitre 2 de ce document.

3.1.1 Quelques notions sur les codes LDPC : Low-Density Parity Check codes

Nous décrivons succinctement la structure des codes LDPC, auxquels le décodage itératif était initialement dédié. Pour plus de détails concernant les codes LDPC et leur décodage, nous invitons le lecteur à consulter entre autres les références suivantes [35, 36, 63, 78, 79, 92, 96, 97].

Les équations qui régissent les codes LDPC *(Low Density Parity Check)* ont la propriété d'avoir peu de coefficients : elles sont dites de « poids faibles », peu denses. Un code LDPC $\mathcal{C}(N, j, k)$ est un code linéaire de longueur N, dont la matrice de parité $\mathbf{H}$ vérifie les propriétés suivantes :

- chaque contrainte fait intervenir un même nombre de bits, k. Autrement dit, le nombre de '1' par ligne dans la matrice $\mathbf{H}$ est fixe, égal à k ;
- chaque bit appartient au même nombre de contraintes, j. Autrement dit, le nombre de '1' par colonne est fixe, égal à j.

En notant K la dimension du code, R le rendement de ce dernier, nous déduisons des deux assertions ci-dessus les relations suivantes :

$$k \times (N - K) = j \times N \tag{3.1}$$

$$R \geq (1 - \frac{j}{k}) \tag{3.2}$$

où la relation (3.2) est une égalité lorsque l'indépendance linéaire des lignes de la matrice $\mathbf{H}$ est vérifiée. Remarquons que si la dimension du code $\mathcal{C}(N, j, k)$ n'est pas mentionnée, c'est parce qu'elle se déduit immédiatement des paramètres N, j, k par la relation (3.2).

Exemple 4 *Un exemple de code LDPC :*

$$\mathbf{H} = \begin{pmatrix} 111111000000000000 \\ 000000111111000000 \\ 000000000000111111 \\ ---------- \\ 100100111000010000 \\ 010010000101001100 \\ 001001000010100011 \end{pmatrix}.$$

Il s'agit d'un code de longueur $N = 18$, de dimension $K = 6$ et de paramètres $k = 6$, $j = 2$: $\mathcal{C}(18, 2, 6)$. Cette matrice a été construite en suivant la méthode suggérée par Gallager : considérons la sous-matrice constituée des trois premières lignes. Chacune de ses lignes contient k '1' consécutifs (ici $k = 6$) ; ces k '1' sont étagés d'une ligne à l'autre. La matrice constituée des trois dernières lignes est déduite de la première par une simple permutation de ses colonnes.

Le procédé de construction d'un code LDPC décrit dans l'exemple ci-dessus est généralisable à des matrices ayant un plus grand nombre j de '1' par colonne : la première des j sous-matrices est construite en mettant k '1' par ligne et en étageant ces '1'. Les $(j - 1)$ matrices restantes sont déduites de la première par une permutation des colonnes. Cette permutation aléatoire des colonnes confère au code de bonnes propriétés asymptotiques (cf. [62]), puisque ce dernier tend vers une structure aléatoire (cf. chapitre 1). Remarquons que la présence de l'entrelaceur des turbo-codes (voir annexe E) n'est pas sans rappeler l'introduction d'aléa dans la conception de ce code.

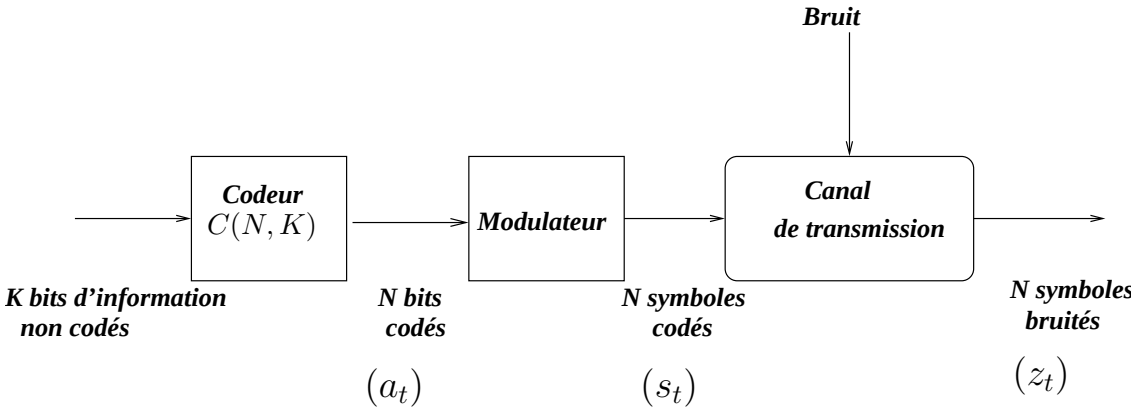


Fig. 3.1 – Modélisation de la chaîne d'émission dans le cadre d'une transmission d'information classique.

Donnons les propriétés essentielles des codes LDPC :

- les codes LDPC ayant deux '1' par colonne, *i.e.* tels que $j = 2$ ne sont jamais asymptotiquement bons (cf. chapitre 1) ;
- les codes LDPC $\mathcal{C}(N, j, k)$ sont presque toujours asymptotiquement bons dès que $j \geq 3$;
- lorsque j augmente, la distance minimale du code se rapproche de la borne de Gilbert-Varshamov (cf. chapitre 1).

3.1.2 Description de l'algorithme de décodage de Gallager : version α, Somme-Produit

L'algorithme de Gallager, également appelé « Belief Propagation » (B.P.) (cette dénomination est originellement associée aux réseaux de neurones, cf. [65]) fut publié pour la première fois en 1962 par Gallager [36]. Dans ce document, nous parlerons indifféremment d'« algorithme Belief Propagation » (B.P.) ou « algorithme de Gallager ».

Cet algorithme fut, à l'origine, développé pour décoder des codes LDPC, *Low Density Parity Check*. Nous décrivons l'algorithme de Gallager dans un cadre plus général : considérons un code $\mathcal{C}(N, K)$ caractérisé par sa matrice de parité $\mathbf{H} = (h_{i,j})_{i=0\ j=0}^{M-1\ N-1}$, où, rappelons-le, M désigne le nombre total d'équations de parité. Le nombre moyen d'équations de parité par bit sera, quant à lui, noté μ_b. Dans le cas d'un code classique, M est égal ou légèrement supérieur à $N - K$, mais dans le cas de la cryptanalyse des registres filtrés, M pourra être très supérieur à $N - K$ selon le système considéré. Dans cette présentation, nous adoptons une approche « turbo », dans la mesure où nous avons utilisé la même dénomination que celle introduite par C. Berrou et A. Glavieux dans l'article [4] (information extrinsèque et probabilité *a posteriori* partielle).

Comme illustré sur la figure 3.1, la séquence de symboles codés en entrée du canal sera notée $(s_t)_{t=0}^{N-1}$, la séquence reçue après passage dans le canal sera notée $(z_t)_{t=0}^{N-1}$.
Dans le cas d'un canal additif à bruit blanc gaussien (AWGN, *Additive White Gaussian Noise*), si la modulation utilisée est une BPSK *(Binary Phase Shift Keying)*, symboles et bits sont liés de la façon suivante : $s_t = (-1)^{a_t}$ où a_t est la sortie du codeur. Dans le cas d'un canal binaire symétrique, nous supposerons que la modulation est transparente : $s_t = a_t$. Nous parlerons donc indifféremment de symboles ou de bits.

La matrice de parité $\mathbf{H}$ est telle que

$$(a_0, \ldots, a_{N-1})\mathbf{H} = 0.$$

Considérons la m-ième équation de parité vérifiée par un mot du code $(a_0, \ldots, a_{N-1})$:

$$a_0 h_{m,0} + ... + a_{N-1} h_{m,N-1} = 0 \bmod 2.$$

Réécrivons cette équation de façon strictement équivalente sous la forme :

$$a_t + a_{t+\theta_{m,1}} + \ldots + a_{t+\theta_{m,d-1}} = 0 \bmod 2, \tag{3.3}$$

où

- $\forall m,\ \forall i,\ h_{m,i} \neq 0 \iff \exists\ j \leq d-1$ tel que $i = t + \theta_{m,j}$ ou bien $i = t$;
- d est le poids de l'équation, *i.e.* le nombre de termes qu'elle implique.

Introduisons quelques quantités supplémentaires :

1. À chaque instant t, soit Z_t la variable aléatoire en sortie du canal dont z_t constitue une réalisation.
2. $Obs(a_t) = P\Big(Z_t = z_t \mid a_t = 1\Big)$: c'est « **l'observation** » du bit a_t, l'information sur le bit a_t dont on dispose en sortie du canal. L'évaluation de cette quantité ne peut se faire sans utiliser les propriétés statistiques du canal. Donnons quelques exemples de canaux et les observations qui leur sont associées :
   - si le canal est un canal binaire symétrique, l'observation du bit a_t s'exprime en fonction de la probabilité de transition :

$$\begin{array}{rcll} Obs(a_t) & = & p & \text{si } z_t = 0 \\ Obs(a_t) & = & 1-p & \text{sinon ;} \end{array}$$

   - si le canal est un canal AWGN tel que le bruit est de variance N_0 et que la modulation utilisée est du type $s_t = (-1)^{a_t}$ (s_t étant le symbole émis), on aura

$$Obs(a_t) = \frac{1}{\sqrt{2\pi N_0}} e^{-\frac{(z_t+1)^2}{2N_0}}.$$

3. $APP(a_t) = P\Big(a_t = 1 \mid (Z_i)_{i=0}^{N-1} = (z_i)_{i=0}^{N-1}\Big)$: c'est la **probabilité *a posteriori*** du bit a_t, la probabilité qu'il soit égal à 1 connaissant la séquence reçue et en utilisant les contraintes qui régissent le mot de code. Cette quantité est celle que l'on cherche à évaluer afin de pouvoir décider de la valeur du bit a_t : si $APP(a_t) > 1/2$, le bit est décodé en '1', dans le cas contraire il est décodé en '0'.
4. $S_m(t)$: ensemble des indices de la m-ième équation de parité du t-ième bit : par exemple, dans l'équation (3.3), on a $S_m(t) = \{t, t+\theta_{m,1}, \ldots, t+\theta_{m,d-1}\}$.
5. $S'_m(t)$: ensemble des indices de la m-ième équation de parité du t-ième bit, t exclu : $S'_m(t) = S_m(t) \backslash \{t\}$, ce qui dans l'équation (3.3) correspond à $S'_m(t) = \{t+\theta_{m,1}, \ldots, t+\theta_{m,d-1}\}$.
6. $\mathcal{E}^b(t)$ est l'ensemble des équations binaires auxquelles le t-ième bit appartient.
7. Un élément de $\mathcal{E}^b(t)$ sera noté e et la m-ième équation du bit a_t, e_m.

Les définitions de probabilité données ci-dessus (et dans la suite de ce chapitre) se réfèrent toutes à l'évènement $\{a_t = 1\}$: ce choix est totalement arbitraire, nous aurions pu nous référer à l'évènement contraire et écrire les équations qui suivent en conséquence.
Considérons un bit a_t du mot de code. Ce bit intervient dans plusieurs équations de parité, et, d'après les notations qui précèdent, les indices des bits liés à a_t par une équation de parité appartiennent à l'ensemble $\bigcup_{m\in[1,\mu_b]} S'_m(t)$.

Les hypothèses faites sur le système sont les suivantes :
- H_0 : les bits a_t en sortie du codeur sont équiprobables ;
- H_1 : le canal de transmission est sans mémoire :

$$P\Big((Z_t)_0^{N-1} = (z_t)_0^{N-1} \mid (a_t)_0^{N-1}\Big) = \prod_{t=0}^{N-1} P\Big(Z_t = z_t \mid a_t\Big).$$

  Un symbole reçu ne dépend que du symbole émis qui lui correspond et est indépendant des symboles précédents émis et reçus ;
- H_2 : l'intersection des supports des équations de parité du bit a_t est réduite à a_t :

$$\bigcap_{m\ \in\ [1\ ;\mu_b]} S_m(t) = t$$

  qui peut s'écrire de façon équivalente

$$\bigcap_{m\ \in\ [1;\mu_b]} S'_m(t) = \emptyset\ .$$

L'algorithme de Gallager évalue la probabilité qu'un bit soit égal à 1 en fonction des autres bits du code en utilisant le lemme suivant, lequel exploite la linéarité des contraintes du code :

Lemme 1 *Soit $(x_1, \ldots, x_\ell) \in \{0,1\}^\ell$, ℓ bits statistiquement indépendants.*
Notons $P(x_t = 1) = p_t$ la probabilité que le t-ième bit soit égal à '1'.
Alors, la probabilité que le ℓ-uplet $(x_1, \ldots, x_\ell)$ contienne un nombre impair de 1 est donnée par :

$$\frac{1 - \prod_{i=1}^{\ell}(1-2p_i)}{2} \tag{3.4}$$

Preuve :
Le lecteur peut se reporter à la référence [36] afin d'y trouver le détail de cette preuve.

∎

Quel est le lien entre le lemme 1 et le décodage d'un code linéaire ?

La probabilité que le ℓ-uplet $(x_1, \ldots, x_\ell)$ contienne un nombre impair de '1' est aussi la probabilité que la somme de ses composantes, $\sum_{i=1}^{\ell} x_i$, soit égale à 1 modulo 2. Posons $\ell = d - 1$.

Supposons que nous ayons une connaissance *a priori* sur chacun des bits du mot de code, *i.e.* que nous disposions d'une information du type $P(a_t = 1) = p_t \neq 1/2 \ \ \forall t$. Considérons une équation

de parité $e_m \in \mathcal{E}^b(t)$. L'application du lemme 1 à cette équation permet d'évaluer une nouvelle probabilité sur a_t notée $P'(a_t = 1)$, en fonction des autres bits $(a_i)_{i \neq t} \in e_m$ de cette équation. En effet, considérons l'équation (3.3) et intéressons-nous au bit a_t :

$$a_t + \sum_{i=1}^{d-1} a_{t+\theta_{m,i}} = 0 \mod 2 \iff a_t = \sum_{i=1}^{d-1} a_{t+\theta_{m,i}} \mod 2 \tag{3.5}$$

$$\implies P'(a_t = 1) = P\Big(\sum_{i=1}^{d-1} a_{t+\theta_{m,i}} = 1\Big) \tag{3.6}$$

$$\implies P'(a_t = 1) = \frac{1 - \prod_{i=1}^{d-1}(1 - 2p_{t+\theta_{m,i}})}{2} \tag{3.7}$$

Chaque équation de parité $e_m \in \mathcal{E}^b(t)$ permet d'évaluer une probabilité sur a_t du type de celle de l'équation (3.7) ; chacune de ces probabilités est alors intégrée dans une probabilité globale qui tient compte de toutes les équations auxquelles a_t appartient. Cette probabilité totale sera d'autant plus fine, plus juste, que nous aurons utilisé toute l'information dont nous disposions.

Le traitement séquentiel et non conjoint de ces équations n'apparaît légitime que si les informations qu'elles génèrent sont statistiquement indépendantes. Nous supposerons dans la suite qu'elles le sont, ce qui, au premier ordre, est vérifié lorsque ces équations ont des supports d'intersection réduite à a_t, d'où la pertinence de l'hypothèse (H_2).

D'autre part, il apparaît naturel d'étendre un tel calcul à tous les bits du code, de calculer pour chacun d'entre eux une probabilité globale qui tiendra compte de toutes les équations auxquelles ils appartiennent. Une fois que ces « nouvelles » probabilités ont été calculées, la suite logique est d'itérer ce calcul, c'est-à-dire de réactualiser les probabilités en fonction de celles qui viennent d'être générées.

Développons l'expression de la probabilité *a posteriori* du bit a_t en fonction des quantités que nous avons définies précédemment. Rappelons que nous avons supposé que le canal était sans mémoire d'une part, que les informations apportées par chacune des équations $e_m \in \mathcal{E}^b(t)$ étaient statistiquement indépendantes d'autre part. Nous pouvons alors écrire :

$$\begin{aligned}
APP(a_t) &= P\Big(a_t = 1 \mid (Z_i)_0^{N-1} = (z_i)_0^{N-1}\Big) && (3.8)\\
&= P\Big(a_t = 1 \mid Z_t = z_t, (Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big) && (3.9)\\
&= P\Big(Z_t = z_t \mid a_t = 1, (Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big) \times \frac{P\Big(a_t = 1, (Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big)}{P(Z_t = z_t)} && (3.10)\\
&= P\Big(Z_t = z_t \mid a_t = 1\Big) \\
&\qquad \times \frac{P\Big(a_t = 1 \mid (Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big) \times P\Big((Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big)}{P(Z_t = z_t)} && (3.11)\\
&\propto Obs(a_t) \times P\Big(a_t = 1 \mid (Z_i)_{i \neq t} = (z_i)_{i \neq t}\Big)\ . && (3.12)
\end{aligned}$$

Détaillons les calculs ci-dessus :
– l'équation (3.9) est obtenue en isolant z_t ;

– l'équation (3.10) est obtenue en appliquant la définition de la probabilité conditionnelle à l'équation précédente : si A et B sont deux évènements, alors

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \ ; \tag{3.13}$$

– le premier terme de l'équation (3.11) est dû à l'absence de mémoire du canal, le second terme est déduit d'une nouvelle application de la définition de la probabilité conditionnelle ;
– la dernière équation (3.12) est simplement liée à l'apparition du signe proportionnel en lieu et place d'une égalité et à la définition de l'observation.

Examinons la dernière quantité : la probabilité totale que a_t soit égal à '1' est égale à la probabilité que a_t soit égal à '1' dans chacune de ses équations. Comme par hypothèse, ces équations sont indépendantes, la probabilité totale est égale au produit des probabilités. Par ailleurs, comme nous l'avons vu précédemment, la probabilité qu'un bit soit égal à '1' dans une de ses équations est égale à la probabilité que la somme des autres bits constituant cette équation soit égale à '1'. Ainsi

$$P\Big(a_t = 1 \mid (Z_i)_{i\neq t} = (z_i)_{i\neq t}\Big) = \prod_{m=1}^{\mu_b} P\left(\sum_{\ell \in S'_m(t)} a_\ell = 1 \mid (Z_i)_{i\neq t} = (z_i)_{i\neq t}\right), \tag{3.14}$$

et alors

$$APP(a_t) \propto Obs(a_t) \times \prod_{m=1}^{\mu_b} \underbrace{P\left(\sum_{\ell \in S'_m(t)} a_\ell = 1 \mid (Z_i)_{i\neq t} = (z_i)_{i\neq t}\right)}_{Extr_m(a_t) = \frac{1 - \prod\limits_{\ell \in S'_m(t)} \left(1 - 2P(a_\ell = 1 \mid (Z_i)_{i\neq t} = (z_i)_{i\neq t})\right)}{2}} . \tag{3.15}$$

Ces mêmes calculs nous conduisent à définir deux nouvelles quantités :

– **l'information extrinsèque** du bit a_t dans sa m-ième équation, figurée dans l'égalité (3.15) :

$$Extr_m(a_t) = P\left(\sum_{\ell \in S'_m(t)} a_\ell = 1 \;\Big|\; \forall i \in S'_m(t)\ Z_i = z_i,\ \forall j \notin S'_m(t)\ Z_j = z_i\right). \tag{3.16}$$

Cette information est qualifiée d'extrinsèque car c'est une information sur a_t évaluée à l'aide des **autres** bits de l'équation $e_m \in \mathcal{E}^b(t)$. On aura donc une probabilité extrinsèque par bit et par équation ;

– **la probabilité *a posteriori* partielle**

$$A_m(a_t) \propto Obs(a_t) \prod_{i,i\neq m} Extr_i(a_t), \tag{3.17}$$

qui est la probabilité *a posteriori* globale de a_t (cf. égalité (3.15) ci-dessus) privée d'une information extrinsèque, celle générée par la m-ième équation $e_m \in \mathcal{E}^b(t)$. De la même façon que pour l'information extrinsèque, on aura une APP partielle par bit et par équation. Lors des calculs, la probabilité partielle que le bit a_t soit égal à '1' sera normalisée par la somme de cette même probabilité avec la probabilité partielle que le bit soit égal à '0'.

L'algorithme de Gallager est exposé dans le tableau 3.1.

Les probabilités *a posteriori* partielles sont indexées par j, indice de l'itération en cours : par exemple, $Extr_m^j(a_t)$ désigne la probabilité extrinsèque associée au bit a_t dans sa m-ième équation à la j-ième itération. L'indice $j = 0$ correspond à la phase d'initialisation.
Détaillons les opérations effectuées dans l'algorithme de Gallager :

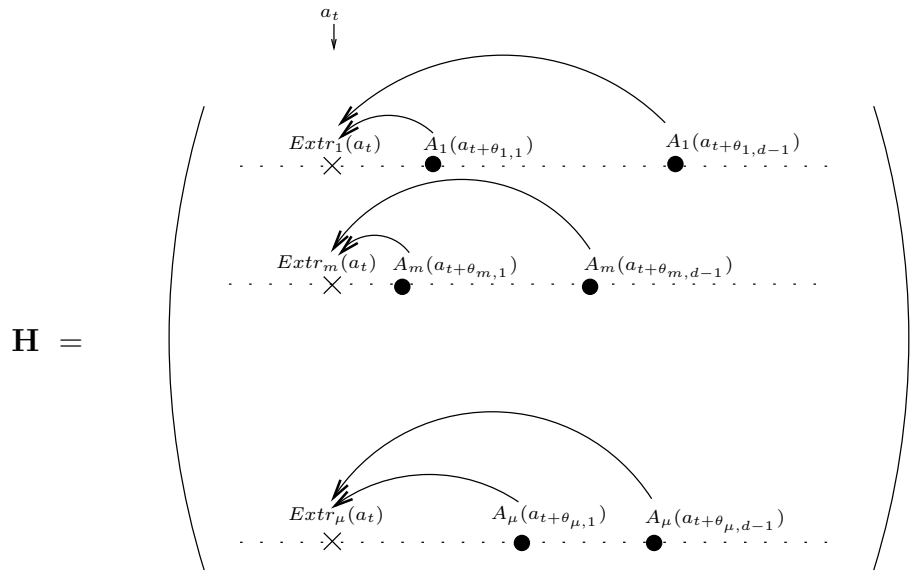
- à l'étape (1), phase d'initialisation, les informations extrinsèques sont initialisées à 1/2 puisque nous n'avons généré aucune information utilisant les équations et que nous ne possédons aucune information préalable (au sens probabiliste) dite information *a priori* sur les bits de la séquence ;
- l'algorithme démarre ensuite lors du calcul des probabilités *a posteriori* partielles : lors de la première itération, l'étape (2a) revient à imposer $\forall m,\ \forall t,\ A_m^{(1)}(a_t) = Obs(a_t)$. La valeur des observations étant directement liée aux propriétés du canal, le fonctionnement de l'algorithme et son initialisation vont être grandement influencés par ce dernier ;
- d'autre part, la formule (3.20) souligne l'intérêt d'avoir des équations de poids faible. En effet, cette équation s'écrit

$$\begin{aligned} Extr_m^{(j)}(a_t) &= \frac{1}{2}\left(1 - \prod_{i \in S'_m(t)} \left(1 - 2A_i^{(j)}(a_t)\right)\right) \\ &= \frac{1}{2} - \frac{1}{2} \prod_{i \in S'_m(t)} \left(1 - 2A_i^{(j)}(a_t)\right). \end{aligned}$$

  Or le produit $\prod_{i \in S'_m(t)} \left(1 - 2A_i^{(j)}(a_t)\right)$ implique $(d-1)$ termes inférieurs à 1 en valeur absolue : nous pouvons alors dire de façon approximative (puisque la valeur de ce produit dépend des probabilités *a posteriori*) que plus d est élevé, plus ce produit tend vers 0. Or,

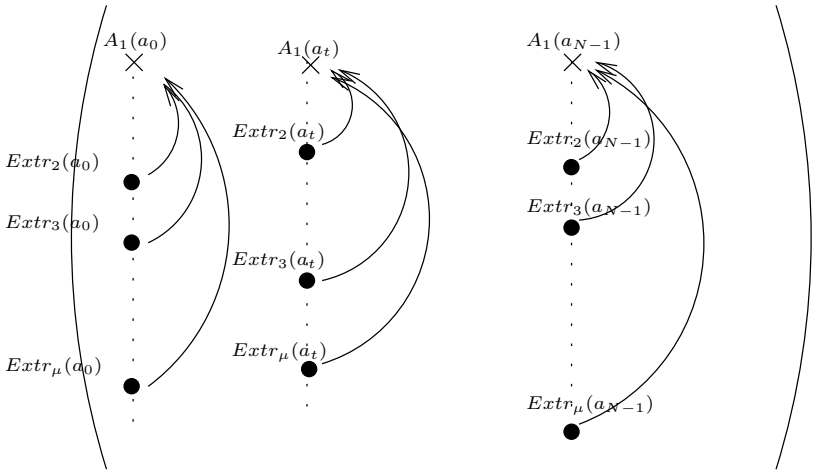
$$\begin{aligned} \prod_{i \in S'_m(t)} \left(1 - 2A_i^{(j)}(a_t)\right) \longrightarrow 0 \quad &\Longrightarrow \quad Extr_m^{(j)}(a_t) \longrightarrow \frac{1}{2} \\ &\Longrightarrow \quad A_m^{(j)}(a_t) \longrightarrow Obs(a_t) \end{aligned} \tag{3.18}$$

  et ainsi de suite : l'algorithme ne peut démarrer ;
- à une itération (j) donnée, les probabilités *a posteriori* partielles, $A_m^{(j)}$ sont évaluées à partir des probabilités extrinsèques calculées à l'itération précédente $(j-1)$. Les informations extrinsèques sont ensuite réactualisées en utilisant les probabilités *a posteriori* partielles que l'on vient de générer. Le coefficient de proportionnalité apparaissant aux étapes (2a) et (3) provient du fait que nous avons omis les étapes de normalisation des probabilités ; les quantités *a posteriori* évaluées aux étapes (2a) et (3) doivent cependant faire l'objet d'une normalisation, afin de conserver leur « statut » de probabilités ;
- si nous revenons à la représentation du code sous la forme d'une matrice de parité, nous constatons que l'algorithme de Gallager fait alternativement des mouvements horizontaux et verticaux dans la matrice : les mouvements horizontaux évaluent les extrinsèques, les mouvements verticaux calculent les APP (partielles et globales). Nous illustrons notre propos sur les figures 3.2 et 3.3 avec un code LDPC $(N, 4, 3)$: chaque équation fait intervenir trois bits, chaque bit appartient à quatre équations.



H =

Fig. 3.2 – Mouvements horizontaux de l'algorithme de Gallager dans la matrice de parité du code lors du calcul des probabilités extrinsèques.



H =

Fig. 3.3 – Mouvements verticaux de l'algorithme de Gallager dans la matrice de parité du code lors du calcul des probabilités *a posteriori*.

Algorithme de Gallager (version α)

1. ***Initialisation :***
   – initialiser les quantités $Obs(a_t)$, $\forall t$ conformément aux propriétés du canal (voir section 3.1.2). En particulier, si le canal de transmission est un canal binaire symétrique,

$$\begin{aligned} Obs(a_t) &= 1-p \quad \text{si } z_t = 1 \\ Obs(a_t) &= p \quad \text{si } z_t = 0 \ ; \end{aligned}$$

   – initialiser les informations extrinsèques :

$$Extr_m^{(0)}(a_t) = \frac{1}{2} \ .$$

2. ***Déroulement :***
   à l'itération $j \geq 1$: $\forall t$, $\forall m$, calculer
   (a)

$$A_m^{(j)}(a_t) \propto Obs(a_t) \prod_{i, i \neq m} Extr_i^{(j-1)}(a_t) \ , \tag{3.19}$$

   (b)

$$Extr_m^{(j)}(a_t) = \frac{1}{2} \left(1 - \prod_{i \in S'_m(t)} \left(1 - 2A_i^{(j)}(a_t) \right) \right) \ . \tag{3.20}$$

3. ***Terminaison :***
   à l'itération nb_{iter} , calcul de l'APP totale de chacun des bits

$$APP(a_t) \propto Obs(a_t) \prod_m Extr_m^{(nb_{iter})}(a_t)$$

$$\begin{aligned} APP(a_t) > \frac{1}{2} &\implies a_t \text{ est décodé en '1'}, \\ APP(a_t) < \frac{1}{2} &\implies a_t \text{ est décodé en '0'} \ . \end{aligned}$$

Tab. 3.1 – Algorithme de Gallager, version α.

Remarquons que la probabilité extrinsèque du bit a_t dans l'équation m, $Extr_m(a_t)$, se calcule à partir des APP partielles des autres bits qui constituent l'équation m, *i.e.* ceux dont l'indice appartient à l'ensemble $S'_m(t)$. Or, comme le montre l'équation de l'étape (2a), ces APP partielles sont calculées à partir de l'information disponible sur ces bits dans toutes les autres équations **exceptée** l'équation m. Cela est directement lié à une représentation du code sous la forme d'un graphe, appelé graphe de Tanner, comme illustré sur la figure 3.4. Les noeuds « carrés » représentent les équations de parité, les noeuds ronds représentent les bits. Le processus de décodage apparaît comme une propagation ascendante de l'information le long de l'arbre. La base de l'arbre contient les noeuds bits dont la probabilité est initialement fixée à la valeur de l'observation (phase d'initialisation de l'algorithme). Cette information est alors propagée vers les équations de parité afin d'évaluer les probabilités extrinsèques ; ces dernières sont ensuite transmises aux noeuds « bits » pour évaluer les probabilités *a posteriori* et le procédé est itéré. La propagation de l'information telle qu'elle vient d'être décrite souligne la **pertinence du caractère itératif de l'algorithme** : plus on itère, plus les probabilités calculées tiennent compte de la structure globale du code. L'algorithme tel qu'il a été présenté est exact si le code a une structure strictement arborescente, ce qui ne sera pas le cas en pratique : l'équation (3.15) (cf. page 66) a été obtenue en faisant l'hypothèse que les informations apportées par chacune des équations sont indépendantes, c'est-à-dire que le code est complètement arborescent, sans cycle. En pratique, la longueur du code est finie et puisque chaque bit appartient à plusieurs équations de poids supérieur à deux, il est inévitable que des cycles apparaissent au cours des itérations.

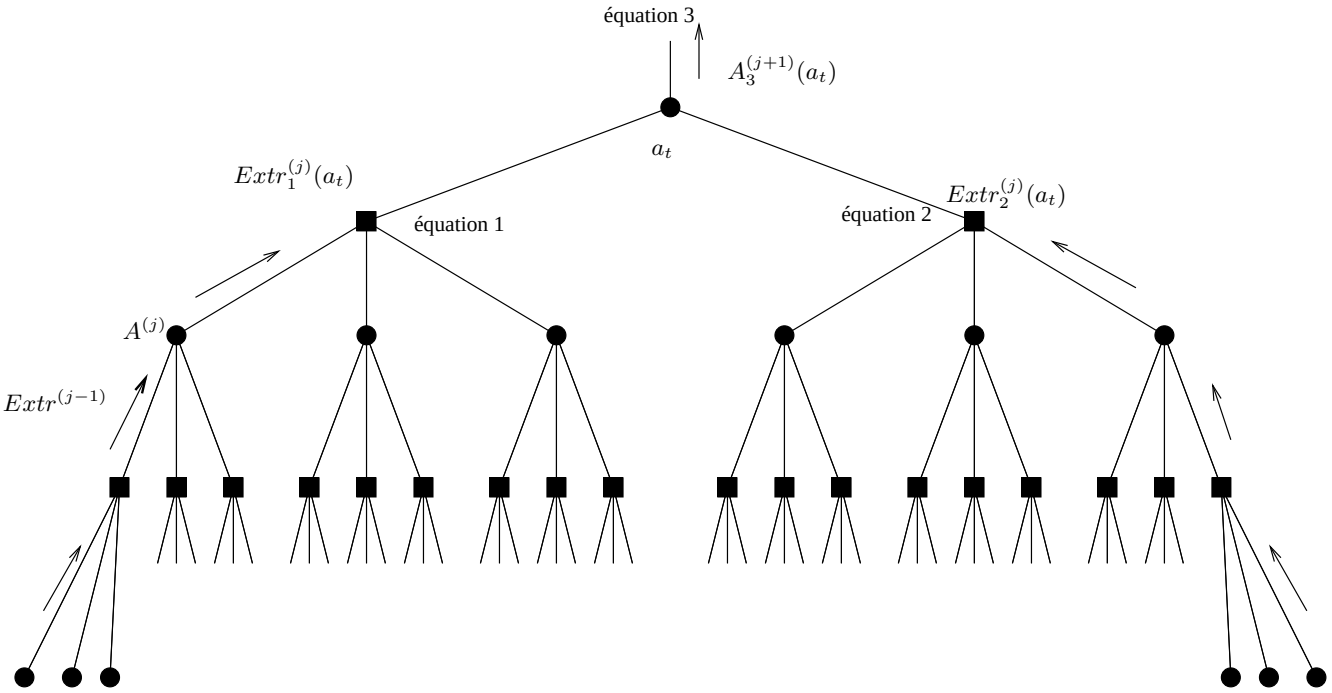


Fig. 3.4 – Propagation ascendante de l'information le long du graphe de Tanner du code.

La décision finale se fait sur la probabilité *a posteriori* totale : si elle est supérieure à $\frac{1}{2}$, le bit décodé est un 1, autrement c'est un 0.

Comme nous l'avons mentionné plus haut, ces probabilités sont calculées à un coefficient de

proportionnalité près et nécessitent donc une normalisation à chaque calcul. Une façon d'éviter ces normalisations est de manipuler des quantités logarithmiques, couramment utilisées car elles remplacent les multiplications par des additions, et parce qu'elles donnent lieu à des approximations qui facilitent les calculs. Avant d'entrer plus en détail dans les approximations qui peuvent être faites, nous réécrivons l'algorithme en utilisant des logarithmes. Nous définissons l'application suivante :

Définition : La fonction de « **log-vraisemblance** » (ou log-likelihood), $\mathcal{L}$, est définie de la façon suivante :

$$\begin{array}{rcl} \mathcal{L} :]0;1[& \longrightarrow &]-\infty;+\infty[\\ x & \longrightarrow & \log \dfrac{x}{1-x} \end{array} \tag{3.21}$$

Par exemple $\mathcal{L}[Obs(a_t)] = \log \frac{Obs(a_t)}{1-Obs(a_t)} = \log \frac{P(Z_t=z_t \mid a_t=1)}{1-P(Z_t=z_t \mid a_t=1)}$ ce qui, dans le cas d'un canal binaire symétrique de probabilité de transition p, correspond à : $\mathcal{L}[Obs(a_t)] = \log \frac{1-p}{p}$ lorsque le bit reçu est un '1'.

Le tableau 3.2 donne l'écriture de l'algorithme de Gallager en utilisant la fonction de log-likelihood.

Cette version est dite du « Somme-Produit » (cf. [54] par exemple) en raison des opérations qui sont effectuées : les extrinsèques sont sommées lors du calcul d'une APP partielle, les $\Lambda(i) = \frac{1-e^{\mathcal{L}[A_m(a_i)]}}{1+e^{\mathcal{L}[A_m(a_i)]}}$ sont multipliées lors du calcul d'une extrinsèque.
La version non logarithmique décrite dans le tableau 3.1 est strictement équivalente à celle-ci ; elle est donc également qualifiée de « Somme-Produit ».

Étude de la complexité :

– *mémoire* : il nous faut stocker N observations et $2N\mu_b$ probabilités (extrinsèques et APP partielles). D'où une complexité en mémoire en

$$\mathcal{O}\left(N\mu_b\right) \ ; \tag{3.22}$$

– *calculs* : le calcul d'une APP partielle nécessite μ_b opérations, celui d'une information extrinsèque environ d opérations. Le coeur de l'algorithme comporte nb_{iter} itérations ; la complexité calculatoire qui lui est associée est donc :

$$nb_{iter}\ N\mu_b(\mu_b + d).$$

Le calcul de l'APP finale nécessite $N\mu_b$ opérations. D'où une complexité en calculs totale de :

$$\mathcal{O}\Big(nb_{iter}\ N\mu_b(\mu_b + d)\Big) \tag{3.23}$$

où, précisons-le, l'opération de multiplication a été comptabilisée comme une opération élémentaire.

Algorithme de Gallager (version α) sous forme logarithmique :

algorithme « Somme-Produit »

1. ***Initialisation :***
   - $\mathcal{L}[Extr_m^{(0)}(a_t)] = 0$
   - $\mathcal{L}[Obs(a_t)] = \log \frac{1-p}{p}$ si $z_t = 1$
     $\mathcal{L}[Obs(a_t)] = \log \frac{p}{1-p}$ si $z_t = 0$.
2. ***Déroulement :***
   à l'itération j,$\forall m, \forall t$
   (a) $\mathcal{L}[A_m^{(j)}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_{i,i\neq m} \mathcal{L}[Extr_i^{(j-1)}(a_t)]$
   (b) $\mathcal{L}[Extr_m^{(j)}(a_t)] = \log \frac{1- \prod_{i\in S'_m(t)} \Lambda^{(j)}(i)}{1+ \prod_{i\in S'_m(t)} \Lambda^{(j)}(i)}$
   avec $\Lambda^{(j)}(i) = \frac{1-e^{\mathcal{L}[A_m^{(j)}(a_i)]}}{1+e^{\mathcal{L}[A_m^{(j)}(a_i)]}}$.
3. ***Terminaison :***
   à l'itération nb_{iter}
   $\mathcal{L}[APP(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_m \mathcal{L}[Extr_m^{(nb_{iter})}(a_t)]$.

$$\mathcal{L}[APP(a_t)] > 0 \implies a_t \text{ est décodé en '1'},$$
$$\mathcal{L}[APP(a_t)] < 0 \implies a_t \text{ est décodé en '0'}.$$

Tab. 3.2 – Algorithme de Gallager version α, écriture logarithmique.

Le lecteur pourrait être surpris par la complexité calculatoire énoncée ci-dessus. En effet, une objection naturelle à ce résultat serait la suivante : la probabilité *a posteriori* partielle diffère de la probabilité *a posteriori* totale d'une seule valeur extrinsèque. Alors, pour un bit donné a_t, pourquoi effectuer μ_b produits impliquant $(\mu_b - 1)$ termes pour calculer chacune des probabilités partielles au lieu de calculer une fois la probabilité *a posteriori* totale (μ_b opérations), mémoriser cette quantité, et la diviser successivement par une quantité extrinsèque (une opération) pour obtenir chaque probabilité partielle ? La complexité du calcul des probabilités *a posteriori* partielles serait alors divisée par μ_b. Le calcul des extrinsèques pourrait être presque de la même façon optimisé, et la complexité de chacun de ces calculs serait d fois inférieure à celle présentée ci-dessus.
Ces méthodes de calculs présentent cependant un inconvénient majeur : les quantités par lesquelles on divise peuvent être arbitrairement faibles en théorie. Diviser par une quantité proche de zéro comporte des risques, la précision des ordinateurs étant limitée.
Nous avons donc fait le choix d'énoncer la complexité de l'algorithme de Gallager (et des versions

dérivées qui suivent) en tenant compte de cette remarque.

L'algorithme du paragraphe 3.1.3 qui suit résulte d'une approximation de calculs faite sur les quantités logarithmiques introduites dans ce paragraphe.

3.1.3 Une première simplification de l'algorithme de Gallager : version β (Min-Sum souple)

Une première approximation possible (voir [45] pour plus de détails) est de remplacer le produit $\prod \Lambda(i)$ par le minimum des termes en valeur absolue, pondéré par le signe du produit :

$$\prod_{i \in S'_m(t)} \Lambda(i) \approx \text{sgn}\left(\prod_{i \in S'_m(t)} \Lambda(i) \right) \times \min_{i \in S'_m(t)} |\Lambda(i)| \,,$$

ce qui donne lieu à la simplification suivante :

$$\mathcal{L}[Extr_m(a_t)] = -\text{sgn}\left(\prod_{i \in S'_m(t)} \mathcal{L}[A_m(a_i)] \right) \times \min_{i \in S'_m(t)} \Big| \mathcal{L}[A_m(a_i)] \Big|.$$

Les deux étapes d'initialisation et de terminaison de l'algorithme sont inchangées. L'algorithme complet est donné dans le tableau 3.3. Cette version de l'algorithme B.P. est souvent dite du « Min-Sum » en raison des opérations effectuées dans la boucle principale. Au contraire, l'algorithme B.P. d'origine est dit du « Somme-Produit ».

Étude de la complexité :

- *mémoire* : il nous faut stocker N observations et $2N\mu_b$ probabilités (extrinsèques + APP partielles) d'où une complexité en mémoire de l'ordre de

$$\mathcal{O}\left(N\mu_b\right) \ ; \tag{3.24}$$

- *calculs* : le calcul d'une APP partielle nécessite μ_b opérations, celui d'une information extrinsèque est réduit à l'évaluation d'un minimum, qui nécessite l'examen de $d-1$ termes. Le coeur de l'algorithme, qui comporte nb_{iter} itérations nécessite donc :

$$nb_{iter} N\mu_b(\mu_b + d - 1) \text{ calculs.}$$

  Le calcul de l'APP finale nécessite $N(\mu_b + 1)$opérations. D'où une complexité en calculs totale de l'ordre de :

$$\mathcal{O}\Big(nb_{iter} \ N\mu_b(\mu_b + d)\Big). \tag{3.25}$$

Notons toutefois que la complexité de cet algorithme a été évaluée en considérant que, lors du calcul du minimum, l'instruction d'examen d'un terme coûte une opération élémentaire. Lors du calcul de complexité effectué pour la version α, nous avions considéré que la multiplication de deux nombres réels étaient également une opération élémentaire. La différence entre le calcul

Algorithme de Gallager approché (version β)
Min-Sum souple

1. ***Initialisation :***
   - $\mathcal{L}[Extr_m^{(0)}(a_t)] = 0$
   - $\mathcal{L}[Obs(a_t)] = \log \frac{1-p}{p}$ si $z_t = 1$
     $\mathcal{L}[Obs(a_t)] = \log \frac{p}{1-p}$ si $z_t = 0$.
2. ***Déroulement :*** à l'itération j, $\forall m,\ \forall t$,
   (a) $\mathcal{L}[A_m^{(j)}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_{i,i\neq m} \mathcal{L}[Extr_i^{(j-1)}(a_t)]$

   (b) $\mathcal{L}[Extr_m^{(j)}(a_t)] = -\text{sgn}\left(\prod_{i\in S'_m(t)} \mathcal{L}[A_m^{(j)}(a_i)]\right) \times \min_{i\in S'_m(t)} \left|\mathcal{L}[A_m^{(j)}(a_i)]\right|$.
3. ***Terminaison :*** :
   à l'itération nb_{iter}
   $\mathcal{L}[APP(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_m \mathcal{L}[Extr_m^{(nb_{iter})}(a_t)]$

$$\mathcal{L}[APP(a_t)] > 0 \implies a_t \text{ est décodé en '1'},$$
$$\mathcal{L}[APP(a_t)] < 0 \implies a_t \text{ est décodé en '0'}.$$

Tab. 3.3 – Algorithme de Gallager version β, Min-Sum souple.

du minimum d'un ensemble de d termes et la multiplication de d nombres réels se résume à un facteur multiplicatif en terme de complexité, donc l'évaluation de la complexité ci-dessus est correcte. Toutefois, la version α de l'algorithme B.P. sera beaucoup plus lente que la version approchée qui calcule le minimum, en raison de l'importance de la mémoire et du temps C.P.U. nécessaires au processeur pour effectuer la multiplication de deux nombres réels.

3.1.4 Une seconde simplification de l'algorithme de Gallager : version γ

Au lieu d'itérer les calculs sur les extrinsèques et les probabilités *a posteriori* partielles, une solution moins coûteuse est d'itérer sur les probabilités *a posteriori* globales : l'algorithme résultant est plus économe en terme de calculs et en terme de mémoire. Il s'agit néanmoins d'une version approchée de l'algorithme exact, ce qui risque d'avoir des conséquences néfastes en terme de performances.
Cet algorithme, baptisé « version γ de l'algorithme de Gallager », est présenté dans le tableau 3.4. Ce tableau ne figure des probabilités extrinsèques que pour faire le parallèle avec les algorithmes décrits précédemment. En pratique, ces quantités sont omises et les itérations se font directement sur les probabilités *a posteriori* globales.

Algorithme de Gallager approché : version γ

1. ***Initialisation :***
   $\mathcal{L}[Obs(a_t)] = \log \frac{1-p}{p}$ si $z_t = 1$
   $\mathcal{L}[Obs(a_t)] = \log \frac{p}{1-p}$ sinon.
2. ***Déroulement*** :
   $\forall j \geq 1,\ \forall t,$
   (a) $\mathcal{L}[APP^{(j)}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_i \mathcal{L}[Extr_i^{(j-1)}(a_t)]$
   (b)

$$\forall\ m\ \ \mathcal{L}[Extr_m^{(j)}(a_t)] = -\text{sgn}\left(\prod_{i\in S'_m(t)} \mathcal{L}[APP^{(j)}(a_i)]\right) \times \min_{i\in S'_m(t)} \left|\mathcal{L}[APP^{(j)}(a_i)]\right|.$$

3. ***Terminaison :***
   à l'itération nb_{iter}
   $\mathcal{L}[APP^{(nb_{iter})}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_i \mathcal{L}[Extr_i^{nb_{iter}}(a_t)]$

$$\mathcal{L}[APP^{(nb_{iter})}(a_t)] > 0 \implies a_t \text{ est décodé en '1'},$$
$$\mathcal{L}[APP^{(nb_{iter})}(a_t)] < 0 \implies a_t \text{ est décodé en '0'}.$$

Tab. 3.4 – Algorithme de Gallager version γ.

Étude de la complexité :

– *mémoire* : il nous faut stocker N observations, N APP de l'itération $j-1$ et les N APP de l'itération j, soit $3N$ probabilités d'où une complexité en

$$\mathcal{O}(N)\ ; \tag{3.24}$$

– *calculs* : le calcul d'une APP nécessite $\mu_b d$ opérations. Le coeur de l'algorithme, qui comporte nb_{iter} itérations nécessite donc :

$$\mathcal{O}\left(nb_{iter}\ N\mu_b d\right)\ \text{opérations.} \tag{3.25}$$

3.1.5 Une troisième simplification de l'algorithme de Gallager : version δ

Cette simplification s'inspire de la précédente : au lieu d'itérer les calculs sur les APP partielles, nous allons itérer sur les APP totales, ce qui évite de conserver en mémoire toutes les APP partielles d'une itération à l'autre. Néanmoins, nous n'allons pas effectuer l'approximation

utilisée dans la version γ : nous allons calculer la valeur réelle de l'extrinsèque dans chacune des équations.
L'algorithme est alors donné dans le tableau 3.5.

Algorithme de Gallager approché (version δ)

1. ***Initialisation :***
   $\mathcal{L}[Obs(a_t)] = \log \frac{1-p}{p}$ si $z_t = 1$
   $\mathcal{L}[Obs(a_t)] = \log \frac{p}{1-p}$ sinon.
2. ***Déroulement :***
   à l'itération j, $\forall m,\ \forall t$
   (a) $\mathcal{L}[APP^{(j)}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_i \mathcal{L}[Extr_i^{(j-1)}(a_t)]$
   (b) $\forall m\ \mathcal{L}[Extr_m^{(j)}(a_t)] = \log \frac{1- \prod_{i \in S'_m(t)} \Lambda(i)}{1+ \prod_{i \in S'_m(t)} \Lambda(i)}$ avec $\Lambda^{(j)}(i) = \frac{1-e^{\mathcal{L}[APP^{(j)}(a_i)]}}{1+e^{\mathcal{L}[APP^{(j)}(a_i)]}}$.
3. ***Terminaison :***
   à l'itération nb_{iter}
   $\mathcal{L}[APP^{(nb_{iter})}(a_t)] = \mathcal{L}[Obs(a_t)] + \sum_i \mathcal{L}[Extr_i^{nb_{iter}}(a_t)]$

$$\mathcal{L}[APP^{(nb_{iter})}(a_t)] > 0 \implies a_t \text{ est décodé en '1',}$$
$$\mathcal{L}[APP^{(nb_{iter})}(a_t)] < 0 \implies a_t \text{ est décodé en '0'.}$$

Tab. 3.5 – Algorithme de Gallager version δ.

Étude de la complexité :

– *mémoire* : il nous faut stocker $3N$ probabilités, d'où une complexité en mémoire de

$$\mathcal{O}(N)\ ; \tag{3.26}$$

– *calculs* : le calcul d'une APP nécessite $(\mu_b + 1)d$ opérations.
L'algorithme nécessite donc :

$$\mathcal{O}(nb_{iter}\ N\mu_b d) \text{ calculs.} \tag{3.27}$$

3.1.6 Algorithme du Min-Sum « discret »

Nous décrivons brièvement l'algorithme du Min-Sum « discret » : contrairement aux algorithmes précédemment exposés, cet algorithme ne manipule pas de probabilités mais travaille sur

des distances de Hamming d'où sa qualification de « discret ». Ce Min-Sum évalue le coût que chacun des bits du code soit égal à '1' ou à '0', en comptant le nombre de bits à complémenter pour que cela soit vérifié dans toutes les équations auxquelles il appartient. Le lecteur pourra trouver une description approfondie extrêmement claire dans [96].
Pour chaque bit de la séquence à décoder, nous sommes amenés à calculer deux quantités, chacune d'entre elles étant relative au nombre de complémentations qu'implique le bit mis à 0 ou mis à 1. Nous allons dorénavant utiliser des quantités vectorielles. Posons

$$\begin{aligned}
\boldsymbol{\mathcal{L}}[Obs(a_t)] &= [d_H(z_t,0), d_H(z_t,1)], \\
\boldsymbol{\mathcal{L}}[Extr_m(a_t)] &= [Extr_m^0(a_t), Extr_m^1(a_t)], \\
\boldsymbol{\mathcal{L}}[A_m(a_t)] &= [A_m^0(a_t), A_m^1(a_t)], \\
\boldsymbol{\mathcal{L}}[APP(a_t)] &= [APP^0(a_t), APP^1(a_t)].
\end{aligned}$$

Chacune des composantes est indexée par '0' ou '1' suivant la valeur du bit auquel elle se réfère. L'algorithme est alors présenté dans le cadre 3.6. Nous constatons que l'écriture de cet algorithme est très proche de l'écriture logarithmique des algorithmes B.P.. La différence essentielle réside à l'étape 2b pour le calcul de $\boldsymbol{\mathcal{L}}[Extr_m(a_t)]$: cette étape consiste en fait à minimiser le coût relatif à chacune des deux valeurs possibles de a_t sur l'ensemble des configurations possibles compatibles avec l'équation de parité considérée m. Cette étape revient à minimiser le nombre de coordonnées à changer dans chacune des équations auxquelles a_t appartient pour que $a_t = 0$ et $a_t = 1$.

Étude de la complexité :

- *mémoire* : il nous faut stocker $N + 2N\mu_b$ probabilités (généralement, on mémorise la différence des deux composantes de chacun des vecteurs). D'où une complexité en mémoire de l'ordre de

$$\mathcal{O}(N\mu_b) \; ; \tag{3.28}$$

- *calculs* : le calcul d'une APP partielle nécessite $N(\mu_b - 1)$ opérations, celui d'une extrinsèque $2^{d-2}(d-1)$ opérations. D'où une complexité globale de calculs en

$$\mathcal{O}\left(nb_{iter}\ N\mu_b(\mu_b + d2^d)\right). \tag{3.29}$$

3.1.7 Comparaison des complexités des différentes versions de l'algorithme de Gallager

Dans la plupart des cas que nous traiterons, le poids d des équations sera relativement faible ($d \leq 6$), alors que le nombre moyen d'équations de parité par bit pourra être très élevé, tout comme la longueur N du code. Sous ces hypothèses, le tableau 3.7 résume les complexités des différentes versions de l'algorithme B.P..

Notons cependant que nous avons considéré que les opérations effectuées dans l'algorithme du Min-Sum « discret » étaient aussi coûteuses que celles des versions de l'algorithme Belief Propagation. Ceci est inexact : les algorithmes B.P. (versions α, β, γ ou δ) effectuent des opérations sur des nombres réels, puisqu'ils évaluent des probabilités. Or, ces opérations nécessitent davantage de mémoire et de temps de calcul que celles effectuées par l'algorithme Min-Sum, qui, lui, travaille sur des entiers. Ceci n'est pas sans rappeler la remarque faite au sujet du calcul du minimum dans les version β et γ de l'algorithme B.P. (cf. paragraphe 3.1.3 de ce chapitre).

Algorithme de Gallager « discret » : le Min-Sum

1. ***Initialisation :***
   $\boldsymbol{\mathcal{L}}[Obs(a_t)] = [d_H(z_t,0), d_H(z_t,1)]$
   $\boldsymbol{\mathcal{L}}[A_m(a_t)] = [0,0]$
   $\boldsymbol{\mathcal{L}}[Extr_m(a_t)] = \boldsymbol{\mathcal{L}}[Obs(a_t)]$.

2. ***Déroulement :***
   à l'itération j, $\forall m,\ \forall t$,
   (a) $\boldsymbol{\mathcal{L}}[A_m^{(j)}(a_t)] = \boldsymbol{\mathcal{L}}[Obs(a_t)] + \sum_{i\neq m} \boldsymbol{\mathcal{L}}[Extr_i^{(j-1)}(a_t)]$
   (b) $\boldsymbol{\mathcal{L}}[Extr_m^{(j)}(a_t)] =$
   $[\ \min_{\mathbf{b}/\sum_{\ell\in S'_m(t)} b_\ell=0} \sum_{i\in S'_m(t)} A_m^{(j),b_i}(a_i)\ ,$
   $\min_{\mathbf{b}/\sum_{\ell\in S'_m(t)} b_\ell=1} \sum_{i\in S'_m(t)} A_m^{(j),b_i}(a_i)\ ]$.

3. ***Terminaison :***
   à l'itération nb_{iter}
   $\boldsymbol{\mathcal{L}}[APP^{(nb_{iter})}(a_t)] = \boldsymbol{\mathcal{L}}[Obs(a_t)] + \sum_{i=1}^{m} \boldsymbol{\mathcal{L}}[Extr_i^{nb_{iter}}(a_t)]$.

Tab. 3.6 – Algorithme de Gallager version « discrète ».

Algorithme considéré	Complexité en terme de mémoire	Complexité calculatoire
B.P. (version α)	$\mathcal{O}(N\mu_b)$	$\mathcal{O}(nb_{iter}\ N{\mu_b}^2)$
B.P. avec APP partielles, min (version β)	$\mathcal{O}(N\mu_b)$	$\mathcal{O}(nb_{iter}\ N{\mu_b}^2)$
B.P., sans APP partielles, min (version γ)	$\mathcal{O}(N)$	$\mathcal{O}(nb_{iter}\ N\mu_b)$
B.P., sans APP partielles (version δ)	$\mathcal{O}(N)$	$\mathcal{O}(nb_{iter}\ N\mu_b)$
Min-Sum discret, min	$\mathcal{O}(N\mu_b)$	$\mathcal{O}(nb_{iter}\ N{\mu_b}^2)$

Tab. 3.7 – Complexités comparées des différentes versions de l'algorithme de Gallager sous l'hypothèse $N, \mu_b \gg d$.

3.1.8 Application de l'algorithme de Gallager et de ses versions approchées au registre filtré : quelques résultats de simulation

Les algorithmes que nous avons décrits précédemment s'appliquent à un mot du code (N, K) bruité par un canal. Le décodeur reçoit un mot bruité en sortie du canal et, à l'aide de la connaissance dont il dispose sur les structures du code et du canal, il cherche à identifier le mot de code émis.
À ce stade de description des algorithmes, il apparaît légitime de se demander pourquoi les

cryptanalystes ont eu l'idée d'appliquer l'algorithme de Gallager au système du registre filtré. Par ailleurs, la description des algorithmes qui a été faite précédemment fait apparaître dans les versions α, β, γ et δ de l'algorithme B.P. une probabilité de transition, qui sert à initialiser ces algorithmes. Cette probabilité est, dans le cas « standard » d'un canal de transmission, dictée par les propriétés du canal. Dans le cas où le canal est remplacé par une fonction booléenne, il nous faut examiner la façon dont vont être initialisés ces algorithmes. Nous nous efforçons ci-après d'éclaircir ces deux points.

Q. 1 : D'où vient l'idée d'appliquer l'algorithme de décodage des LDPC à la séquence produite par un registre filtré ?

Rappelons tout d'abord que, comme nous l'avons déjà vu aux chapitres 1 et 2, l'adaptation de ces algorithmes au registre filtré se fait de la façon suivante :

- les K bits d'initialisation du registre sont perçus comme les K bits d'information d'un code (N, K) ;
- la fonction booléenne est modélisée comme un canal binaire symétrique (BSC) sans mémoire ;
- la séquence (z) qui sort de la fonction est vue comme le résultat de la distorsion de la séquence PN par le canal BSC.

L'algorithme Belief Propagation appliqué à un code de faible densité exploite les relations linéaires qui régissent ce code : c'est un algorithme relativement « simple », qui est à relier au peu de contraintes dont on dispose sur les codes LDPC. Des exemples d'algorithmes de décodage plus complexes seraient ceux qui s'appliquent aux codes BCH ou aux codes Reed-Solomon (cf. [61]), qui exploitent les propriétés algébriques de ces codes.
Le code issu de la séquence PN vérifie une relation de récurrence linéaire liée au polynôme de rétroaction. Ce code est parfaitement caractérisé, mais nous ne savons pas comment exploiter la connaissance dont nous disposons pour le décoder. Ainsi, l'idée est de se ramener au décodage de Gallager et d'utiliser non plus le polynôme de rétroaction de la séquence, mais des multiples de poids aussi faible que possible de ce dernier afin de se ramener au décodage de codes à faible densité (ces multiples incluent éventuellement le polynôme de rétroaction lorsque ce dernier est creux).
Comme nous l'avons déjà souligné dans le paragraphe 3.1.2, l'efficacité du décodage de Gallager est grandement conditionnée par le poids des équations de parité. Afin de se ramener à des codes de faible densité, il nous faut donc, dans le cas du registre filtré, exhiber des équations de poids aussi faible que possible. Dans le chapitre 2, nous présentons quelques méthodes de génération d'équations de parité. Tout au long de ce rapport, nous nous sommes restreints aux algorithmes de Meier et Staffelbach (cf. [67]) et de Canteaut et Trabbia (cf. [10], [40]) ; la description de ces algorithmes peut être trouvée au chapitre 2, paragraphe 2.2.1 (cf. page 33).
Rappelons brièvement que le premier algorithme consiste à déduire des équations de parité du polynôme de rétroaction en opérant sur ce dernier des élévations au carré successives. Les équations obtenues sont alors de poids strictement égal à celui du polynôme de rétroaction et elles sont peu nombreuses.
L'algorithme suggéré par Canteaut et Trabbia cherche, quant à lui, tous les multiples de poids d donné du polynôme de rétroaction à l'aide d'une recherche par table. Cet algorithme est, certes, plus complexe que le premier, mais le nombre d'équations obtenues peut être élevé et leur poids est choisi (dans une certaine mesure (cf. page 36)) par le cryptanalyste. Nous renvoyons le lecteur au chapitre 2 page 33 pour une description approfondie de ces deux algorithmes.

Q.2 : Dans le cas d'un registre filtré, *i.e.* lorsqu'une fonction booléenne filtre la séquence (a), quelle est la probabilité de transition équivalente ?

Comme nous l'avons vu au chapitre 1, les séquences obtenues par combinaisons linéaires sur les vecteurs d'entrée de la fonction sont plus ou moins proches de la séquence de sortie. Chacune de ces combinaisons linéaires peut être vue comme l'entrée d'un canal binaire symétrique dont la probabilité de transition est évaluée grâce au spectre de la fonction booléenne, et dont la sortie est la séquence (z). La question Q.2 peut donc être reformulée de la façon suivante : quelle séquence allons-nous chercher à décoder et quelle est la probabilité d'erreur associée à cette opération de décodage ?
Naturellement, nous allons choisir de décoder la forme linéaire qui permettra de minimiser la probabilité d'erreur globale, favorisant ainsi la convergence des algorithmes B.P.. Nous expliquons ci-après pourquoi la séquence d'entrée de la fonction peut être retrouvée via une forme linéaire et développons le calcul de la probabilité de transition équivalente à celle d'un canal binaire symétrique.

A : Décoder une forme linéaire de la séquence d'entrée

Rappelons qu'il n'est pas obligatoire dans le cas du registre filtré de décoder directement la séquence d'entrée de la fonction, (a). Dans certains cas, celui de fonctions résilientes (cf. chapitre 1) notamment, nous serons même obligés de décoder une séquence issue de combinaisons linéaires des bits d'entrée de la fonction.

La structure linéaire du système implique que toute forme linéaire de la séquence d'entrée vérifie exactement les mêmes équations de parité que la séquence (a) : ainsi, à l'aide des équations de parité obtenues, nous pouvons choisir de décoder une forme linéaire de la séquence d'entrée qui serait, par exemple, moins « bruitée ».
Se pose ensuite le problème de retrouver la séquence (a) à partir de la séquence décodée.

Dans le cas du LFSR, le polynôme g est, sauf exception, choisi irréductible (voir chapitre 1). Toute forme linéaire des entrées de la fonction booléenne a une écriture polynomiale de la forme $\ell(x)$ avec $\deg(\ell) \leq K-1$. Or, d'après le théorème 1 énoncé au chapitre 1, tout polynôme de degré strictement inférieur à K est inversible modulo $g(x)$.
Donc $\forall \ell$ tel que $\deg(\ell) < K$, $\exists\ \Psi$ tel que $\ell.\Psi = 1 \mod g$.
L'inverse de ℓ modulo g est un polynôme de degré strictement inférieur à K et va donc constituer une forme linéaire, qui, appliquée à la séquence (a^ℓ) issue de ℓ, produira (a).

B : Déterminer la probabilité de transition équivalente à celle d'un BSC

Tout d'abord, notons que la probabilité de transition sur un canal binaire symétrique est la probabilité de commettre une erreur, *i.e.* que la sortie soit différente de l'entrée. Il s'agit donc de la probabilité de recevoir un '1' sachant qu'un '0' a été émis sur le canal, ou inversement de recevoir un '0' sachant qu'un '1' a été émis. En utilisant nos notations, évaluer la probabilité de transition d'un canal qui serait symétrique revient à évaluer

$$P(Z_t = 0 \mid a_t = 1) = P(Z_t = 1 \mid a_t = 0)$$

ou de façon plus générale

$$P(Z_t = 0 \mid a_t^{\ell\mathbf{u}} = 1) = P(Z_t = 1 \mid a_t^{\ell\mathbf{u}} = 0)$$

où $\ell_{\mathbf{u}}$ est la séquence que l'on cherche à décoder, qui correspond à une combinaison linéaire des bits de la séquence d'entrée (triviale ou non).
Le vecteur $\mathbf{u} \neq \mathbf{0_n}$ caractérise la forme linéaire en formant le produit scalaire avec le vecteur d'entrée de la fonction à l'instant t pour former le bit z_t.
Ainsi, nous cherchons en fait à évaluer

$$P[f(\mathbf{X}) = 1 \mid \mathbf{u}.\mathbf{X} = 0] \ . \tag{3.30}$$

En détaillant les équations (1.50) développées dans le chapitre 1 (cf. page 22) et en supposant que la fonction est équilibrée, c'est-à-dire que $P[f(\mathbf{X}) = 0] = P[f(\mathbf{X}) = 1] = \frac{1}{2}$, nous pouvons écrire :

$$\begin{aligned} P[f(\mathbf{X}) = 1 \mid \mathbf{u}.\mathbf{X} = 0] &= \frac{1}{2^n} \times [2^{n-1} + \widehat{f}(\mathbf{u})] \\ &= \frac{1}{2} + \frac{\widehat{f}(\mathbf{u})}{2^n} \qquad (3.31) \\ &= \frac{1}{2} - \frac{\widehat{f_\chi}(\mathbf{u})}{2^{n+1}} \qquad (3.32) \end{aligned}$$

où $\widehat{f}$ est la transformée de Fourier de f, $\widehat{f_\chi}$ la transformée de Fourier de la fonction signe, $f_\chi = (-1)^f$ (voir chapitre 1).

À première vue, il paraît naturel de choisir la probabilité de transition p telle que la probabilité d'erreur sur la séquence en sortie de la fonction soit la plus faible possible. Comme on peut décoder soit une forme linéaire, soit son contraire (ce qui équivaut à dire que nous allons décoder non pas des fonctions linéaires mais affines de la séquence d'entrée), nous allons simplement chercher à ce que la probabilité d'erreur soit la plus éloignée de $\frac{1}{2}$ possible. Et, si cette probabilité d'erreur est supérieure à $\frac{1}{2}$, nous décoderons non pas la séquence correspondant à p mais sa séquence complémentaire, bruitée avec une probabilité $1 - p$.

$$\begin{aligned} \left| p - \frac{1}{2} \right| &= \max_{\mathbf{u}} \left| P[\mathbf{u}.\mathbf{x} = 1 \mid f(\mathbf{x}) = 1] - \frac{1}{2} \right| \\ &= \max_{\mathbf{u}} \left| \frac{\widehat{f}(\mathbf{u})}{2^n} \right| \ . \qquad (3.33) \end{aligned}$$

Nous décodons alors la séquence la moins bruitée ou l'opposée de la séquence la plus bruitée.

Cependant, supposons que la fonction ne soit pas résiliente et qu'elle possède plusieurs formes linéaires de poids 1 (*i.e.* plusieurs entrées) ayant une corrélation non nulle avec la sortie : dans certains cas, il pourrait être plus efficace de considérer l'ensemble de ces valeurs non nulles et ainsi, de tenir compte de l'observation de plusieurs décalées d'un même bit.
Cette remarque est également valable si les espacements entre les entrées de la fonction sont tels qu'une forme linéaire non triviale (de poids supérieur à 1) se répète plusieurs fois et que sa corrélation avec la sortie ainsi que la corrélation de ses décalées avec la sortie apportent de l'information sur la séquence d'entrée.

Ainsi, pour une fonction booléenne donnée, le choix de la probabilité de transition suivant l'équation (3.33) et qui correspond donc à l'initialisation de l'algorithme B.P., n'est optimale à coup sûr que dans le cas d'une fonction résiliente et telle que ses espacements sont tous premiers entre

eux. Dans tous les autres cas, il faudra examiner les différentes probabilités existantes afin de faire le choix optimal comme nous l'avons fait pour différentes fonctions (figures 3.5, 3.6 et 3.7). Si l'utilisation de fonctions résilientes dans le cas d'un registre filtré ne présentait à première vue qu'un intérêt modéré, il apparaît dorénavant que **ces fonctions améliorent la résistance du système (vis à vis du type d'attaques que nous présentons)** : leur utilité est donc ici légitimée, de façon certes moins directe que dans le cas d'un registre par combinaison.

Les figures 3.5, 3.6 et 3.7 permettent de comparer les performances obtenues en initialisant différemment l'algorithme B.P. (version δ) sur un même système. Nous avons testé deux types d'initialisation sur des fonctions non résilientes :

- nous avons tout d'abord initialisé « classiquement » l'algorithme en ne tenant compte que du maximum du spectre de la fonction. La première étape de l'algorithme fut alors :
  soit $\mathbf{u_0} \neq \mathbf{0_n}$ tel que $|\widehat{f}(\mathbf{u_0})| = \max_{\mathbf{u}\neq\mathbf{0_n}} |\widehat{f}(\mathbf{u})|$,
  soit $p_0 < 1/2$ la probabilité de transition équivalente

$$\begin{aligned} z_t = 1 &\implies Obs(a_t) = 1 - p_0 \\ z_t = 0 &\implies Obs(a_t) = p_0 \ ; \end{aligned} \tag{3.34}$$

- une initialisation tenant compte de toutes les corrélations des versions décalées de la forme linéaire triviale de poids 1, la fonction étant non résiliente. Soient $z_t, z_{t-\lambda_0}, \ldots, z_{t-\sum\lambda_i}$ générés par les vecteurs $\mathbf{X}(t), \ldots, \mathbf{X}(t-\sum\lambda_i)$ qui contiennent tous le bit a_t. Afin d'alléger l'écriture, notons ces bits $z_{t_1}, \ldots, z_{t_n}$, qui correspondent respectivement à a_t en première, deuxième, ..., dernière position du vecteur d'entrée.
  Soit $c_i = \frac{\widehat{f}(\mathbf{v^i})}{2^n}$ où $\mathbf{v^i}$ est le i-ième vecteur canonique : $\mathbf{v^i} = (0, \ldots, 0, 1, 0, \ldots 0)$, la seule composante non nulle de $\mathbf{v^i}$ étant la i-ième. Alors, l'initialisation est faite de la façon suivante :

$$Obs(a_t) = \frac{\prod_{i=1}^{n} \frac{(1+(-1)^{z_{t_i}} c_i)}{2}}{\prod_{i=1}^{n} \frac{(1+(-1)^{z_{t_i}} c_i)}{2} + \prod_{i=1}^{n} \frac{(1-(-1)^{z_{t_i}} c_i)}{2}} \tag{3.35}$$

  puisque $P(a_t = 1) \propto P\left[\bigcap_{\theta=0,\lambda_0,\ldots,\sum\lambda_i} \{a_t = 1 \text{ dans } \mathbf{X}(t+\theta)\}\right]$
  c'est-à-dire que la probabilité que a_t soit égal à 1 est la probabilité que a_t vaille 1 dans chacun des vecteurs d'entrée auxquels il appartient. L'équation (3.35) a été obtenue en supposant que tous ces évènements sont indépendants.

L'ensemble des résultats présentés sur les figures 3.5, 3.6 et 3.7 a été obtenu à partir de 1000 tirages, effectués sur un registre de mémoire $K = 21$, de polynôme de rétroaction : $g(X) = 1 + X^2 + X^3 + X^5 + X^{10} + X^{11} + X^{12} + X^{14} + X^{21}$, suggéré dans [10]. Les équations de parité utilisées sont de poids $d = 3$ et générées à l'aide de l'algorithme Canteaut-Trabbia. Les simulations ont été effectuées en prenant comme valeurs des espacements : $(\lambda_0, \lambda_1, \lambda_2) = (3, 5, 4)$, $(\lambda_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4) = (3, 5, 4, 3, 2)$ dans le cas de la fonction f_{14} à $n = 7$ entrées.

Nous constatons sur ces figures que la plupart du temps, l'initialisation tenant compte de toutes les versions décalées de la séquence d'entrée permet à l'algorithme de converger plus rapidement. Cependant, la différence de performance n'est pas très élevée : les simulations ont montré

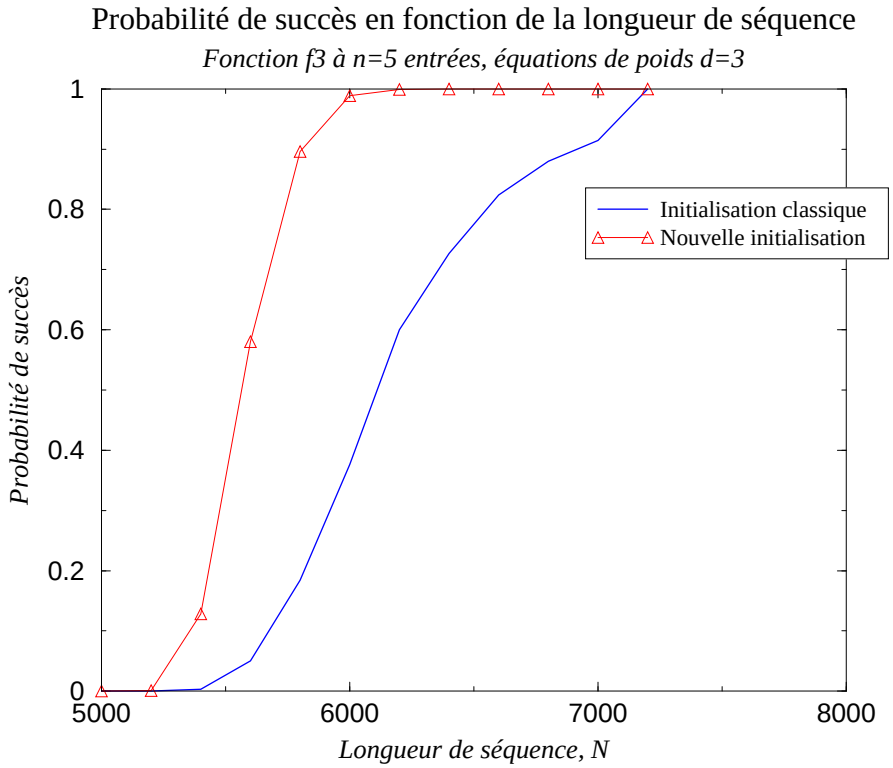


Fig. 3.5 – Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_3 à $n = 5$ entrées.

que quelques centaines de bits supplémentaires suffisent à combler l'écart de performance entre les deux types d'initialisation. Remarquons toutefois que chacune des deux initialisations donne des résultats similaires sur f_4, contrairement aux résultats obtenus sur f_3 et f_{14}. L'explication peut en être la suivante :

- si nous observons le spectre de f_3, donné en annexe, nous constatons que trois formes linéaires atteignent le maximum du spectre (en valeur absolue). Cela signifie que l'algorithme initialisé classiquement, *i.e.* en utilisant le maximum du spectre, a trois séquences solutions qui peuvent donc interférer entre elles et ralentir la convergence de l'algorithme. D'où l'intérêt d'initialiser en utilisant les décalées de la forme linéaire de poids 1 ;
- la transformée de Fourier de f_{14} est non nulle en chacune des formes linéaires de poids 1 et possède des valeurs élevées relativement au maximum du spectre : il paraît donc intéressant d'utiliser ces valeurs ;
- il est intéressant de comparer les résultats de performances obtenues avec f_3 et f_4 : trois formes linéaires correspondent au maximum du spectre dans le cas de f_3, deux seulement en ce qui concerne f_4. Cela semble avoir pour conséquence directe une différence de performances de l'algorithme de Gallager initialisé classiquement. Par ailleurs, les formes linéaires de poids 1 sont, dans le cas de f_4, toutes corrélées à la sortie, contrairement au cas de f_3. Nous constatons donc que l'algorithme de Gallager initialisé à l'aide de toutes les corrélations non nulles des formes linéaires de poids 1 donne de meilleurs résultats dans le cas de f_4· Cependant, les deux initialisations conduisent à des résultats quasi-identiques pour ce qui est de f_4. L'explication de cette similitude de comportements semble s'expliquer par

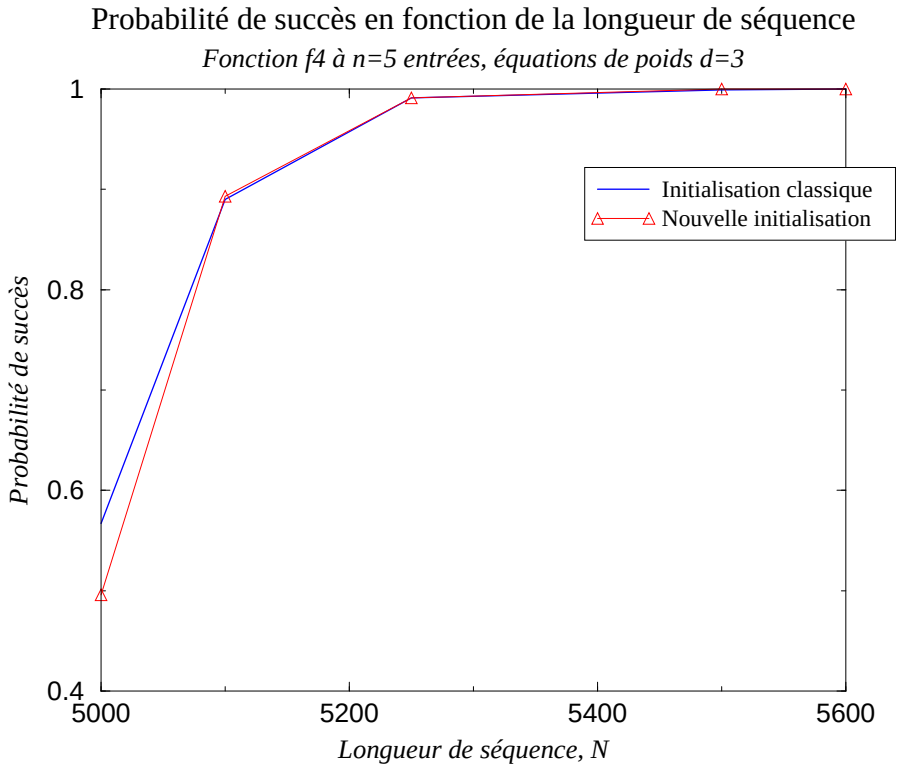


Fig. 3.6 – Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_4 à $n = 5$ entrées.

un spectre « favorable » à la convergence de l'algorithme de Gallager classique dans le cas où deux formes linéaires seulement atteignent le pic de corrélation, ce qui pourrait combler le retard de l'algorithme initialisé classiquement que nous observons sur f_3.

Comparaison des différentes versions de l'algorithme B.P. sur canal binaire symétrique et sur canal booléen à l'aide de quelques résultats de simulation

En raison de ce qui a été mentionné précédemment, les fonctions utilisées dans les simulations qui suivent sont résilientes et les espacements entre les entrées de la fonction sont choisis premiers entre eux et couvrant toute la mémoire du registre. Ainsi, l'initialisation des registres est faite de façon classique, en fonction du maximum du spectre de la fonction.
Nous présentons plusieurs ensembles de résultats qui permettront de comparer les performances dans différents cas de figure. D'une part nous allons faire varier le canal de transmission qui pourra être un canal binaire symétrique ou un canal booléen. D'autre part, nous allons faire varier le polynôme de rétroaction utilisé, ce qui se traduit par des équations de parité de nature différente et plus ou moins nombreuses.
Quel que soit le système considéré, le nombre d'itérations maximal a été fixé à soixante et chaque

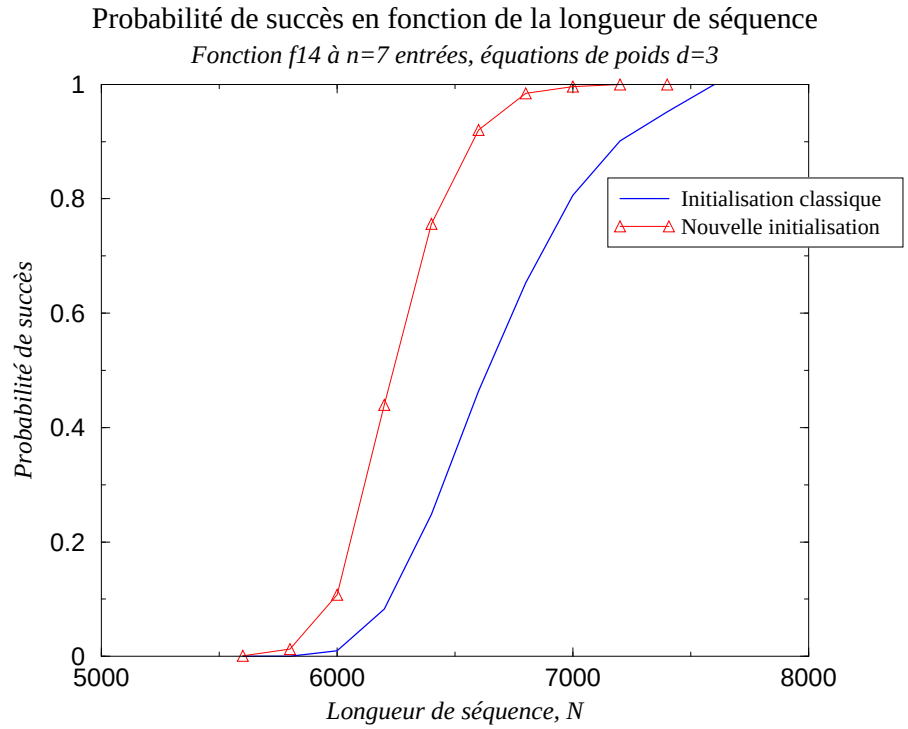


Fig. 3.7 – Performances de l'algorithme B.P. version δ initialisé de deux façons différentes, fonction f_{14} à $n = 7$ entrées.

probabilité de succès a été évaluée à partir des résultats obtenus sur mille tirages.

Comparaison des algorithmes B.P. sur un canal BSC

Ce premier ensemble de résultats a pour objectif de comparer les performances de l'algorithme B.P. et des versions qui en dérivent. Nous avons utilisé un canal binaire symétrique, et avons fait varier sa probabilité de transition. Dans la mesure où la complexité des algorithmes B.P. version α et β est proportionnelle au carré du nombre moyen d'équations par bit, μ_b, (cf. équations (3.23) et (3.25)) nous avons considéré un système dans lequel nous disposions de relativement peu d'équations de parité. Le polynôme de rétroaction utilisé fut le suivant

$$g(x) = 1 + x^{37} + x^{100} \ .$$

Les équations de parité ont été générées à l'aide d'élévations au carré successives, comme l'ont suggéré Meier et Staffelbach dans [66] (cf. algorithme décrit au chapitre 2). Les tables 3.8 et 3.9 ci-après présentent le taux moyen de succès du décodage.

Le tableau 3.8 présente les résultats obtenus pour une longueur de séquence $N = 10000$. Le tableau 3.9 a été obtenu pour une longueur $N = 20000$. Chacun de ces tableaux figure, pour différentes probabilités de transition, le pourcentage de succès des algorithmes α, β, γ, δ et Min-Sum. Nous avons également mentionné le nombre moyen d'équations de parité par bit dans chaque cas de figure puisqu'il constitue un facteur déterminant du succès des algorithmes.

p	**0.58**	**0.59**	**0.60**
Algorithme α	0.013	0.194	0.742
Algorithme β	$< 10^{-3}$	0.006	0.166
Algorithme γ	$< 10^{-3}$	0.046	0.395
Algorithme δ	0.011	0.188	0.759
Min-Sum discret	$< 10^{-3}$	$< 10^{-3}$	0.011

Tab. 3.8 – Taux de succès des différents algorithmes B.P. en fonction de la probabilité du canal BSC(p), avec $N = 10000$, $K = 100$, poids des équations $d = 3$, $\mu_b \approx 17.2$ équations par bit.

p	**0.57**	**0.58**	**0.59**
Algorithme α	0.052	0.625	0.990
Algorithme β	$< 10^{-3}$	0.006	0.295
Algorithme γ	$< 10^{-3}$	0.075	0.734
Algorithme δ	0.030	0.552	0.984
Min-Sum discret	$< 10^{-3}$	0.003	0.069

Tab. 3.9 – Taux de succès des différents algorithmes B.P. en fonction de la probabilité du canal BSC(p) avec $N = 20000$, $K = 100$, poids des équations $d = 3$, $\mu_b \approx 20.2$ équations par bit.

Commentaires :

Remarquons tout d'abord que les résultats obtenus avec une longueur de séquence égale à $N = 10000$ sont moins bons que ceux obtenus pour une longueur de séquence $N = 20000$: le nombre moyen d'équations de parité par bit dans le premier système est en effet inférieur (17,2 contre 20,2 dans le second) ce qui se traduit par une perte d'information dans les probabilités générées.

Par ailleurs, les tableaux 3.8 et 3.9 montrent que les approximations réalisées sur la version originale de l'algorithme B.P. engendrent une perte en terme de performances : les versions β, γ, δ de l'algorithme B.P., le Min-Sum sont moins performants que l'algorithme α. Nous observons également que l'algorithme le plus proche des performances de l'algorithme B.P. est l'algorithme δ. Rappelons que ce dernier consiste à calculer de façon exacte l'information extrinsèque, mais à propager des APP totales au lieu d'APP partielles. L'approximation la plus pénalisante semble donc être celle qui consiste à ne pas calculer la probabilité extrinsèque de façon exacte : les versions β, γ, Min-Sum qui réalisent cette approximation, sont nettement moins performantes que la version δ de l'algorithme B.P.. Cette observation est surprenante dans la mesure où dans le cas des codes LDPC, l'approximation du Min-Sum est très fréquemment utilisée.
Comment expliquer un tel phénomène ?
Nous suggérons la justification suivante : dans les systèmes que nous traitons (registres filtrés par une fonction non-linéaire), les bits appartiennent à un grand nombre d'équations de parité par rapport au nombre d'équations par bit dans le cas d'un code LDPC. L'effet de considérer une APP globale et non partielle est donc quasi-transparent, puisqu'APP partielle et APP globale ne

diffèrent que d'une extrinsèque sur un nombre total « élevé ». En revanche, le calcul approximatif de l'extrinsèque dans un système où les équations sont de poids $d = 3$ se révèle très pénalisant : lorsque les équations sont de poids faible, on peut espérer récupérer un biais significatif lors du calcul exact des probabilités extrinsèques.

Fait surprenant néanmoins, l'algorithme β, qui ne réalise que l'approximation sur la probabilité extrinsèque est, semble-t-il, moins performant que l'algorithme γ, qui réalise non seulement cette approximation mais aussi celle de ne pas utiliser d'APP partielles.

Enfin, nous observons que les performances du Min-Sum « discret » sont très mauvaises : la propagation d'informations discrètes ne semble pas adaptée au cas d'un vrai canal BSC.

Concluant à l'aune de ces simulations que l'algorithme B.P. version δ était le meilleur compromis performances-complexité (au moins pour des systèmes décodés à l'aide de peu d'équations), nous avons évalué ses performances dans deux contextes différents afin d'étudier la sensibilité des algorithmes itératifs à certains paramètres du système.

Comparaison des performances obtenues sur un canal binaire symétrique et sur une fonction booléenne

Nous nous intéressons ici aux différences de performances obtenues en utilisant un même algorithme, mais sur deux systèmes différents : le système réel incluant une fonction booléenne et le système modélisé en canal binaire symétrique.
Nous avons appliqué la version δ de l'algorithme B.P. à diverses fonctions booléennes résilientes. Pour chacune d'entre elles, nous avons calculé la probabilité de transition p d'un canal binaire symétrique équivalent et simulé le même algorithme δ sur une séquence bruitée par un BSC(p). Rappelons que la fonction étant résiliente et les espacements premiers entre eux, la question du choix de l'initialisation de l'algorithme ne se pose pas.

Plusieurs points de comparaison nous ont paru pertinents et les courbes qui suivent s'inscrivent dans cette logique ; nous avons fait varier les paramètres suivants :

- **la probabilité d'erreur sur la séquence décodée :** nous avons considéré une fonction booléenne de probabilité de transition équivalente $p = 0.25$ et deux fonctions booléennes de probabilité de transition équivalente $p = 0.375$;
- **le nombre d'entrées de la fonction :** pour une même probabilité de transition équivalente $p = 0.375$, nous avons considéré une fonction à $n = 5$ entrées et une fonction à $n = 7$ entrées, toutes deux « plateau » ;
- **le poids des équations de parité :** nous avons cryptanalysé des systèmes en utilisant des équations de poids $d = 3$ et de poids $d = 5$;
- **le nombre d'équations de parité :** pour un poids d'équation donné ($d = 3$ ou 5), nous avons testé des systèmes pour lesquels le nombre d'équations relativement à la longueur de séquence est soit faible, soit élevé.

Par ailleurs, dans certains cas, nous avons également appliqué l'algorithme du Min-Sum, afin de comparer ses performances à celles de l'algorithme δ.

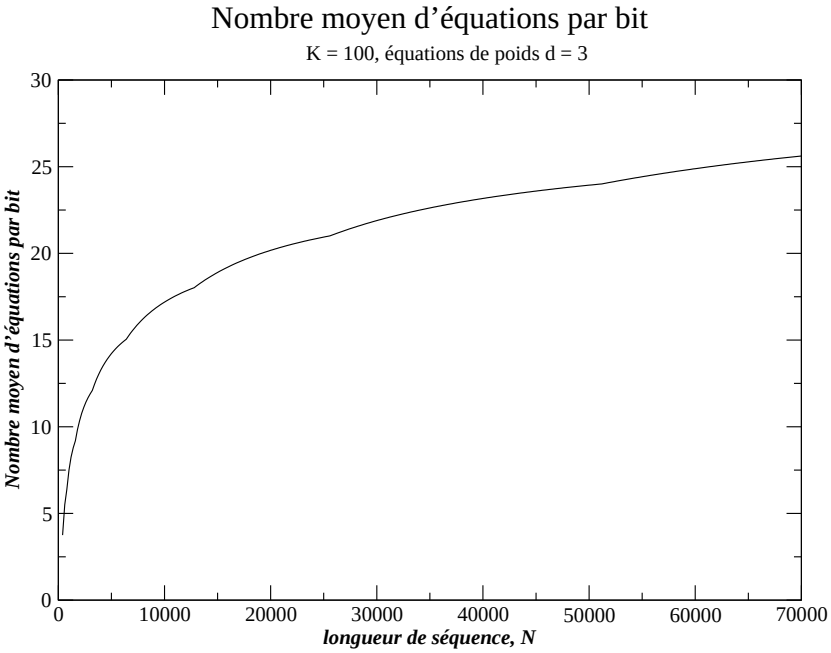


Fig. 3.8 – Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 100$, $d = 3$, équations générées par élévations au carré successives.

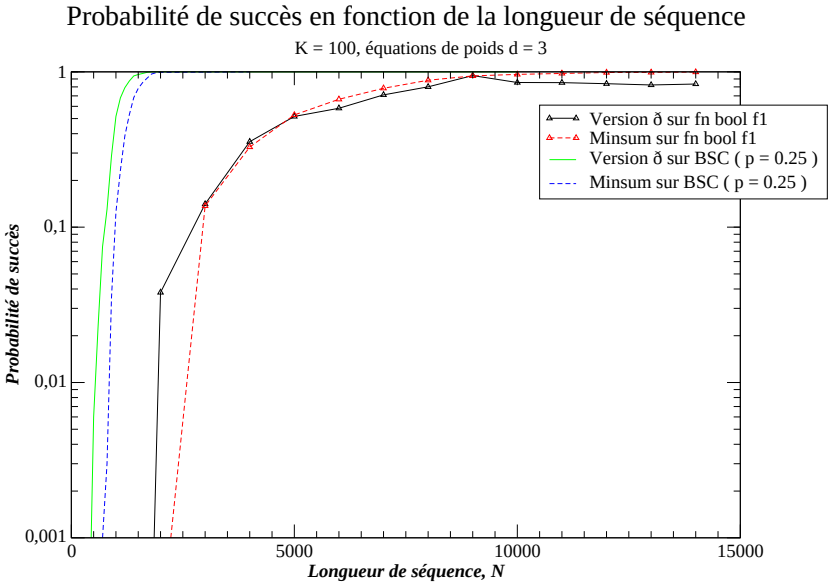


Fig. 3.9 – Comparaison des taux de convergence des algorithmes de Gallager version δ et Min-Sum sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 100$, $d = 3$.

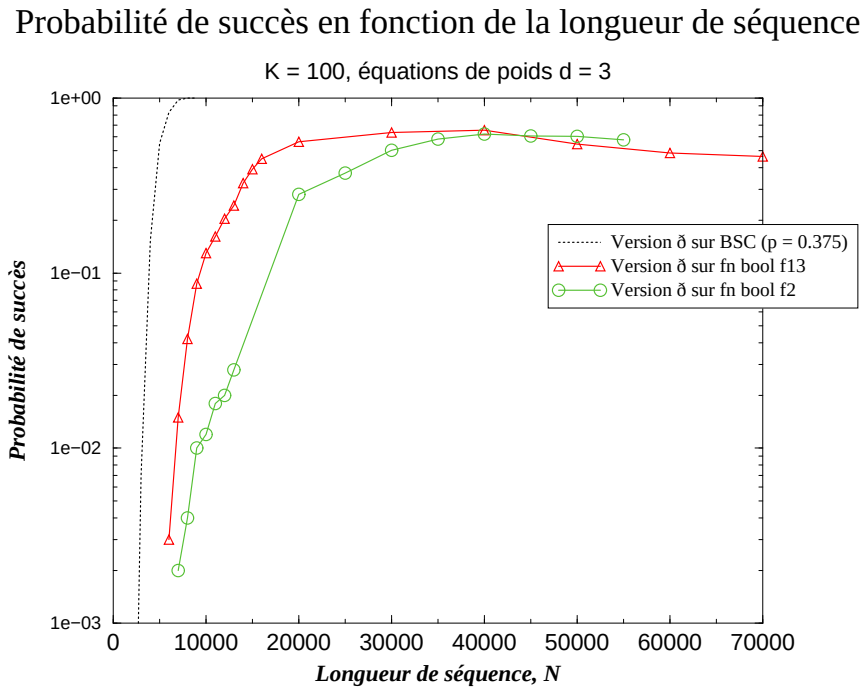


Fig. 3.10 – Comparaison des taux de convergence de de l'algorithme de Gallager version δ sur BSC($p = 0.375$) et sur deux fonctions de même probabilité de transition équivalente, f_2 à $n = 5$ entrées, f_{13} à $n = 7$ entrées ; $K = 100$, $d = 3$.

Afin de faciliter la lecture de ce document, présentons dans l'ordre l'ensemble des résultats obtenus :

- **A. Résultats obtenus avec des équations de parité de poids $d = 3$:**
  - **(a)** *cas où le nombre d'équations est faible :*
    le polynôme de rétroaction utilisé est

$$g(X) = 1 + X^{37} + X^{100} \ .$$

    La figure 3.8 donne à titre indicatif le nombre moyen d'équations par bit en fonction de la longueur de séquence disponible.
    La figure 3.9 présente les résultats obtenus avec la fonction f_1 et avec un canal de probabilité de transition $p = 0.25$.
    La figure 3.10 présente les résultats obtenus sur un canal binaire symétrique de probabilité de transition $p = 0.325$ et sur les fonctions f_{13} (fonction à $n = 7$ entrées) et f_2 (fonction à $n = 5$ entrées) ;
  - **(b)** *cas où l'on dispose d'un grand nombre d'équations (relativement à la longueur de la séquence) :*
    le polynôme de rétroaction utilisé est :

$$g(X) = 1 + X^2 + X^3 + X^5 + X^{10} + X^{11} + X^{12} + X^{14} + X^{21} \ .$$

    La figure 3.11 donne le nombre moyen d'équations par bit en fonction de la longueur de séquence disponible.
    Comme précédemment, la figure 3.12 présente les résultats obtenus avec la fonction f_1 et

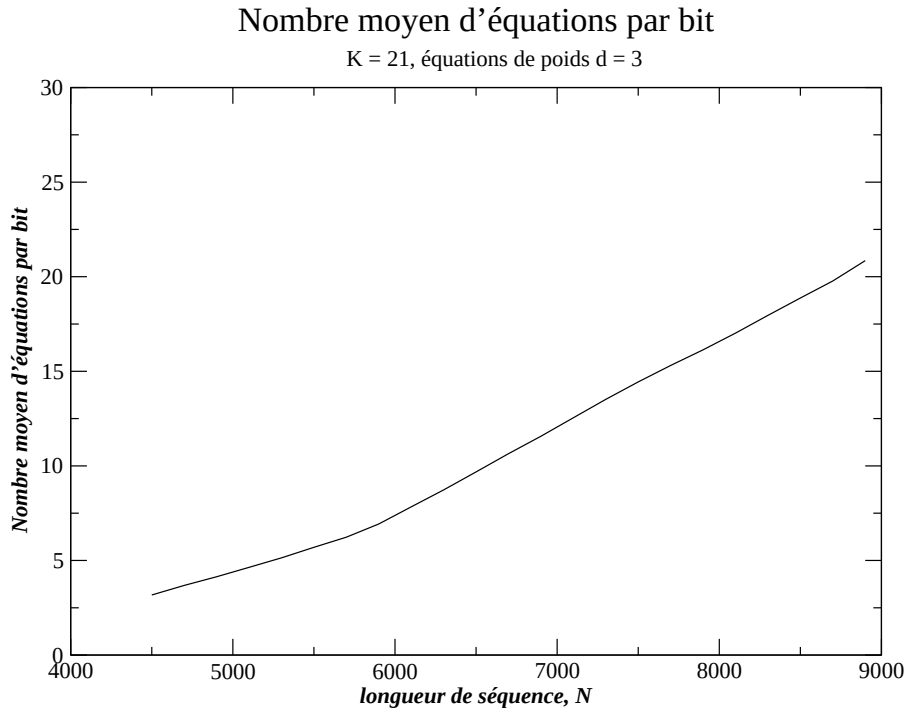


Fig. 3.11 – Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 21$, $d = 3$, équations générées par l'algorithme Canteaut-Trabbia [10].

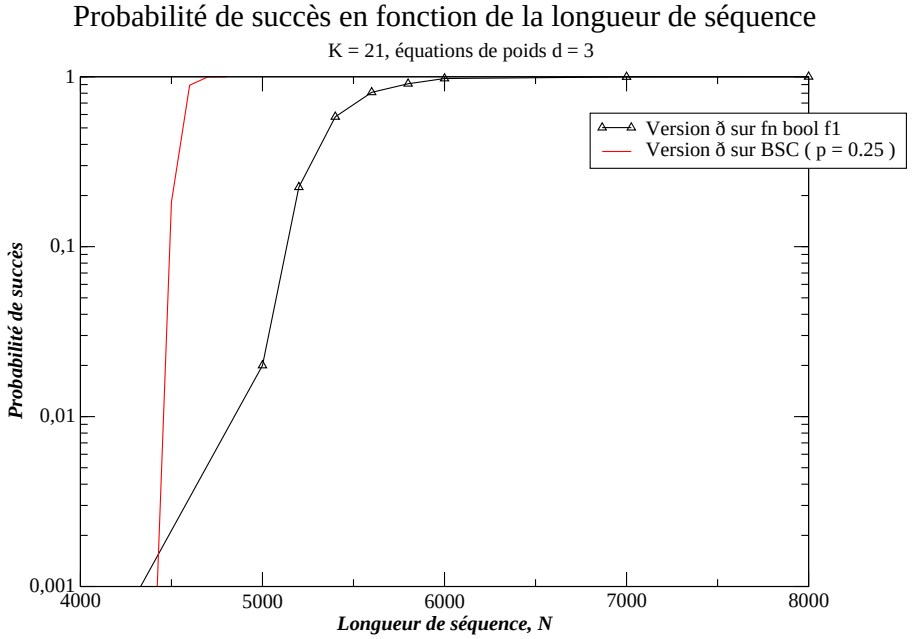


Fig. 3.12 – Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 21$, $d = 3$.

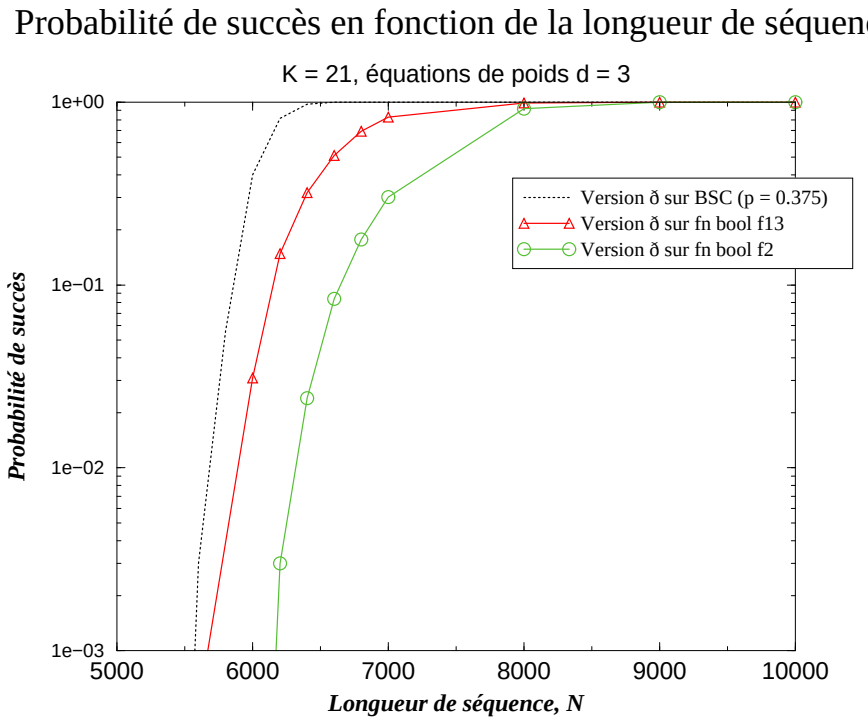


Fig. 3.13 – Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.375$) et sur deux fonctions de même probabilité de transition équivalente, f_2 à $n = 5$ entrées, f_{13} à $n = 7$ entrées ; $K = 21$, $d = 3$.

avec un canal de probabilité de transition $p = 0.25$. La figure 3.13 présente les résultats obtenus sur un canal binaire symétrique de probabilité de transition $p = 0.325$ et sur les fonctions f_{13} et f_2.

– **B. Résultats obtenus avec des équations de parité de poids $d = 5$:**
  – **(a)** *cas où le nombre d'équations est faible :*
  le polynôme de rétroaction utilisé est

$$g(X) = 1 + X^7 + X^{111} + X^{118} + X^{120} \ .$$

  La figure 3.14 donne le nombre moyen d'équations par bit en fonction de la longueur de séquence disponible.
  La figure 3.15 donne les résultats obtenus avec la fonction f_1 et avec un canal de probabilité de transition $p = 0.25$. Nous ne donnons pas de résultats de simulation pour une probabilité d'erreur supérieure : le nombre d'équations étant faible, la probabilité de succès est très peu élevée. Ainsi, la présentation de résultats pour une probabilité supérieure à $p = 0.25$ sur ce système aurait requis une longueur d'observation très élevée et un temps de simulation déraisonnable ;
  – **(b)** *cas où l'on dispose de plus d'équations :*
  le polynôme utilisé est

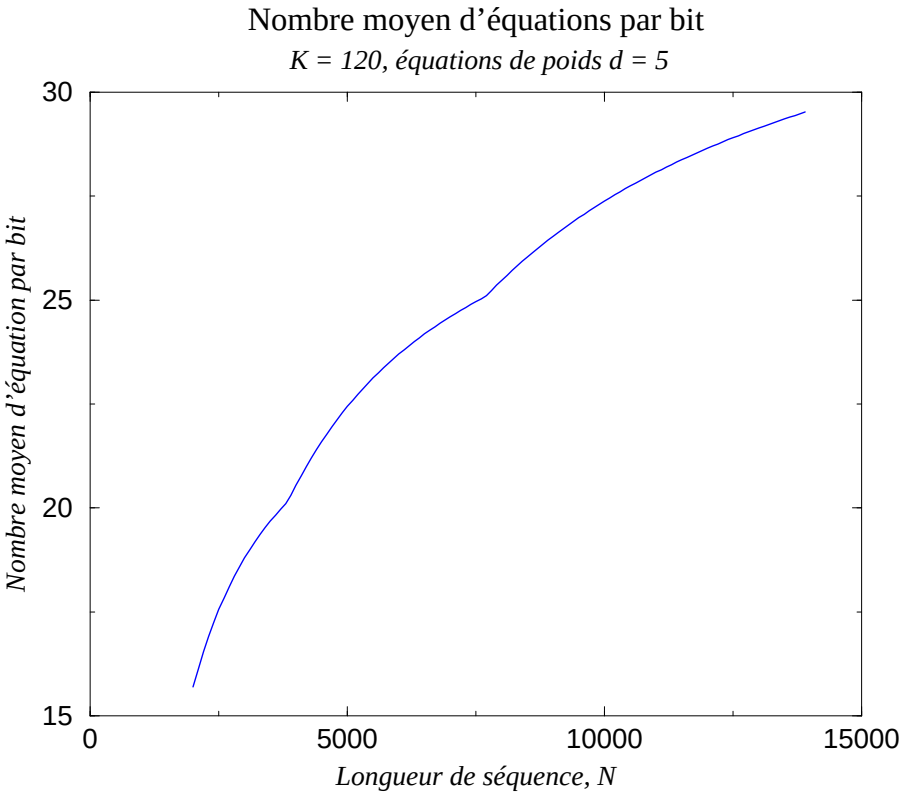


Fig. 3.14 – Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 120$, $d = 5$, équations générées par élévations au carré successives.

$$g(X) = 1 + X + X^3 + X^5 + X^9 + X^{11} + X^{12} + X^{17} + X^{19} + X^{21} + X^{25} + X^{27} + X^{29} + X^{32} + X^{33} + X^{38} + X^{40} \ .$$

Le nombre moyen d'équations par bit en fonction de la longueur de séquence est représenté sur la figure 3.16. La figure 3.17 présente les résultats obtenus sur un canal de probabilité de transition $p = 0.25$ et avec la fonction f_1.

Commentaires

Les simulations que nous avons effectuées ont montré que les résultats obtenus sur canal binaire symétrique sont constamment meilleurs que ceux obtenus sur fonction booléenne.

Les raisons essentielles qui justifient une telle différence nous semblent être les suivantes :

– le canal binaire symétrique est un canal sans mémoire qui atténue chacun des bits de façon indépendante. En revanche, de par sa structure, le LFSR filtré par une fonction booléenne est indéniablement sujet à un effet de mémoire qui va à l'encontre des hypothèses de l'algorithme Belief Propagation (cf. paragraphe 3.1.2, hypothèse H_1) ;

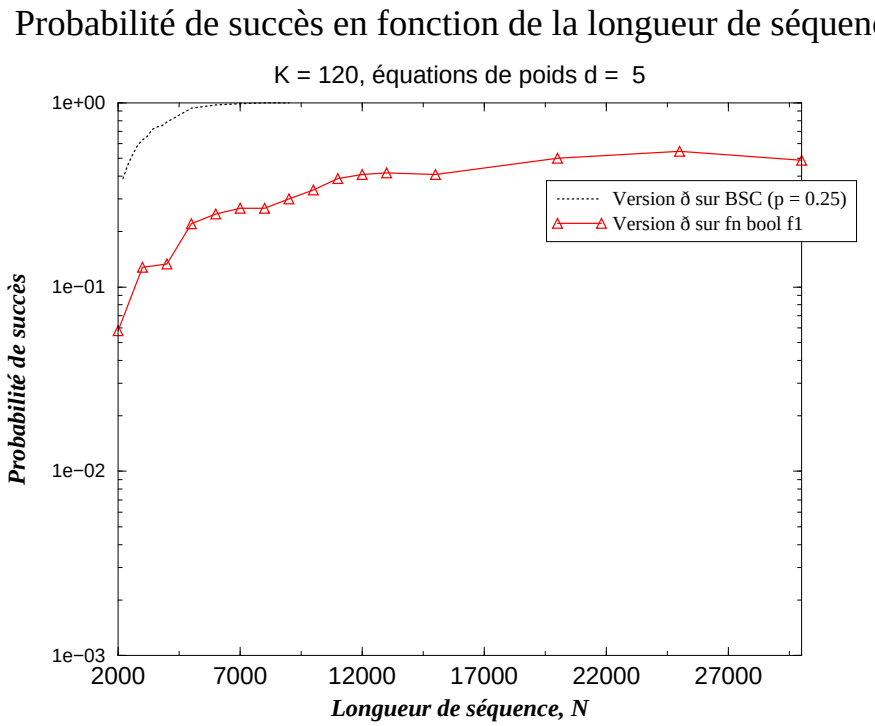


Fig. 3.15 – Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 120$, $d = 5$.

– dans le cas d'une fonction booléenne, il arrive que le maximum de la transformée de Fourier soit atteint par plusieurs formes linéaires. Il arrive également que plusieurs valeurs de la transformée de Fourier soient proches de la valeur maximale. En terme de codage, cela correspond à la situation où plusieurs mots de code sont à une même distance minimale du mot de code reçu, sinon à des distances très proches de cette distance minimale. Cette situation est handicapante : le décodeur va « hésiter » entre les différentes solutions possibles, ce qui va dégrader ses performances.

•Figure 3.8, 3.11, 3.14, 3.16 :
Ces figures donnent l'évolution du nombre moyen d'équations par bit en fonction de la longueur de séquence. Nous avons calculé ces quantités sur les séquences dont nous disposions ; notons que, comme prévu par la théorie (voir paragraphe 2.2.1 du chapitre 2, équations 2.4 et 2.5 page 35 et 36 respectivement), l'évolution du nombre d'équations dans le cas où ces dernières sont obtenues par élévations au carré successives suit une loi logarithmique (figures 3.8 et 3.14) ; en revanche, les équations obtenues grâce à l'algorithme de Canteaut-Trabbia [10] donnent lieu à une évolution linéaire de μ_b (figures 3.11 et 3.16).

• Figure 3.9 :
Nous remarquons tout d'abord que, dans ce cas, quel que soit le support de transmission de la séquence, les performances du Min-Sum et de l'algorithme δ sont très proches. Or, nous avions

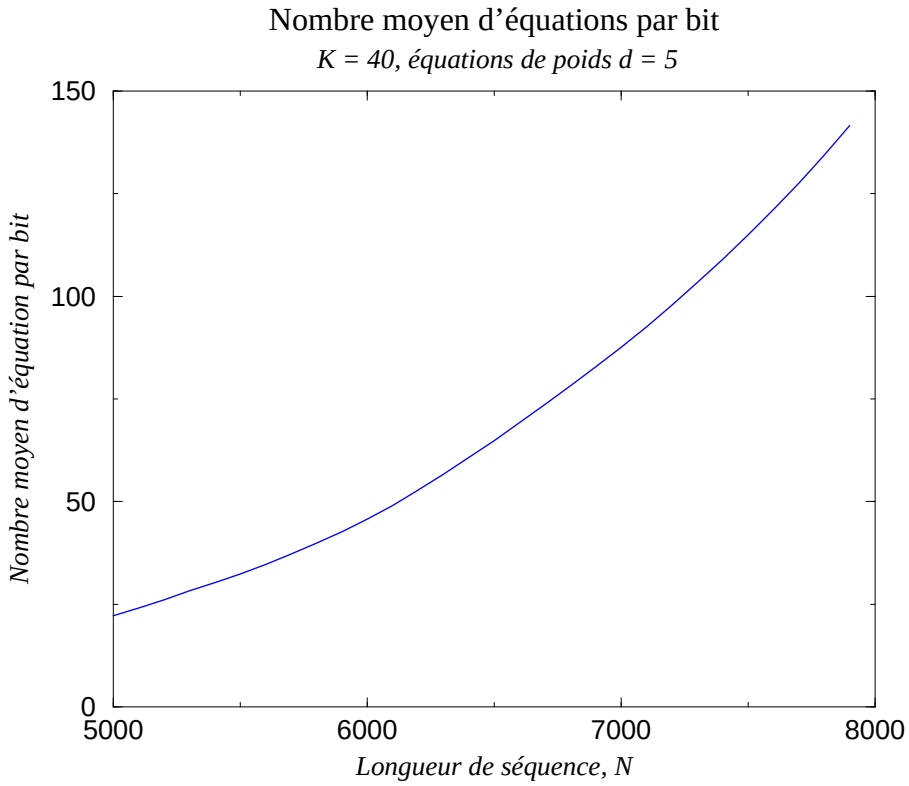


Fig. 3.16 – Nombre moyen d'équations de parité par bit en fonction de la longueur de séquence, $K = 40$, $d = 5$, équations générées par l'algorithme Canteaut-Trabbia [10].

observé dans les tableaux 3.8 et 3.9 que le Min-Sum avait des performances très mauvaises sur le canal binaire symétrique en comparaison avec les versions approchées de l'algorithme B.P.. Dans le cas de l'expérience sur la fonction à $n = 5$ entrées, il est important de noter que la probabilité de transition, et donc le nombre d'erreurs induites sur la séquence de départ, est faible, égal à 25%. Nous pouvons donc en déduire que l'algorithme Min-Sum semble (au moins) adapté à la cryptanalyse de séquences produites par une fonction booléenne telle que la probabilité d'erreur correspondante est assez faible. Nous observerons par la suite que le Min-Sum n'est pas adapté au décodage de séquences plus bruitées.

Par ailleurs, nous observons une différence de comportement des algorithmes sur canal binaire symétrique et sur fonction booléenne : le cas du canal binaire symétrique apparaît, comme prévu, plus favorable que celui de la fonction booléenne. La différence est surtout notable dans le cas où l'on se fixe un objectif de probabilité de succès proche de 1 : il faut environ $N = 2000$ bits pour que les algorithmes convergent sur le canal binaire symétrique contre $N = 9000$ sur la fonction booléenne.

• Figure 3.10 :
La figure 3.10 présente la probabilité d'erreur obtenue sur deux fonctions résilientes de même probabilité de transition $p = 0.375$. La fonction f_{13} est donnée en annexe ; il s'agit d'une fonction à $n = 7$ entrées, 3-résiliente. La fonction f_2 est une fonction 1-résiliente, à $n = 5$ entrées. Nous

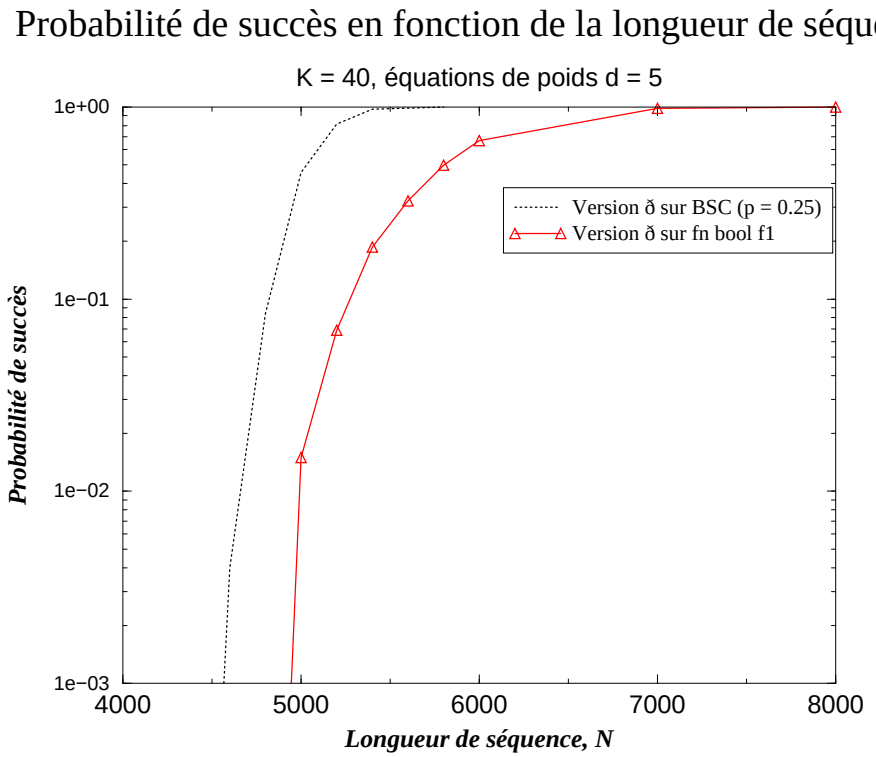


Fig. 3.17 – Comparaison des taux de convergence de l'algorithme de Gallager version δ sur BSC($p = 0.25$) et sur la fonction f_1 à $n = 5$ entrées ; $K = 40$, $d = 5$.

remarquons que, pour une probabilité d'erreur donnée, deux fonctions booléennes différentes ne donnent pas les mêmes résultats : la fonction f_{13} à $n = 7$ entrées présente de meilleures performances que la fonction f_2. Pourtant, ces fonctions sont toutes deux de type « plateau ». Nous proposons toutefois une explication à ce phénomène : considérons la fonction f_2. D'après la description qui en est faite dans l'annexe A, et en vertu de l'égalité de Parseval, nous pouvons dénombrer 2^4 valeurs non nulles de sa transformée de Fourier. Il existe donc 2^4 combinaisons linéaires de la séquence d'entrée qui ont une même corrélation (en valeur absolue) non nulle avec la sortie. Et donc, parmi tous les 2^5 « mots de code » possibles (où un mot de code correspond à une combinaison linéaire appliquée à la séquence (a)), la **moitié** d'entre eux est à une même distance du mot reçu (ou de son complémentaire).
Considérons maintenant la fonction f_{13} : nous pouvons, pour cette fonction, dénombrer 2^4 valeurs non nulles de sa transformée de Fourier. Ainsi, il existe 2^4 mots de code à même distance du mot reçu (ou son complémentaire), sur les 2^7 mots de code possibles. Le ratio de mots de code solutions de l'algorithme de Gallager est donc nettement moins favorable à la convergence de ce dernier dans le cas de la fonction f_2 à $n = 5$ entrées que dans celui de la fonction f_{13} à $n = 7$ entrées. D'où les résultats observés.

Nous n'avons pas indiqué les performances de l'algorithme du Min-Sum dans ce cas de figure, car elles étaient très mauvaises quand le système comprenait une fonction booléenne : la probabilité de succès restait très proche de zéro. Cela conforte l'idée selon laquelle l'algorithme du Min-Sum est efficace lorsque la probabilité d'erreur n'est pas trop élevée.

Enfin, nous constatons une différence importante entre les figures 3.9 et 3.10 : un taux de 80%

de réussite est atteint dans le premier cas avec environ $N = 5000$ bits, contre $N = 20000$ bits dans le second cas.

• Figures 3.12 et 3.13 :
Nous constatons que les algorithmes convergent beaucoup plus vite que précédemment. Cela est certainement lié au nombre d'équations qui est, dans ce cas de figure, plus élevé : l'information que l'on peut espérer obtenir sur chacun des bits est d'autant plus fine que le nombre d'équations est élevé. Par ailleurs, l'effet de « saturation » que nous observions sur la figure 3.10 disparaît lorsque le nombre d'équations augmente : la convergence est possible dès lors que la longueur d'observation est suffisamment importante.

• Figures 3.15 :
Cette figure n'est pas sans rappeler la figure 3.10 sur laquelle nous observions déjà le phénomène de saturation : le nombre moyen d'équations de parité (obtenues par élévation au carré) saturant avec la longueur d'observation, l'algorithme ne converge pas, même si l'on rallonge la séquence observée. Ici, le phénomène de saturation est observable avec la fonction f_1 à $n = 5$ entrées, fonction pour laquelle la probabilité d'erreur est faible alors que nous ne l'observions pas sur la figure 3.9. Cela est dû au poids des équations de parité qui vaut $d = 5$: le biais engendré lors du calcul de l'extrinsèque est moins conséquent que dans le cas où $d = 3$ (le contexte expérimental étant identique).
Cette figure, ainsi que la figure 3.10, montre une grande différence de comportement entre le canal binaire symétrique et la fonction booléenne ; nous concluons de ces deux ensembles de résultats que l'écart entre fonction booléenne et BSC est lié à la probabilité de transition, le poids des équations et le nombre d'équations de parité par bit.

• Figures 3.17 :
L'algorithme Canteaut-Trabbia générant davantage d'équations de parité, nous observons sur cette figure que l'algorithme de Gallager converge sur la fonction f_1 ; la différence entre canal binaire symétrique et fonction booléenne est, ici aussi, de l'ordre de quelques centaines de bits.

Dans cette partie, nous avons ainsi tenté d'évaluer les différences en terme de performances de différents algorithmes de cryptanalyse du registre filtré. Tous ces algorithmes reposent sur l'existence d'équations de parité liant les bits de la séquence d'entrée de la fonction et s'inspirent de l'algorithme de décodage des codes à faibles densité, initialement introduit par Gallager en 1962.

Nous avons tenté d'étendre le principe de cet algorithme itératif en utilisant la table des valeurs de la fonction. Les contraintes utilisées sont non plus des équations de parité, mais des vecteurs d'entrées de la fonction booléenne. L'intérêt est alors, dans certains cas, de pouvoir s'affranchir de la recherche d'équations de parité de poids faibles qui représente une contrainte fondamentale dans de tels systèmes.
Nous développons cet algorithme dans le paragraphe suivant.

3.2 Décodage itératif de la fonction booléenne : l'algorithme IDBF (Iterative Decoding of a Boolean Function)

Nous présentons dans cette partie un algorithme itératif de décodage de la fonction booléenne. S'inspirant du principe de l'algorithme de Gallager, cet algorithme a pour but de décoder la séquence de clé en utilisant la séquence reçue (z) et un ensemble de contraintes qui sont ici représentées par la table de la fonction booléenne. Cet algorithme pourra ensuite être associé à d'autres contraintes, telles que les équations de parité, pour améliorer le décodage.

Le parallèle avec l'algorithme Belief Propagation provient de l'observation suivante : considérons un bit a_t de la séquence d'entrée. Si les espacements sont quelconques, ce bit appartient à n vecteurs distincts ne se chevauchant pas. L'appartenance à ces n vecteurs peut être considérée comme une contrainte : dans le cas de l'algorithme B.P., les contraintes sont les équations de parité ; dans le cas de notre algorithme, les contraintes seront l'appartenance d'un bit à n vecteurs d'entrée de la fonction.
Comment allons-nous générer de l'information à partir de ces « nouvelles » contraintes ? Rappelons que la table de vérité de la fonction est connue : nous allons en extraire de l'information sur les bits de la séquence.

Nous débutons cette partie par une description de l'algorithme baptisé IDBF (Iterative Decoding of a Boolean Function). Nous allons ensuite montrer comment, en associant cet algorithme à d'autres algorithmes de décodage, celui de Gallager notamment, nous pourrons utiliser de façon efficace la connaissance du système.

3.2.1 Description de l'algorithme IDBF

L'algorithme de décodage itératif de la fonction se déroule de façon similaire à l'algorithme de Gallager utilisant des équations de parité.

Supposons que nous ayons à notre disposition des probabilités *a priori* sur les bits de la séquence (a), que nous noterons $\pi(a_t)$. Soit $\mathcal{X}(t)$ l'ensemble des vecteurs d'entrée auxquels le bit a_t appartient. Il est aisé de vérifier que

$$\mathcal{X}(t) = \left\{ \mathbf{X}(t), \mathbf{X}(t-\lambda_0), \mathbf{X}(t-\lambda_0-\lambda_1), \ldots, \mathbf{X}(t-\sum_{i=0}^{n-2}\lambda_i) \right\} .$$

Rappelons que les λ_i représentent les espacements entre les entrées de la fonction relativement au registre à décalage dans lequel chacune des composantes du vecteur d'entrée est prélevée.
De la même façon que dans l'algorithme de Gallager, nous écrivons :

$$P(a_t = 1 \mid f, z) \propto \prod_{\mathbf{X} \in \mathcal{X}(t)} P(a_t = 1 \text{ dans } \mathbf{X}(t) \mid f, z) \tag{3.36}$$

et nous allons évaluer $P(a_t = 1 \text{ dans } \mathbf{X}(t) \mid f, z)$ en utilisant l'information apportée par les bits qui constituent le vecteur $\mathbf{X}(t)$ et qui diffèrent de a_t. Afin de conserver une structure aussi arborescente que possible, nous allons même, comme dans l'algorithme de Gallager, introduire une information extrinsèque par bit et par vecteur, ainsi qu'une APP partielle par bit et par vecteur.

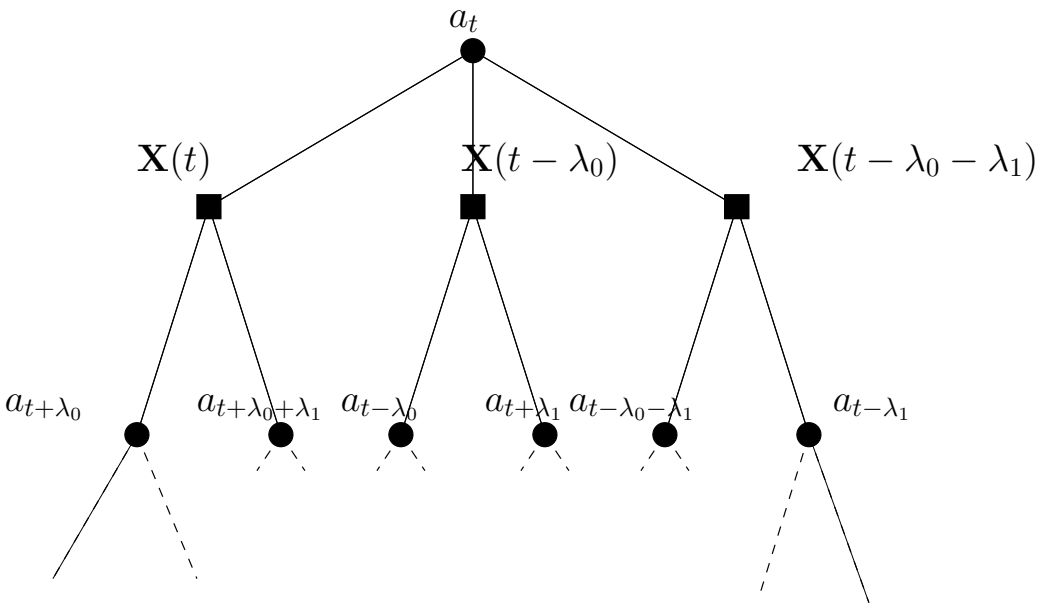


Fig. 3.18 – Représentation de l'algorithme IDBF sous la forme d'un arbre.

Nous introduisons donc :

- $Extr_m(a_t)$: il s'agit de la probabilité que le bit a_t soit égal à '1' dans son m-ième vecteur d'entrée, $m \in [1, n]$; cette probabilité est calculée à partir des informations dont on dispose sur les autres bits qui constituent le m-ième vecteur d'entrée de a_t ;
- $A_m(a_t)$ est la probabilité que le bit a_t soit égal à '1' dans l'ensemble de ses vecteurs d'entrée, excepté le m-ième.

La figure 3.18 illustre la représentation sous forme de graphe de l'algorithme IDBF : les noeuds ronds représentent les bits, comme dans le cas de la figure 3.4. Les noeuds carrés représentent quant à eux les vecteurs d'entrée de la fonction booléenne. L'information est propagée de bas en haut dans l'arbre. L'information extrinsèque est calculée sur un vecteur d'entrée de la fonction et propagée vers un noeud bit pour permettre le calcul de la probabilité *a posteriori*.

La proposition qui suit nous permet de calculer efficacement les informations extrinsèques de chacun des bits de la séquence.

Proposition 20 *Soit $\mathbf{X} = (x_1, \ldots, x_n)$ le vecteur d'entrée d'une fonction booléenne. Supposons que les bits a_i soient indépendants et posons*

$$\begin{aligned} \forall 1 \leq i \leq n, \quad p_i &= P(x_i = 1) && (3.37)\\ c_i &= 1 - 2p_i && (3.38)\\ \mathbf{c}(i) &= (c_1, \ldots, c_{i-1}, 1, c_{i+1}, \ldots, c_n) \\ \overline{\mathbf{c}}(i) &= (c_1, \ldots, c_{i-1}, -1, c_{i+1}, \ldots, c_n)\,. \end{aligned}$$

Alors, si le bit reçu est un 1,

$$P\Big(x_i = 1 \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n\Big) = \frac{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\overline{\mathbf{c}}(i)^{\mathbf{t}}}{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\mathbf{c}(i)^{\mathbf{t}} + \sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\overline{\mathbf{c}}(i)^{\mathbf{t}}} \qquad (3.39)$$

si le bit reçu est un 0, en posant $\overline{f} = 1 + f \mod 2$

$$P\Big(x_i = 1 \mid x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n \Big) = \frac{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{\overline{f}}(\mathbf{t})\overline{\mathbf{c}}(i)^{\mathbf{t}}}{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{\overline{f}}(\mathbf{t})\mathbf{c}(i)^{\mathbf{t}} + \sum_{\mathbf{t}\in\{0,1\}^n} \widehat{\overline{f}}(\mathbf{t})\overline{\mathbf{c}}(i)^{\mathbf{t}}} \tag{3.40}$$

où

$$\forall \mathbf{x},\ t \in \{0,1\}^n,\ \mathbf{x^t} \ = \ \prod_{i=1}^{n} x_i^{t_i} \quad \textit{avec la convention } \mathbf{0^0} = 1\ .$$

Preuve :
Nous supposons dans le cadre de cette démonstration que le bit reçu, correspondant au filtrage du vecteur $\mathbf{X}$ par la fonction f est un '1'. La démonstration pour le cas complémentaire du '0' reçu est identique à celle qui suit : il suffit d'y remplacer f par $\overline{f}$.

Évaluons la probabilité que le bit x_i soit égal à '0' dans un des vecteurs auxquels il appartient, vecteur qui, rappelons-le, appartient à l'image réciproque de '1' par hypothèse.

$$P(x_i = 0 \mid (x_j)_{j\neq i}) = \sum_{\mathbf{x} \mid f(\mathbf{x})=1} \mathbb{1}_{\{x_i=0\}}(\mathbf{x}) \prod_{j\neq i} [p_j x_j + (1-p_j)(1-x_j)].$$

Les termes sous le produit doivent être interprétés de la façon suivante : si $x_j = 1$ alors c'est $p_j = P(x_j = 1)$ qui intervient, sinon c'est $1 - p_j = P(x_j = 0)$. On peut réécrire chaque terme du produit de façon équivalente :

$$\begin{aligned} p_j x_j + (1-p_j)(1-x_i) &= \frac{1 + (-1)^{x_j} c_j}{2} \\ &= \frac{1 + (-1)^{\mathbf{u}_j \cdot \mathbf{x}} c_j}{2} \end{aligned}$$

où $\mathbf{u}_j$ est le j-ième vecteur canonique, c'est-à-dire le vecteur dont toutes les composantes sont nulles, la j-ième exceptée.
En posant $c_i = 1$, on obtient alors :

$$\begin{aligned} P(x_i = 0 \mid (x_j)_{j\neq i}) &= \sum_{\mathbf{x}} f(\mathbf{x}) \prod_j \frac{1 + (-1)^{\mathbf{u}_j \cdot \mathbf{x}} c_j}{2} \\ &= \frac{1}{2^n} \sum_{\mathbf{x}} f(\mathbf{x}) \times \Big[1 + \sum_{k=1}^{n} \sum_{1\leq j_1 < \ldots < j_k \leq n} (-1)^{(\mathbf{u_{j_1}} + \ldots + \mathbf{u_{j_k}}).\mathbf{x}} c_{j_1} \ldots c_{j_k}\Big] \\ &\propto \sum_{k=1}^{n} \Big[1 + \sum_{1\leq j_1 < \ldots < j_k \leq n} c_{j_1} \ldots c_{j_k} \sum_{\mathbf{x}} f(\mathbf{x}) (-1)^{(\mathbf{u_{j_1}} + \ldots + \mathbf{u_{j_k}}).\mathbf{x}}\Big]. \end{aligned}$$

Or d'après la définition des $\mathbf{u_j}$, la somme $\mathbf{u_{j_1}} + \ldots + \mathbf{u_{j_k}}$ décrit l'espace vectoriel $\{0,1\}^n$ excepté le vecteur nul dans l'expression ci-dessus. Le vecteur nul peut être introduit dans le terme égal à 1 afin d'aboutir à l'expression suivante :

$$P(x_i = 0 \mid (x_j)_{j\neq i}) \propto \sum_{\mathbf{t}} \mathbf{c}^{\mathbf{t}} \widehat{f}(\mathbf{t})$$

et donc

$$P(x_i = 1 \mid (x_j)_{j\neq i}) \propto \sum_{\mathbf{t}} \overline{\mathbf{c}}^{\mathbf{t}} \widehat{f}(\mathbf{t})$$

ce qui, après normalisation par la somme de ces deux dernières quantités, donne le résultat attendu. ∎

En notant $\pi(a_t)$ les probabilités *a priori* dont nous disposons sur les bits de la séquence (a), l'algorithme est décrit dans le tableau 3.10.

Algorithme IDBF

1. ***Initialisation :***
   $\forall t < N,\ \ \forall m\ \in [1,n]\ , Extr_m(a_t) = \pi(a_t).$
2. ***Déroulement :***
   $\forall t < N,\ \forall m \in [1,n]$
   (a) $A_m^{(j)}(a_t) = \prod_{i \neq m} Extr_i^{(j-1)}(a_t)$
   (b) $Extr_m^{(j)}(a_t) \propto \sum_{\mathbf{t} \in \{0,1\}^n} \widehat{\overline{f}}(\mathbf{t}) \overline{\mathbf{c}}(j)^{\mathbf{t}}$ où

$$\begin{aligned} \overline{\mathbf{c}}(j) &= (c_1, \ldots, c_n) \\ c_j &= -1 \\ \text{et } \forall i \neq j,\ c_i &= 1 - 2A_m^{(j)}(a_i). \end{aligned}$$

3. ***Terminaison :***
   $\forall t < N,\ APP^{(nb_{iter})}(a_t) = \prod_{i=1}^{n} Extr_i^{(nb_{iter})}(a_t).$

Tab. 3.10 – Description de l'algorithme IDBF.

Évaluons la complexité de l'algorithme IDBF : le calcul de la transformée de Fourier de la fonction peut être fait en $n2^n$ opérations. L'étape (2a) de l'algorithme nécessite $\mathcal{O}(Nn \times n)$ opérations à chaque itération ; l'étape (2b) nécessite $\mathcal{O}(Nn \times 2^n n)$ opérations. D'où une complexité globale en

$$\mathcal{O}\left(n2^n + nb_{iter}\ \times Nn \times [n + n2^n]\right) \approx \mathcal{O}\left(nb_{iter}\ N\right) \tag{3.41}$$

l'approximation étant vérifiée lorsque $n \ll N$.

Tout comme dans l'algorithme B.P., nous pouvons travailler sur des APP totales au lieu de travailler sur des APP partielles, afin de réduire la complexité de l'algorithme. Nous avons observé en pratique que cela ne changeait quasiment pas les résultats. Les résultats donnés dans les tableaux 3.11, 3.12 et 3.13 ont été obtenus en itérant sur l'APP totale.

L'algorithme IDBF peut être utilisé seul ou en combinaison avec d'autres algorithmes. Dans le paragraphe suivant, nous étudions l'algorithme IDBF utilisé seul. Nous étendons ensuite son champ d'application en le couplant à un décodeur Belief Propagation (sur des équations de parité) : l'association de ces deux décodeurs permet d'améliorer les performances de cryptanalyse.

3.2.2 Performances de l'algorithme IDBF seul

L'utilisation de l'algorithme IDBF seul revient à considérer que nous ne possédons aucune information *a priori* sur les bits de la séquence à décoder. Dans l'algorithme décrit dans le tableau 3.10, cela revient à poser

$$\forall t < N,\ \pi(a_t) = 0.5\ .$$

Observons alors ce qui se passe à la première itération : considérons par exemple

$$\mathbf{X}(t) = (a_t, a_{t+\lambda_0}, \ldots, a_{t+\sum_i \lambda_i})$$

et supposons que le bit reçu après filtrage soit un 1.
Intéressons-nous par exemple au premier bit de $\mathbf{X}(t)$, *i.e.* a_t.
D'après l'équation (3.39), on a :

$$Extr_1^{(1)}(a_t) = \frac{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\overline{\mathbf{c}}(1)^{\mathbf{t}}}{\sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\mathbf{c}(1)^{\mathbf{t}} + \sum_{\mathbf{t}\in\{0,1\}^n} \widehat{f}(\mathbf{t})\overline{\mathbf{c}}(1)^{\mathbf{t}}}. \tag{3.42}$$

Or

$$\forall i,\ \pi(a_i) = 0.5 \Longleftrightarrow \forall i,\ c_i = 0$$

donc $\overline{\mathbf{c}}(j)^{\mathbf{t}} \neq 0 \Longrightarrow \mathbf{t} = (*, 0, \ldots, 0)$ puisque par convention $\mathbf{0^0} = 1$ et que l'on a posé $\pi(a_i) = 0.5$, soit $\overline{\mathbf{c}}(j) = (-1, 0, 0, \ldots, 0)$. Alors, l'équation (3.42) devient :

$$\begin{aligned} Extr_1^{(1)}(a_t) &= \frac{\widehat{f}(0,\ldots,0) - \widehat{f}(1,0,\ldots,0)}{2\widehat{f}(0,\ldots,0)} \\ &= \frac{1}{2} - \frac{\widehat{f}(1,0,\ldots,0)}{2\widehat{f}(0,\ldots,0)}\ . \end{aligned}$$

Nous pouvons donc énoncer la proposition suivante :

Proposition 21 *Si la fonction booléenne de filtrage, f, est résiliente (quel que soit son ordre de résilience $t > 0$), l'algorithme IDBF seul n'est d'aucune utilité.*

Preuve :
Si la fonction est t-résiliente, $t > 0$, alors $\forall \mathbf{u}$ tel que $0 < w_H(\mathbf{u}) \leq 1,\ \widehat{f}(\mathbf{u}) = 0$ et on a donc

$$Extr_1^{(1)}(a_t) = \frac{1}{2}$$

et l'algorithme ne peut démarrer : il reste à l'étape initiale. ∎

Cette propriété vient conforter **l'utilité des fonctions résilientes dans le cas de registres filtrés**, comme nous l'avions déjà soulignée au paragraphe 3.1.8.

Faisons maintenant l'hypothèse que la fonction est non résiliente équilibrée. Nous remarquons qu'à la première itération, l'extrinsèque générée par le vecteur $\mathbf{X}(t)$ sur son premier bit, a_t, prend la valeur suivante :

$$Extr_1^{(1)}(a_t) = \frac{1}{2} - \frac{\widehat{f}(\mathbf{u}_1)}{2^n} \ .$$

Il s'agit de la probabilité issue de la corrélation entre la forme linéaire $\mathbf{x} \longrightarrow \mathbf{u}_1\mathbf{x}$ et la sortie de la fonction. L'APP totale du bit à l'issue de la première itération correspond donc au produit des probabilités issues des différentes corrélations entre les formes linéaires de poids 1 et la sortie. Ceci est à rapprocher de la remarque faite au paragraphe 3.1.8, dans lequel nous avions testé différents types d'initialisation du décodage de Gallager.

L'algorithme IDBF décrit dans le tableau 3.10 génère des APP. Comment allons-nous utiliser la sortie de cet algorithme ?
Des probabilités *a posteriori* obtenues sur chacun des bits de la séquence, nous allons déduire une probabilité sur $(a^{\ell_\mathbf{u}})$, la séquence obtenue en appliquant $\ell_\mathbf{u}$ à la séquence (a), où cette forme linéaire $\ell_\mathbf{u}$ sera choisie en fonction des propriétés de la fonction, c'est-à-dire en fonction de son spectre.
À partir de cette séquence, nous générons des probabilités sur la séquence $(a^{\ell_\mathbf{u}})$ en utilisant le lemme de Gallager.
Écrivons $a^{\ell_\mathbf{u}} = a_{j_1} + \ldots + a_{j_h}$.
Le lemme 1 de Gallager nous donne :

$$P(a^{\ell_\mathbf{u}} = 1) = \frac{1 - \prod_{i=1}^{h} \Big(1 - 2APP(a_{j_i})\Big)}{2} \ .$$

Nous sélectionnons ensuite les K bits les plus fiables et linéairement indépendants, leur faisons subir un décodage à seuil et inversons le système linéaire alors obtenu.

Nous présentons ci-après quelques résultats de simulation, obtenus sur des fonctions non résilientes, dont nous précisons la table soit directement, soit en annexe. Dans toutes ces expériences, le polynôme de rétroaction utilisé est $g(x) = 1 + x^{37} + x^{100}$.

La première série de résultats présentés dans la table 3.11 a été obtenue sur une fonction à quatre entrées dont la table est

$$0101010011110100 \ .$$

Quant aux tableaux de résultats 3.12 et 3.13, ils ont été obtenus sur les fonctions f_5 et f_9 respectivement.

N	**Proba d'erreur minimale**
1000	$< 8\ .10^{-5}$ sur 100% des tirages
800	1.10^{-4} sur 100% des tirages
500	$1,7.10^{-4}$ sur 100% des tirages
200	7.10^{-3} sur 100% des tirages

Tab. 3.11 – Probabilité d'erreur obtenue en appliquant l'IDBF seul à une fonction à $n = 4$ entrées (0101010011110100) ; $K = 100$.

N	Proba d'erreur minimale
1000	7.5% sur 71.7% des tirages
2000	6.4% sur 94% des tirages
5000	4.8% sur 99.9% des tirages

Tab. 3.12 – Probabilité d'erreur obtenue en appliquant l'IDBF seul à la fonction f_5 à $n = 5$ entrées ; $K = 100$.

N	Proba d'erreur minimale
15000	7.9% sur 80.3% des tirages
20000	7.6% sur 84.9% des tirages
25000	7.5% sur 89.4% des tirages

Tab. 3.13 – Probabilité d'erreur obtenue en appliquant l'IDBF seul à la fonction f_9 à $n = 6$ entrées ; $K = 100$.

L'efficacité de cet algorithme est cependant freinée par l'apparition de cycles : les informations générées après les deux premières itérations ne sont en effet plus indépendantes et la structure arborescente du code (voir paragraphe 3.1.2) qui justifie les itérations, est brisée .

Illustrons notre propos sur un exemple :

Exemple 5 *Considérons une fonction booléenne à trois entrées et examinons l'arborescence partant du bit a_t de la séquence.*
Au premier niveau apparaissent les trois vecteurs

$$\begin{aligned} \mathbf{X}(t) &= (a_t, a_{t+\lambda_0}, a_{t+\lambda_0+\lambda_1}) \\ \mathbf{X}(t-\lambda_0) &= (a_{t-\lambda_0}, a_t, a_{t+\lambda_1}) \\ \mathbf{X}(t-\lambda_0-\lambda_1) &= (a_{t-\lambda_0-\lambda_1}, a_{t-\lambda_1}, a_t) \end{aligned}$$

Chacun des nouveaux bits apparus appartient aux vecteurs suivants :
- *bits fils de $\mathbf{X}(t)$:*
  - $a_{t+\lambda_0}$ *appartient à* $\mathbf{X}(t+\lambda_0)$ *et* $\mathbf{X}(t-\lambda_1)$ *;*
  - $a_{t+\lambda_0+\lambda_1}$ *appartient à* $\mathbf{X}(t+\lambda_0+\lambda_1)$ *et* $\mathbf{X}(t+\lambda_1)$.
- *bits fils de $\mathbf{X}(t-\lambda_0)$:*
  - $a_{t-\lambda_0}$ *appartient à* $\mathbf{X}(t-2\lambda_0)$ *et* $\mathbf{X}(t-2\lambda_0-\lambda_1)$ *;*
  - $a_{t+\lambda_1}$ *appartient à* $\mathbf{X}(t+\lambda_1)$ *et* $\mathbf{X}(t-\lambda_0+\lambda_1)$.
- *bits fils de $\mathbf{X}(t-\lambda_0-\lambda_1)$:*
  - $a_{t-\lambda_0-\lambda_1}$ *appartient à* $\mathbf{X}(t-\lambda_1)$ *et* $\mathbf{X}(t+\lambda_0-\lambda_1)$ *;*
  - $a_{t-\lambda_1}$ *appartient à* $\mathbf{X}(t-2\lambda_0-\lambda_1)$ *et* $\mathbf{X}(t-2\lambda_0-2\lambda_1)$.

Ainsi, à ce niveau du décodage, les vecteurs $\mathbf{X}(t-\lambda_1)$, $\mathbf{X}(t+\lambda_1)$,$\mathbf{X}(t-2\lambda_0-\lambda_1)$ apparaissent deux fois, d'où l'existence de cycles.

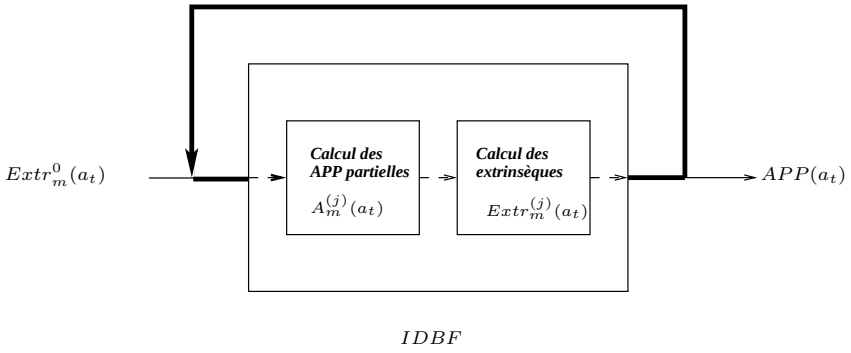


Fig. 3.19 – Principe de l'IDBF, décodeur itératif de la fonction booléenne.

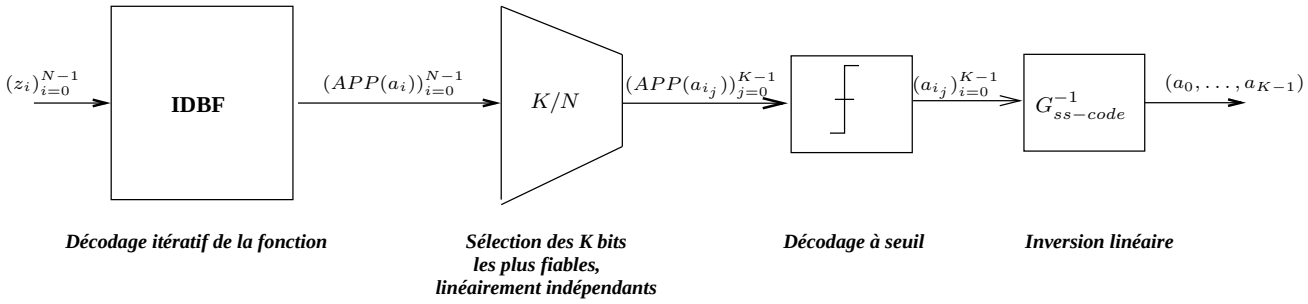


Fig. 3.20 – Schéma complet de cryptanalyse à partir de l'IDBF.

Ces cycles sont à rapprocher directement de ceux qui apparaissent lors de l'algorithme SOJA-itératif (voir chapitre 5, dans le paragraphe 5.3.3 page 179) ; certaines informations vont intervenir plus que les autres, ce qui biaise les résultats.

3.2.3 Utilisation conjointe de l'IDBF et de l'algorithme B.P.

L'algorithme de décodage itératif de la fonction peut également être intégré dans un schéma de décodage itératif global, comme nous l'avons représenté sur la figure 3.21. Ce schéma de décodage est constitué de deux sous-blocs de décodage : un bloc « IDBF » et un bloc « décodage de Gallager » utilisant les équations de parité. Les probabilités *a posteriori* générées par chacun des blocs sont alors passées en probabilités *a priori* à l'autre bloc. À la fin des itérations, nous conservons les K bits les plus fiables qui sont linéairement indépendants et nous leur faisons subir un décodage à seuil. Cette configuration ressemble fort à celle du turbo-décodage présentée en annexe (cf. annexe E). La différence essentielle réside dans la structure des codes : les codes que nous décodons séquentiellement ici sont de structure très différente contrairement au cas turbo, qui décode deux codes convolutifs souvent identiques (dans le cas des turbo-codes parallèles notamment) séparés par un entrelaceur.

Nous présentons les résultats de simulation obtenus en appliquant l'algorithme IDBF au système constitué d'un registre à décalage ayant pour polynôme de rétroaction $g(x) = 1 + x^{37} + x^{100}$. Les équations ont été générées en utilisant la méthode d'élévations au carré successives, décrite dans le paragraphe 2.2.1 du chapitre 2 : le nombre d'équations moyen par bit est donc

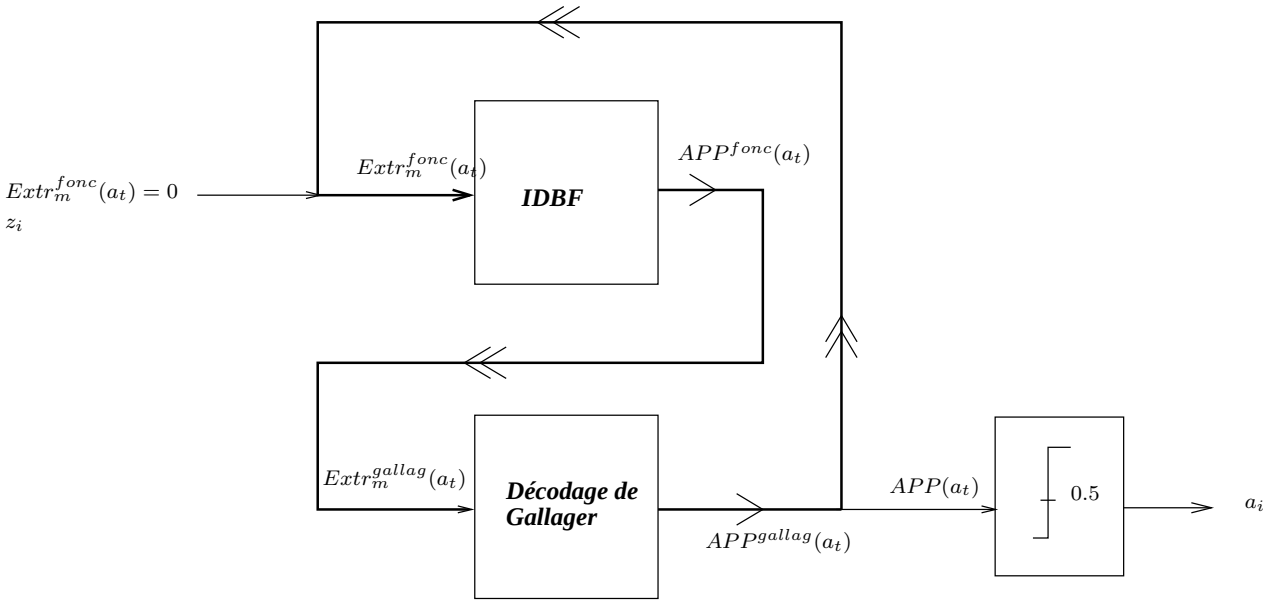


Fig. 3.21 – Décodage IDBF-Gallager.

relativement faible. La fonction utilisée est la fonction à $n = 6$ entrées f_9 décrite en annexe.

Le contexte expérimental est, à dessein, strictement identique à celui correspondant aux résultats figurant dans le tableau 3.13, obtenus en utilisant l'algorithme IDBF seul. Alors que les résultats du tableau 3.13 étaient décevants, les performances présentées dans la table 3.14 apparaissent nettement meilleures : une longueur de $N = 800$ bits suffit à cryptanalyser le système, avec un taux de succès supérieur à 90%. L'utilisation des équations de parité avec l'algorithme IDBF permet donc, dans ce cas de figure, d'améliorer considérablement les performances de l'IDBF. Tempérons cependant notre enthousiasme : nous avons simulé des systèmes régis par

N	**Proba d'erreur minimale**
2500	$< 10^{-5}$ sur 99.7% des tirages
1500	$< 4.5\ .10^{-5}$ sur 98.4% des tirages
1000	$< 5\ .10^{-5}$ sur 97% des tirages
800	$< 9\ .10^{-4}$ sur 91% des tirages

Tab. 3.14 – Résultats de l'IDBF-Gallager, $K = 100$, poids des équations $d = 3$.

peu de contraintes linéaires, qui étaient cette fois de poids $d = 5$. Ayant constaté précédemment (voir paragraphe 3.1.8, page 93) que les résultats obtenus à l'aide de l'algorithme B.P. version δ sur de tels systèmes étaient assez mauvais, nous souhaitions les améliorer en utilisant l'IDBF. En vain : les simulations que nous avons effectuées ont montré que l'introduction de l'algorithme IDBF n'a engendré aucune amélioration significative des résultats.

3.2.4 Perspective : l'algorithme Gallager-IDBF appliqué à une fonction résiliente

À première vue, l'application de l'algorithme IDBF-Gallager à une fonction résiliente paraît impossible : d'après la proposition 21 (cf. page 101), l'IDBF ne peut générer aucun biais sur les bits de la séquence (a) si la fonction est résiliente. L'algorithme de Gallager appliqué tel quel n'en générera pas davantage puisque, les observations étant initialisées à $1/2$, l'algorithme ne peut démarrer (voir page 67). Cependant, en utilisant l'algorithme de Gallager de façon légèrement différente, nous pensons qu'il est possible d'utiliser l'algorithme IDBF-Gallager avec une fonction résiliente.
Expliquons plus avant comment nous suggérons de procéder (nous n'avons pas testé cette méthode de décodage).

Considérons une fonction booléenne f, t-résiliente. Cette fonction n'a aucune corrélation avec des formes linéaires de poids 1, par définition. Cependant, il existe au moins une forme linéaire de poids supérieur à t qui a une corrélation non nulle avec la sortie de la fonction en vertu de l'égalité de Parseval (voir chapitre 1). Intéressons-nous à cette forme linéaire, que nous appellerons ℓ dans la suite.
ℓ a une écriture polynomiale : $\ell(x) = \sum_{i=0}^{K-1} \ell_i x^i$ où $\ell_i \in GF(2)$ et $\ell_i \neq 0 \Longrightarrow \exists j \ : i = \sum_{h=0}^{j} \lambda_h$.
En effet, ℓ ne concerne que les entrées de la fonction. Or, son écriture polynomiale s'étend sur toutes les valeurs possibles de i comprises entre 0 et $K-1$, *i.e.* sur toutes les « cases mémoires » du LFSR.
Le polynôme de rétroaction qui régit le LFSR est de degré K et souvent, par hypothèse, irréductible sur $GF(2)$. D'après le théorème 1 énoncé page 4, l'ensemble des polynômes en x de degré strictement inférieur à K à coefficients dans $GF(2)$ forment un corps. ℓ est donc inversible.

Nous suggérons alors de procéder de la façon suivante :

1. Dans un premier temps, grâce à l'algorithme B.P., nous générons des probabilités sur les bits de la séquence (a^ℓ). Cette séquence ayant une corrélation non nulle avec la sortie, l'algorithme B.P. itéré va permettre de générer de telles informations.

2. Ensuite, en utilisant le caractère inversible de ℓ, nous exprimons la séquence (a) comme une combinaison linéaire des bits de la séquence (a^ℓ) : si $\Psi(x) = \sum_{i=0}^{K} \psi_i x^i$ est l'inverse de ℓ modulo g, alors a s'exprime en fonction de a^ℓ de la façon suivante :

$$\forall t > K,\ a_t = \sum_{i=0}^{K} \psi_i a^\ell_{t+i} \ .$$

3. Le lemme de Gallager (voir page 64) nous permet alors d'évaluer la probabilité que chaque bit de la séquence (a) soit égal à 1 en fonction des probabilités de la séquence (a^ℓ) générées à l'étape 1.

4. Munis de ces probabilités sur la séquence initiale (a), nous pouvons alors appliquer l'algorithme IDBF, éventuellement couplé à un algorithme B.P. utilisant les équations de parité.

Nous pensons ainsi contourner l'obstacle que constitue la résilience (voir proposition 21 page 101) et réussir à initialiser de façon pertinente l'algorithme IDBF.

3.3 Conclusion

Nous avons dans ce chapitre, présenté l'algorithme de Gallager (également appelé algorithme Belief Propagation) dans sa version originale. Cet algorithme ne pouvant généralement pas être appliqué tel quel au système du registre filtré, nous nous sommes intéressés à des versions simplifiées, moins complexes. Souhaitant évaluer leurs performances, nous avons simulé ces différents algorithmes pour constater que la version δ de l'algorithme B.P. était le meilleur compromis complexité-performances. Nous avons donc choisi de nous concentrer sur cet algorithme pour comparer les performances du décodage itératif appliqué à une séquence bruitée par un canal binaire symétrique d'une part, à une séquence en sortie d'une fonction booléenne d'autre part. Dans tous les cas, le modèle du canal BSC s'est révélé plus favorable que le filtrage par une fonction booléenne : la différence peut cependant être aisément comblée par l'augmentation de la longueur de chiffré de quelques milliers de bits, sauf dans quelques systèmes « pathologiques » pour lesquels le nombre d'équations de parité est insuffisant devant la probabilité de transition du canal équivalent et le poids des équations de parité. L'importance du poids et du nombre d'équations dont nous disposons est de façon générale, très visible : plus le poids est faible et plus le nombre d'équations est grand, meilleurs sont les résultats.

Dans une deuxième partie, nous avons présenté un algorithme de décodage de la fonction booléenne : cet algorithme est très approximatif dans la mesure où des cycles apparaissent rapidement dans l'arborescence du code, mais il permet toutefois de cryptanalyser certaines fonctions booléennes sans utiliser d'équations de parité. Il peut également être utilisé en combinaison avec des équations de parité dans une structure « turbo » pour améliorer les performances du décodage itératif.

Chapitre 4

Cryptanalyse du registre filtré via l'utilisation d'un treillis

La structure très particulière d'un unique registre filtré par une fonction booléenne engendre un effet mémoire sur les vecteurs d'entrée de la fonction. Par exemple, dans le cas précis où les n entrées de la fonction sont régulièrement espacées (relativement au LFSR), d'une valeur égale à λ, les vecteurs $\mathbf{X}(t)$ et $\mathbf{X}(t+\lambda)$ ont $n-1$ bits en commun, quel que soit t. Toute connaissance sur le vecteur d'entrée $\mathbf{X}(t)$ peut donc se répercuter en une connaissance sur le vecteur $\mathbf{X}(t+\lambda)$ et cette transmission d'information peut être itérée.

Un treillis (cf. [95]) semble être l'outil adapté à la représentation du code de la séquence PN, dont la structure est relativement proche de celle d'un code convolutif (cf. annexe C), et à la propagation d'information associée à ce code. Dans ce chapitre, nous présentons principalement deux algorithmes utilisant un treillis. Nous nous intéressons dans un premier temps au cas d'un système dans lequel les entrées peuvent être identifiées à une structure régulière. Alors, l'algorithme que nous présentons et les différentes versions qui en dérivent permettent, sauf cas exceptionnel, d'obtenir la clé secrète sans aucune probabilité d'erreur, à partir d'une longueur d'observation relativement faible. Ces algorithmes appartiennent à la famille des algorithmes déterministes décrits précédemment au chapitre 2 (cf. paragraphe 2.1).
Dans un deuxième temps, l'hypothèse d'une structure régulière étant assez restrictive, nous avons étendu le domaine d'application de l'algorithme du treillis à des systèmes constitués d'entrées espacées de façon quelconque. Ces algorithmes seront associés à une probabilité d'erreur et rentrent donc dans le cadre des attaques que nous avons appelées probabilistes au chapitre 2 (cf. paragraphe 2.2).

4.1 Algorithme du treillis déterministe

Nous allons, dans ce paragraphe, présenter des algorithmes utilisant un treillis et qui permettent de retrouver la clé secrète sans erreur. Nous commençons par une approche intuitive, puis présentons une première version de l'algorithme du treillis, qui cherche à identifier des bits constants sur un sous-ensemble de vecteurs d'entrée de la fonction. Le principe de cet algorithme sera ensuite étendu à la recherche de formes linéaires constantes. Par ailleurs, le parcours du treillis pourra être inversé, et le treillis parcouru dans les deux sens : les deux dernières versions de l'algorithme du treillis tiennent compte de toutes ces remarques.

Ayant présenté tous ces algorithmes, nous avons cherché à évaluer le nombre moyen d'états compatibles avec la structure du générateur à décalage et l'observation de la sortie de la fonction,

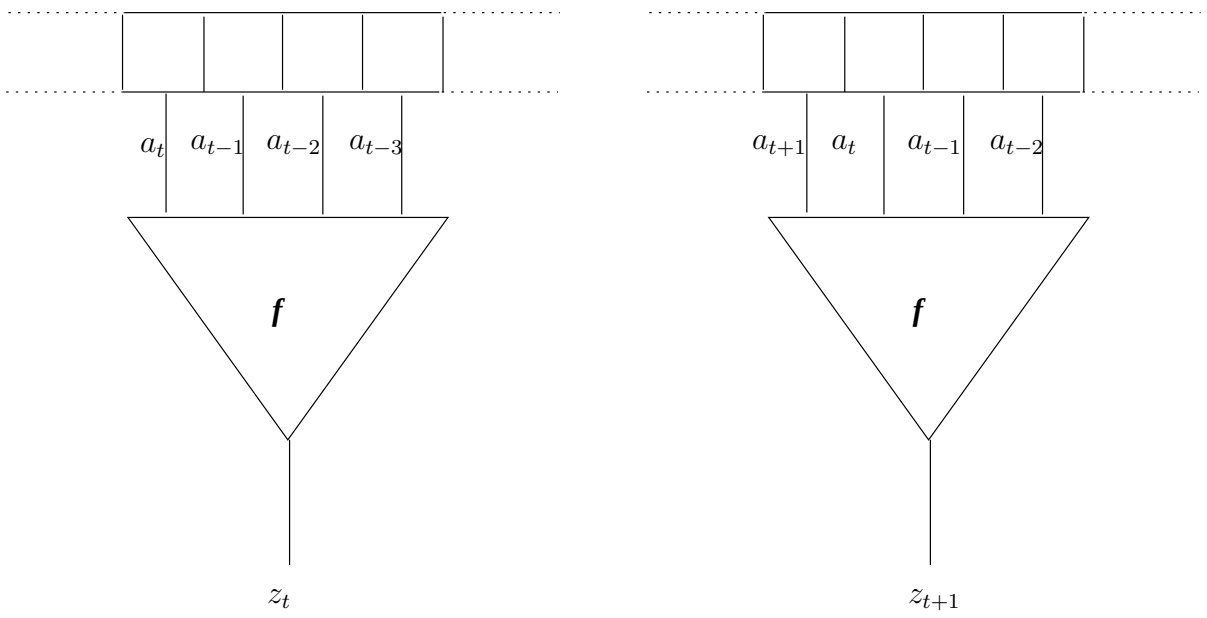


Fig. 4.1 – Évolution du vecteur d'entrée d'une fonction à $n = 4$ entrées entre deux instants consécutifs lorsque les espacements λ_i sont tous égaux à 1.

que nous avons appelés « survivants ». Puis, nous avons appliqué cet algorithme à un automate plus complexe que le LFSR et présentons les résultats obtenus.

4.1.1 Approche intuitive de l'algorithme du treillis

Nous supposons pour l'instant que les entrées de la fonction sont connectées de façon consécutive dans le générateur de pseudo-aléa, ce qui, en utilisant les notations introduites au premier chapitre, revient à écrire : $\forall i, \lambda_i = 1$. Cette hypothèse peut paraître assez restrictive, mais comme nous l'avons vu au paragraphe 1.2 du chapitre 1, une telle configuration présente l'avantage de pouvoir assurer une bonne complexité linéaire à la suite de bits de clé produite.
Observons la figure 4.1 sur laquelle sont représentés les vecteurs d'entrée d'une fonction à n entrées (ici $n = 4$) à deux instants consécutifs, en supposant que le décalage sur les bits d'état interne est opéré de gauche à droite : $\mathbf{X}(t)$ et $\mathbf{X}(t+1)$. **Les vecteurs $\mathbf{X}(t)$ et $\mathbf{X}(t+1)$ ont $n-1$ bits en commun** ; ainsi, si l'on parvient à affiner la connaissance d'un vecteur d'entrée $\mathbf{X}(t)$, on peut espérer propager, au moins partiellement, cette information sur les vecteurs qui suivent. Nous constatons également que, si la valeur de $\mathbf{X}(t+1)$ est, comme nous venons de le voir, partiellement conditionnée par la valeur de son prédécesseur $\mathbf{X}(t)$, elle l'est aussi par la valeur du bit z_{t+1} observé en sortie de la fonction : $\mathbf{X}(t+1)$ **appartient obligatoirement à l'image réciproque par f de z_{t+1}**.

Pour tout t non nul, le vecteur $\mathbf{X}(t+1)$ doit donc satisfaire conjointement les deux conditions suivantes :

$$(X_2^{t+1}, \ldots, X_n^{t+1}) = (X_1^t, \ldots, X_{n-1}^t) \quad (4.1)$$

$$f\Big(\mathbf{X}(t+1)\Big) = z_{t+1} \ . \quad (4.2)$$

L'utilisation d'un treillis permet de propager les informations sur les vecteurs d'entrée : à chaque

instant, passé ($\mathbf{X}(t-1)$) et présent ($\mathbf{X}(t)$) peuvent être utilisés pour déterminer l'ensemble des états compatibles avec la séquence reçue et la structure du code : les « survivants ».

4.1.2 Description de l'algorithme du treillis

Le treillis est construit de la façon suivante :

- un **état** $\sigma = (\sigma_1, \ldots, \sigma_n)$ dans le treillis est constitué de n bits et coïncide avec un vecteur d'entrée de la fonction. Il existe donc 2^n états possibles ;
- d'après la condition (4.1), les **successeurs** de l'état $\sigma = (\sigma_1, \ldots, \sigma_n)$ ont $n-1$ bits en commun avec σ. Le bit restant peut prendre les valeurs 0 ou 1. Ainsi, tout état σ a deux successeurs dans le treillis, noté $succ_0$ et $succ_1$ selon la valeur du bit restant : $succ_0 = (0, \sigma_1, \ldots, \sigma_{n-1})$ et $succ_1 = (1, \sigma_1, \ldots, \sigma_{n-1})$. Par exemple, si l'on considère une fonction à quatre entrées, les deux successeurs de l'état $\sigma = (0110)$ sont $succ_0(\sigma) = (0011)$ et $succ_1(\sigma) = (1011)$;
- chaque état σ a deux **prédécesseurs**, l'un se terminant par un 0 : $(\sigma_2, \ldots, \sigma_n, 0)$, l'autre se terminant par un 1 :$(\sigma_2, \ldots, \sigma_n, 1)$. On note $pred_j(\sigma)$ le prédécesseur de σ se terminant par j. Par exemple, l'état $\sigma = (0110)$ a pour prédécesseurs : $pred_0(\sigma) = (1100)$ et $pred_1(\sigma) = (1101)$[1] ;
- $\Sigma(t)$ désigne **l'ensemble des survivants** à l'instant t et $\nu(t)$ son **cardinal** : $\nu(t) = \mathrm{card}\Big(\Sigma(t)\Big)$. $\Sigma(t)$ représente l'ensemble des vecteurs d'entrée possibles à l'instant t, plus précisément tous les états qui sont compatibles avec le « passé » (condition (4.1)) et le bit reçu à l'instant t (condition (4.2)). Remarquons que si la fonction est équilibrée, $\Sigma(t)$ peut avoir au plus 2^{n-1} éléments, puisqu'au plus la moitié des états peuvent, filtrés par la fonction, produire le bit observé en sortie (condition (4.2)).

Comment exploiter la connaissance de $\Sigma(t)$?
Notons tout d'abord, que si le cardinal de $\Sigma(t)$ est borné supérieurement par 2^{n-1}, il sera, espérons-le, « nettement » inférieur à cette valeur puisque nous allons utiliser la mémoire du système. Il est cependant peu probable que cet ensemble se réduise au seul état d'entrée de la fonction à l'instant t. Ainsi, pour tirer profit de l'ensemble réduit de survivants, l'idée première est de chercher **des bits constants sur cet ensemble**.
Puisque $\Sigma(t)$ contient tous les états qui satisfont les conditions (4.1) et (4.2), si un bit est constant sur les éléments de $\Sigma(t)$, alors le bit de la séquence (a) qui coïncide avec cette entrée ne peut qu'être égal à cette constante :

$$\exists\ j \text{ tel que } \forall \sigma\ \in\ \Sigma(t),\ \ \sigma_j\ =\ b \in \{0,1\}\ \ \Longrightarrow\ \ a_{t+j} = b.$$

Rappelons que tout bit de la séquence (a) peut s'exprimer comme une combinaison linéaire des K premiers bits de la séquence que l'on cherche à récupérer, où K est la longueur du registre. Ainsi, si l'on réussi à trouver K bits linéairement indépendants dans la séquence, il nous restera à inverser un système $(K \times K)$ pour obtenir les K premiers bits de clé.
L'algorithme est alors décrit dans le tableau 4.1.

Remarquons que cet algorithme n'est pas sans rappeler l'algorithme de Viterbi (cf. [28] et l'annexe D) : la description du treillis est identique. Cependant, notre algorithme ne cherche pas

[1] Notons que ce treillis peut être vu comme un graphe de de Bruijn.

Algorithme du treillis

1. ***Initialisation :***
   à t=0 : éliminer tous les états σ de $\Sigma(t=0)$ tels que $f(\sigma) \neq z_0$.
2. ***Déroulement :***
   à $0 < t < N$:
   (a) Déterminer les successeurs possibles des éléments de $\Sigma(t-1)$. Parmi ces successeurs, ne conserver que les états qui, filtrés par la fonction, produisent le bit observé à l'instant t :

$$\begin{aligned}\Sigma(t) &= \Big\{\sigma^\ell \mid \exists\, \sigma^p \in \Sigma(t-1)\ , \exists j \text{ tel que} \\ &\qquad\qquad \sigma^p = pred_j(\sigma^\ell) \text{ et } f(\sigma^\ell) = z_t\Big\} \\ &= succ\Big(\Sigma(t-1)\Big) \cap f^{-1}(z_t).\end{aligned}$$

   (b) Vérifier s'il existe un bit constant sur l'ensemble des survivants : on cherche $i \in [0, n-1]$ tel que

$$\exists\, b \in \{0,1\} \text{ tel que } \forall \sigma \in \Sigma(t),\ \sigma_i = b.$$

   (c) Si le nombre de bits linéairement indépendants est inférieur à K, incrémenter t, retourner en 2a. Sinon aller en 3.
3. ***Terminaison :***
   inverser le système linéaire $(K \times K)$ afin de retrouver l'état initial du générateur.

Tab. 4.1 – Algorithme du treillis - version forward.

à évaluer un chemin qui serait associé à une métrique minimale, contrairement à l'algorithme de Viterbi.
Illustrons le fonctionnement de l'algorithme sur un exemple.

Exemple 6 *Considérons la fonction à quatre entrées décrite dans la table 4.2. Le treillis correspondant à une fonction à quatre entrées est construit en utilisant la méthode décrite plus haut. Il est représenté sur la figure 4.2. Les vecteurs d'entrée binaires sont numérotés par leur correspondant décimal. Supposons que l'on reçoive la séquence* $(z) = (1,0,0,0,1,0,0,1)$ *et développons l'algorithme sur ces huit bits reçus. La figure 4.3 permet de visualiser l'évolution de l'ensemble des survivants en fonction des bits reçus : les états cerclés représentent les survivants. Les transitions épaisses correspondent aux transitions qui sont rendues possibles par l'ensemble des prédécesseurs, les successeurs et le bit reçu. Aucune des transitions en pointillés n'a lieu.*

Détaillons le parcours du treillis (sans détailler la résolution du système linéaire $(K \times K)$*) :*
- $t = 0$ *:* $\Sigma(t=0) = \{\sigma \mid f(\sigma) = z_0 = 1\} = \{1,3,5,8,9,10,11,13\}$ *;*
- $t = 1$ *: d'après la structure du treillis, les successeurs des états appartenant à* $\Sigma(t=0)$ *sont*

vecteur d'entrée	sortie
0 = 0000	0
1 = 0001	1
2 = 0010	0
3 = 0011	1
4 = 0100	0
5 = 0101	1
6 = 0110	0
7 = 0111	0
8 = 1000	1
9 = 1001	1
10 = 1010	1
11 = 1011	1
12 = 1100	0
13 = 1101	1
14 = 1110	0
15 = 1111	0

Tab. 4.2 – Table des valeurs d'une fonction à $n = 4$ entrées.

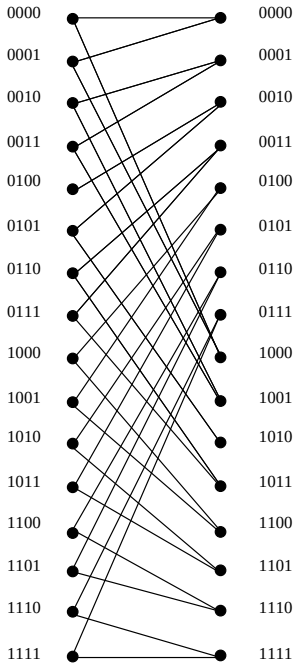


Fig. 4.2 – Treillis d'une fonction à $n = 4$ entrées.

inclus dans l'ensemble $\{0, 8, 1, 9, 2, 10, 4, 12, 5, 13, 6, 14\}$. *Le bit reçu à l'instant* $t = 1$ *est* $z_1 = 0$. *Parmi les états que nous venons d'énumérer, il nous faut supprimer* $8, 1, 9, 10, 5, 13$ *puisque* f *appliquée à ces états vaut* 1. *On obtient donc* $\Sigma(t = 1) = \{0, 2, 4, 12, 6, 14\}$. *Tous ces états se terminent par un* 0. *Le bit* a_1 *émis est donc égal à* 0, $\mathbf{a_1 = 0}$ *;*

– $t = 2$ *: en appliquant le même raisonnement à* $\Sigma(t = 1)$, *on trouve* $\Sigma(t = 2) = \{0, 2, 6, 14, 7, 15\}$.

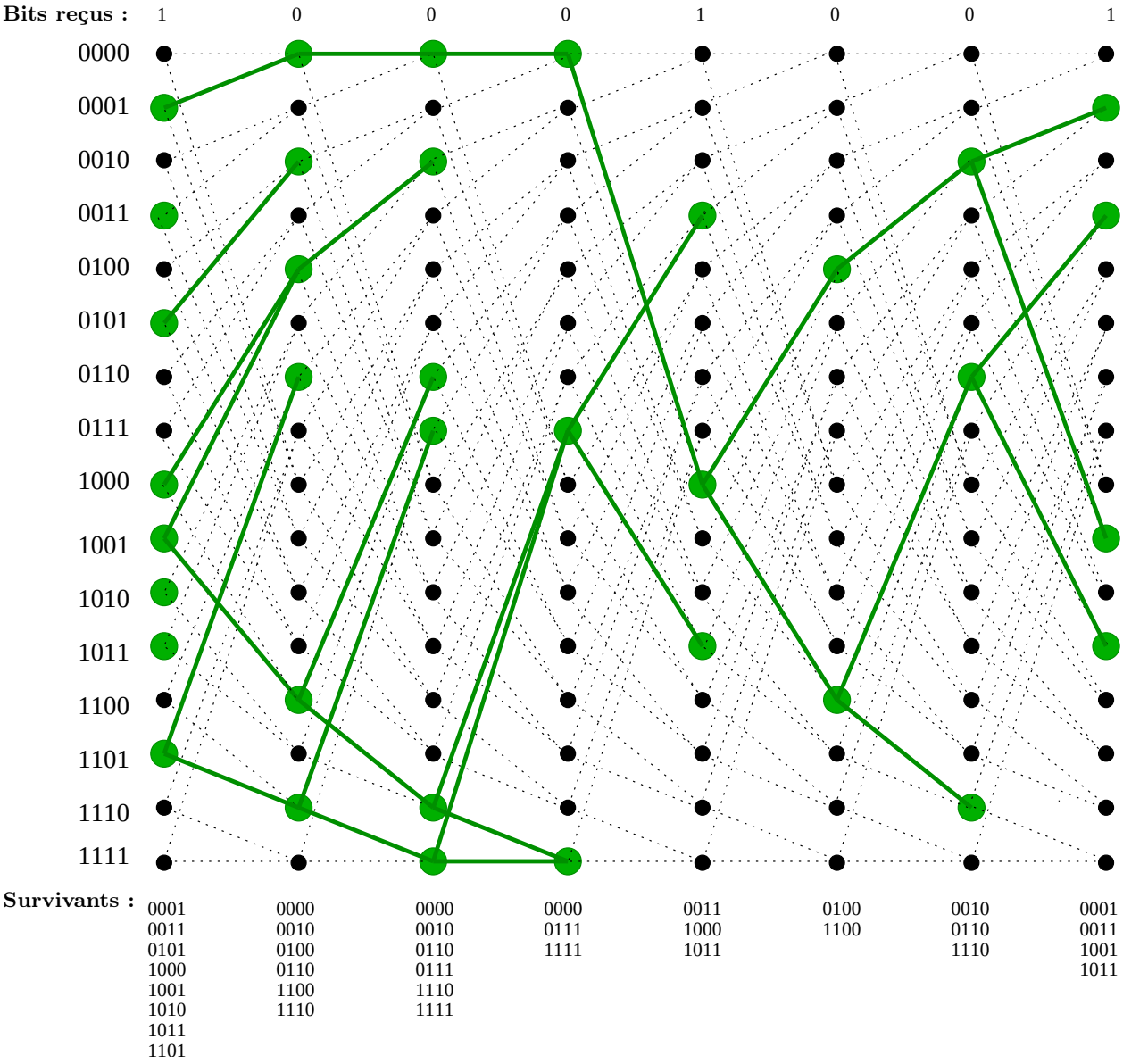


Fig. 4.3 – Déroulement de l'algorithme du treillis sur une fonction à $n = 4$ entrées.

Ces bits n'ont pas de dernier bit commun ;
- $t = 3$ *:* $\Sigma(t = 3) = \{0, 7, 15\}$ *;*
- $t = 4$ *:* $\Sigma(t = 4) = \{8, 3, 11\}$ *;*
- $t = 5$ *:* $\Sigma(t = 5) = \{4, 12\}$ *donc* $\mathbf{a_5} = \mathbf{0}$ *;*
- $t = 6$ *:* $\Sigma(t = 6) = \{2, 6, 14\}$*. Ces états se terminent tous par un 0 donc* $\mathbf{a_6} = \mathbf{0}$ *;*
- $t = 7$ *:* $\Sigma(t = 7) = \{1, 9, 3, 11\}$*. Ces états se terminent tous deux par un 1. Donc* $\mathbf{a_7} = \mathbf{1}$.

Remarques :

1. Les états survivants à l'instant t ne dépendent que de l'instant $t-1$ et de z_t. Ainsi, cet algorithme requiert la seule mémorisation des états à l'instant $t-1$ pour déterminer les survivants à l'instant t. La mémoire requise est donc *a priori* en

$$\mathcal{O}\Big(\max\left(\text{card}[Supp(f)],\ \text{card}[Supp(\overline{f})]\right) \Big)$$

   ce qui, dans le cas d'une fonction équilibrée, vaut 2^{n-1}.

2. Il est inutile d'examiner tous les bits des états survivants : l'examen de leur dernier bit se révèle beaucoup plus efficace puisque tout bit en position $j \neq n$ du vecteur d'entrée à l'instant t se retrouve en position n à l'instant $(t+n-j)$ (sous réserve que $t+n-j < N$). Comme les survivants à l'instant $(t + n - j)$ sont obtenus en décalant $(n - j)$ fois les survivants de l'instant t (et en complétant les $(n - j)$ premières positions par des 0 et des 1), si le j-ième bit est constant à l'instant t, le n-ième bit sera forcément constant à l'instant $(t + n - j)$.

3. Le nombre de survivants à l'instant t est borné par

$$\max\Big(\text{card}[\text{Supp}(f)],\ \text{card}[\text{Supp}(\overline{f})],\ 2\text{card}[\Sigma(t-1)]\Big)$$

   puisque chaque état survivant à l'instant t a au plus deux successeurs.

   Nous remarquons que le nombre de survivants entre l'instant t et l'instant $t+1$ peut décroître pour deux raisons :

   (a) la première raison peut être que deux états survivants à l'instant $t-1$ ont même successeur. Dans le cas traité précédemment, cela se produit par exemple à l'instant $t = 1$, où les deux états 8 et 9 qui ont survécu à l'instant $t = 0$ produisent tous deux l'état 4 à l'instant $t = 1$;

   (b) la seconde raison est qu'un état qui survit à l'instant t peut ne donner aucun successeur à l'instant suivant : par exemple, les états 2, 6 et 7 qui survivent à l'instant $t = 2$ n'ont aucun successeur à l'instant $t = 3$.

4. L'algorithme a été décrit dans le cas où les espacements sont égaux à 1, $\forall i \in [0, n-2],\ \lambda_i = 1$, mais peut être généralisé à d'autres cas de figures :

   (a) si les espacements sont tous égaux à λ, l'algorithme du treillis est appliqué de façon similaire, en opérant sur une séquence décimée de la séquence initiale. La longueur de séquence disponible est toujours identique, égale à N, mais correspond à λ parcours du treillis initialisés en $t = 0, 1, \ldots, \lambda - 1$;

   (b) si les espacements sont quelconques mais que $\Lambda = \frac{\sum \lambda_i}{\text{pgcd}(\lambda_i)}$ n'est pas trop élevé, la fonction à n entrées est artificiellement transformée en une fonction ayant Λ entrées, espacées régulièrement d'une valeur $\text{pgcd}(\lambda_i)$. Il suffit pour cela de « dégénérer » la fonction initiale sur les $\Lambda - n$ positions restantes.

4.1.3 Première amélioration de l'algorithme du treillis : recherche de combinaisons linéaires constantes

La recherche de bits constants sur l'ensemble des survivants peut être étendue à la recherche de relations linéaires constantes sur ce même ensemble : puisque tout bit de la séquence initiale (a) s'exprime linéairement en fonction des K premiers bits $(a_0, \ldots, a_{K-1})$, toute combinaison linéaire de la séquence (a) vérifie la même propriété. Ainsi, si nous parvenons à identifier K formes linéaires indépendantes de $a_0, \ldots, a_{K-1}$, nous pourrons à l'aide de l'inversion linéaire d'un système $(K \times K)$ retrouver $a_0, \ldots, a_{K-1}$. Nous allons donc chercher non plus des bits communs à tous les états survivants, mais des combinaisons linéaires de bits communes à tous les survivants.

L'étape (2) de l'algorithme du treillis (voir tableau 4.1) est alors remplacée par une étape de recherche de formes linéaires constantes.

Recherche de formes linéaires constantes

Afin de mener cette recherche le plus efficacement possible, nous suggérons deux méthodes de recherche de formes linéaires constantes, l'intérêt de chacune d'entre elles dépendant directement du nombre de survivants et du nombre d'entrées de la fonction.

• La première méthode la plus simple est de résoudre un système linéaire : nous écrivons les survivants à l'instant t sous forme binaire dans une matrice :

$$\mathbf{S_{lin}} = \begin{pmatrix} \sigma_1^1 & \ldots & \sigma_n^1 \\ \vdots & \vdots & \vdots \\ \sigma_1^{\nu(t)} & \ldots & \sigma_n^{\nu(t)} \end{pmatrix}. \tag{4.3}$$

La recherche de formes linéaires constantes sur l'ensemble des survivants revient à chercher les solutions $\mathbf{X}$ du système :

$$\mathbf{S_{lin}X + B = 0}_{\nu(\mathbf{t})} \ ,$$

où $\mathbf{B}$ est une matrice colonne tout à zéro ou tout à un.
La complexité de cette recherche dépend du nombre de survivants $\nu(t) = \mathrm{card}\Big(\Sigma(t)\Big)$. Elle est majorée par

$$\mathcal{O}\Big(\nu(t)^3\Big).$$

• Selon le nombre $\nu(t)$ de survivants, une seconde méthode pourra se révéler moins complexe. À chaque instant t, considérons la fonction Φ_t, fonction indicatrice des états survivants, qui vaut 1 sur l'ensemble des survivants, 0 partout ailleurs.

Définition : La **fonction indicatrice** des états survivants à l'instant t, Φ_t, est définie de la façon suivante :

$$\begin{aligned} \Phi_t \ : \{0,1\}^n &\longrightarrow \{0,1\} \\ \sigma &\longmapsto 1 \ \text{ si } \sigma \in \Sigma(t) \qquad (4.4) \\ &\qquad\ 0 \ \text{ sinon .} \qquad (4.5) \end{aligned}$$

Rappelons que $\widehat{\Phi_t}$ désigne la transformée de Fourier de la fonction Φ_t. Alors, la fonction indicatrice des survivants satisfait le lemme suivant :

Lemme 2 *Si* $\exists\ \mathbf{u}\ \in \{0,1\}^n$ *tel que* $|\widehat{\Phi_t}(\mathbf{u})| = \widehat{\Phi_t}(\mathbf{0_n})$
alors, la fonction

$$\begin{array}{rccl} \ell_{\mathbf{u}} & : \{0,1\}^n & \longrightarrow & \{0,1\} \\ & \mathbf{x} & \longmapsto & \mathbf{u}.\mathbf{x} \mod 2 \end{array}$$

est constante sur $\Sigma(t)$.

Preuve :
Par définition,

$$\forall \mathbf{u} \in \{0,1\}^n,\ \widehat{\Phi_t}(\mathbf{u}) = \sum_{\mathbf{x}\in\{0,1\}^n} \Phi_t(\mathbf{x})(-1)^{\mathbf{u}.\mathbf{x}}.$$

Posons

$$\begin{array}{rcl} A_0 & = & \{\mathbf{x} \in \{0,1\}^n \ / \ \ell_u(\mathbf{x}) = 0\} \\ A_1 & = & \{\mathbf{x} \in \{0,1\}^n \ / \ \ell_u(\mathbf{x}) = 1\}. \end{array}$$

On a donc

$$\begin{array}{rcl} \left|\widehat{\Phi_t}(\mathbf{u})\right| & = & \left|\displaystyle\sum_{A_0} \Phi_t(\mathbf{x}) - \sum_{A_1} \Phi_t(\mathbf{x})\right| \\ & = & \left|\mathrm{card}\Big(A_0 \cap \mathrm{Supp}(\Phi_t)\Big) - \mathrm{card}\Big(A_1 \cap \mathrm{Supp}(\Phi_t)\Big)\right| \\ & \leq & \mathrm{card}\Big(\mathrm{Supp}(\Phi_t)\Big) \ = \ \Phi_t(\mathbf{0_n}), \end{array}$$

et l'égalité est vérifiée si et seulement si A_0 ou A_1 est l'ensemble vide. ∎

Ainsi, la recherche de formes linéaires constantes est-elle de complexité identique au calcul d'une transformée de Fourier, soit en $\mathcal{O}(n \times 2^n)$ au lieu de $\mathcal{O}(2^{2n})$ si l'on regardait exhaustivement toutes les formes linéaires.
De plus, pour la même raison que celle invoquée plus haut, on ne s'intéressera qu'aux formes linéaires qui font intervenir le dernier bit σ_n des survivant, ce qui réduit de moitié le nombre de formes linéaires à examiner.

La seconde méthode sera donc plus efficace que la première tant que

$$n \times 2^{n-1} \leq \nu(t)^3.$$

Algorithme du treillis utilisant la recherche de formes linéaires constantes

Utilisant l'un des deux algorithmes de recherche de formes linéaires constantes, nous pouvons énoncer la version généralisée de l'algorithme, décrite dans le tableau 4.3. Notons que la recherche de bits constants décrite dans le paragraphe 4.1.2 apparaît ainsi comme un cas particulier de la recherche de formes linéaires constantes, en se restreignant aux formes linéaires triviales de poids 1.

Algorithme du treillis généralisé à la recherche de formes linéaires constantes

1. ***Initialisation :***
   à t=0, éliminer tous les états σ de $\Sigma(t=0)$ tels que $f(\sigma) \neq z_0$.
2. ***Déroulement :***
   $0 < t < N$
   (a) Déterminer les successeurs possibles des éléments de $\Sigma(t-1)$. Parmi ces successeurs, ne conserver que les états qui, filtrés par la fonction, produisent le bit observé à l'instant t :

$$\Sigma(t) = \Big\{\sigma \mid \exists\ \sigma^p \ \in\ \Sigma(t-1)\ , \exists j \text{ tel que } \sigma^p = pred_j(\sigma) \ \text{ et } \ f(\sigma) \ = \ z_t\Big\}.$$

   (b) Vérifier s'il existe une forme linéaire constante sur l'ensemble des survivants en utilisant l'une des deux méthodes proposées ci-dessous :
   – soit on résout un système matriciel ;
   – soit on cherche ℓ_i tel que :

$$\exists\ b\ \in \{0,1\} \text{ tel que } \forall \sigma \in \Sigma(t),\ \ell_i(\sigma) = b.$$

   Pour cela on calcule $\widehat{\Phi_t}$ (Φ_t définie dans la formule (4.1.3)) et on applique le lemme 2.
   (c) Si le nombre de formes linéaires indépendantes est inférieur à K, incrémenter t, retourner en 2a. Sinon aller en 3.
3. ***Terminaison :***
   inverser le système linéaire $(K \times K)$ afin de retrouver l'état initial du générateur.

Tab. 4.3 – Algorithme du treillis généralisé à la recherche de formes linéaires constantes sur l'ensemble des survivants.

Exemple 7 *Revenons à l'exemple traité précédemment :*
- *à l'instant $t = 3$ par exemple, le dernier bit n'est pas constant sur les survivants. En revanche, il est aisé de vérifier qu'il existe deux formes linéaires indépendantes constantes sur ces états : $\sigma_0 + \sigma_1 = 0$ et $\sigma_0 + \sigma_2 = 0$;*
- *de même, à l'instant $t = 4$, nous constatons que les états survivants vérifient : $\sigma_0 + \sigma_1 = 0$.*

Étude de la complexité :

- *mémoire :* les algorithmes précédemment décrits nécessitent la mémorisation des états survivants à l'instant $t-1$ pour déterminer ceux de l'état t. Ainsi, la mémoire requise est-elle en

$$\mathcal{O}(2^n)\ . \tag{4.6}$$

  Toutefois, lorsque les espacements sont réguliers et la fonction équilibrée, le nombre maximal de survivants à chaque instant est de 2^{n-1}. De plus, comme nous le verrons par la suite, le succès de l'algorithme du treillis est assuré par le fait que le nombre de survivants est souvent très inférieur à cette borne maximale : plus précisément, le nombre de survivants à chaque instant est de l'ordre de n. La complexité en mémoire est donc en moyenne inférieure à l'estimation (4.6) ;
- *calculs :* à chaque instant, nous devons évaluer les formes linéaires constantes. Comme nous l'avons développé précédemment, la complexité de ce calcul est en

$$N \times \mathcal{O}\Big(\min(\nu(t)^2, n2^n)\Big)\ . \tag{4.7}$$

  Toutefois, lorsque le nombre d'entrées de la fonction est supérieur ou égal à cinq et que le nombre de survivants à chaque instant est de l'ordre de n, la complexité de ces calculs sera de l'ordre de :

$$N \times \mathcal{O}(n^3). \tag{4.8}$$

Comme nous l'avons mentionné précédemment, l'ensemble $\Sigma(t)$ des survivants à l'instant t, tient compte à la fois du passé et du bit reçu à l'instant t. En revanche, le futur n'est pas utilisé lors de la sélection des survivants. Nous proposons ci-après deux améliorations de la version basique de l'algorithme : une première version, qui consiste à décrire l'algorithme en parcourant le treillis dans les deux sens, une seconde version, qui vise à restreindre le nombre de survivants à chaque instant en utilisant conjointement passé-présent-futur.

4.1.4 Une seconde amélioration de l'algorithme du treillis : l'algorithme Forward-Backward non intersectant (F.B.N.I.)

Une vue plus complète de l'algorithme suscité consiste à l'appliquer « dans les deux sens », c'est-à-dire de décrire le treillis de gauche à droite, puis de revenir en arrière. La version forward utilise alors le passé et le présent pour déterminer les survivants, la version backward utilise, quant à elle, le futur et le présent.
À la fin du parcours forward, le nombre de survivants est, nous l'espérons, restreint. Nous pouvons donc améliorer l'algorithme backward, en lui attribuant comme état initial, l'état final du parcours forward. L'algorithme est ensuite appliqué dans les deux sens, comme il a été décrit au paragraphe 4.1.3.

L'algorithme forward-backward est décrit dans le tableau 4.4. Nous l'avons qualifié de « non-intersectant » par opposition à l'algorithme forward-backward décrit dans le paragraphe 4.1.5 qui suit.

Algorithme Forward-Backward Non Intersectant (F.B.N.I.)

1. Appliquer l'algorithme forward tel qu'il a été décrit dans le tableau 4.1. En fin de parcours du treillis, conserver les états survivants.
2. En initialisant le parcours backward par l'ensemble final des survivants obtenus à l'étape 1, appliquer le même algorithme en sens inverse, c'est-à-dire en considérant le passé comme futur.
3. Une fois que K formes linéaires de la séquence d'entrée linéairement indépendantes ont pu être identifiées, inverser le système linéaire $(K \times K)$ afin de récupérer la clé secrète.

Tab. 4.4 – Algorithme Forward-Backward (non intersectant).

Exemple 8 *Nous reprenons toujours le même exemple. Rappelons qu'après parcours du treillis dans le sens forward, nous avons obtenu :* $\Sigma_{fwd}(t=7) = \{1, 9, 3, 11\}$. *L'algorithme appliqué dans le sens backward donne alors les résultats suivants :*

$$\begin{aligned} \Sigma_{bwd}(t=7) &= \{1,9,3,11\} & \Sigma_{bwd}(t=6) &= \{2,6,7\} \\ \Sigma_{bwd}(t=5) &= \{4,12,14,15\} & \Sigma_{bwd}(t=4) &= \{8,9,13\} \\ \Sigma_{bwd}(t=3) &= \{0,2\} & \Sigma_{bwd}(t=2) &= \{0,4\} \\ \Sigma_{bwd}(t=1) &= \{0\} & \Sigma_{bwd}(t=0) &= \{1\}. \end{aligned}$$

Comparativement au seul parcours forward du treillis, ce double parcours a permis de réduire le nombre de survivants, notamment aux instants $t \leq 3$, *où de nombreuses formes linéaires constantes sont disponibles. On parvient même, sur cet exemple précis, à identifier de façon unique le vecteur d'entrée de la fonction aux instants* $t = 1$ *et* $t = 0$.

La complexité de cet algorithme forward-backward non intersectant est identique à celle qui a été évaluée précédemment (nous examinons $2N$ instants au lieu de N). Nous avons observé qu'un second parcours dans le treillis en initialisant le second parcours forward par l'ensemble final des états du premier parcours backward n'engendrait aucune amélioration significative : l'ensemble des survivants $\Sigma(t)$ obtenu lors du second parcours devient identique à celui obtenu lors du premier parcours pour de très faibles valeurs de t. Un second parcours n'est donc d'aucune utilité.

La description backward nous fournit, à chaque instant, l'ensemble des états compatibles avec le futur et le présent. Le vecteur qui était réellement en entrée de la fonction est forcément compatible avec le passé et le présent d'une part, avec le futur et le présent d'autre part. Donc le vecteur d'entrée appartient forcément à $\Sigma_{fwd}(t)$ et à $\Sigma_{bwd}(t)$.

4.1.5 Forward-Backward intersectant

D'après la remarque qui a été faite ci-dessus, l'intersection des ensembles forward et backward de survivants contient le « vrai » vecteur d'entrée. Nous proposons alors de considérer l'intersection : :

$$\Sigma_{fwd}(t) \bigcap \Sigma_{bwd}(t) \ .$$

Quel en est l'intérêt ?
D'après les propriétés de la fonction et comme nous l'avons observé, les deux ensembles $\Sigma_{fwd}(t)$ et $\Sigma_{bwd}(t)$ sont généralement différents : ils ont forcément en commun le « bon » état, mais ils peuvent différer sur les autres vecteurs. Prendre l'intersection nous permettra de réduire l'ensemble des états possibles et ainsi espérer obtenir davantage d'information (en terme de formes linéaires notamment) sur le vecteur d'entrée à l'instant t. Cet algorithme est à rapprocher de l'algorithme forward-backward B.C.J.R., initialement introduit dans l'article [3], qui tient compte **du passé**, **du futur** et de **la transition courante dans le treillis** pour évaluer une probabilité sur le mot de code. Une description de cet algorithme est présentée en annexe (voir annexe F). L'algorithme du forward-backward intersectant (F.B.I.) est présenté dans le tableau 4.5.

Exemple 9 *Dans le cas que nous avons traité jusqu'ici, nous constatons par exemple que* $\Sigma_{fwd}(2) = \{0, 2, 6, 14, 7, 15\}$ *alors que* $\Sigma_{bwd}(2) = \{0, 4\}$. *Or l'intersection de ces deux ensembles se réduit à l'état* 0. *Le « vrai » état à l'instant* 2 *est donc l'état 0 : intersecter les deux ensembles de survivants nous a permis d'identifier complètement le vecteur d'entrée de la fonction à l'instant* $t = 2$.

Étude de la complexité :

- *mémoire :* la mémoire requise est ici N fois plus grande que celle utilisée dans les algorithmes précédents, puisqu'il nous faut mémoriser les états survivants dans le sens forward. D'où une complexité en mémoire de l'ordre de

$$\mathcal{O}(N \times 2^n) \ . \tag{4.10}$$

  Le nombre de survivants se réduit en fait très souvent à n (et non 2^n) ;
- *calculs :* la complexité en terme de calculs est identique à celle qui était évaluée précédemment, c'est-à-dire en :

$$\mathcal{O}\Big(N \times \min(n^3, n2^n)\Big). \tag{4.11}$$

Remarque

À première vue, on peut penser qu'itérer le procédé permettrait de réduire le cardinal des survivants à chaque instant. Supposons en effet que nous ayons déjà balayé le treillis dans les sens forward et backward, puis que nous ayons pris l'intersection des survivants à chaque instant. Il semblerait naturel de recommencer à décrire le treillis de gauche à droite, en espérant, à partir de moins d'états à l'instant t faire diminuer le nombre de survivants à l'instant $t + 1$.
En fait, il n'en est rien : une seule application de l'algorithme F.B.I. permet de réduire l'ensemble des survivants au minimum que l'on peut espérer obtenir avec cet algorithme.

Algorithme Forward-Backward Intersectant (F.B.I.)

1. $\forall t \;\; 0 < t < N-1$ déterminer l'ensemble des survivants en décrivant le treillis dans le sens forward, $\Sigma_{fwd}(t)$.
2. $t = N-1 : \Sigma_{\cap}(N-1) = \Sigma_{fwd}(N-1)$.
3. $t > 0$: déterminer l'ensemble des survivants $\Sigma_{\cap}(t)$ en fonction de $\Sigma_{\cap}(t+1)$, de z_t et du treillis parcouru dans le sens forward :

$$\Sigma_{\cap}(t) = \left\{\sigma \text{ tels que } \exists j \text{ tel que } succ_j(\sigma) \in \Sigma_{bwd}(t+1) \text{ et } f(\sigma) = z_t\right\} \bigcap \Sigma_{fwd}(t). \tag{4.9}$$

   Vérifier s'il existe une forme linéaire constante sur l'ensemble des survivants en utilisant l'une des deux méthodes proposées :
   – soit on résout un système matriciel ;
   – soit on cherche ℓ_i tel que :

$$\exists\, b \;\in \{0,1\} \text{ tel que } \forall \sigma \in \Sigma(t),\; \ell_i(\sigma) = b.$$

   Pour cela on calcule $\widehat{\Phi_t}$ (Φ_t définie dans la formule (4.1.3)) et on applique le lemme 2.
4. Si le nombre de formes linéaires indépendantes est inférieur à K, augmenter la longueur de séquence, N et retourner à l'étape 1. Sinon aller en 5.
5. Inverser le système linéaire ($K \times K$) afin de retrouver l'état initial du générateur.

Tab. 4.5 – Algorithme Forward-Backward Intersectant.

En effet, plaçons-nous à l'instant $t = 0$, après qu'un aller-retour a été effectué. Notons $\Sigma_{\cap}^{nbfb}(t)$ l'ensemble des survivants obtenus en appliquant l'algorithme F.B.I. après $nbfb$ aller-retours à l'instant t. Le second parcours dans le treillis est initialisé avec l'ensemble des survivants obtenus à $t = 0$ après une première application. Montrons que la description du treillis dans le sens aller engendre le même ensemble de survivants à l'instant $t = 1$ que $\Sigma_{\cap}^{nbfb=1}(t=1)$, *i.e.* montrons que :

$$\Sigma_{fwd}^{nbfb=2}(t=1) = \Sigma_{\cap}^{nbfb=1}(t=1)\ .$$

– tout d'abord, $\Sigma_{fwd}^{nbfb=2}(t=1) \leq \Sigma_{\cap}^{nbfb=1}(t=1)$, car $\Sigma_{fwd}^{nbfb=2}$ est obtenu en décrivant le treillis sur un sous-ensemble de l'ensemble des états ayant conduit à $\Sigma_{\cap}^{nbfb=1}(t=1)$;
– ensuite, examinons les différents *scenarii* qui permettraient de diminuer le nombre de survivants de $t = 0$ à $t = 1$:
  – une première solution serait qu'un état n'ait qu'un seul successeur au lieu des deux qui appartiendraient à $\Sigma_{\cap}^{nbfb=1}(t=1)$. Cela est impossible : si les deux successeurs survivaient après intersection lors du premier aller-retour, c'est que, filtrés par la fonction,

ils correspondent tous deux au bit observé à l'instant $t = 1$. Ils vont donc survivre à nouveau ;
- une seconde solution serait qu'un survivant de l'instant $t = 0$ n'ait pas de successeur à l'instant $t = 1$: cela est impossible, car si cet état a survécu à l'instant $t = 0$, c'est qu'il avait un prédécesseur (dans la description backward) à l'instant $t = 1$, qui avait survécu par intersection lors du premier aller-retour. Il aura donc ce même état comme successeur dans la description forward ;

– ainsi,

$$\Sigma_{fwd}^{nbfb=2}(t=1) = \Sigma_{\cap}^{nbfb=1}(t=1).$$

Par récurrence, nous montrons que cette égalité est vérifiée quel que soit t et en appliquant le même raisonnement pour la description backward du treillis, nous pouvons montrer que :

$$\Sigma_{bwd}^{nbfb=2}(t) = \Sigma_{\cap}^{nbfb=1}(t).$$

D'où le résultat final :

$$\Sigma_{\cap}^{nbfb=2}(t) = \Sigma_{\cap}^{nbfb=1}(t).$$

Ainsi, un seul aller-retour suffit à déterminer l'ensemble « minimal » des survivants au sens où plusieurs aller-retours ne permettraient pas de réduire davantage son cardinal.

4.1.6 Quelques résultats de simulation

Nous présentons ici quelques résultats de simulation[2] ; ces résultats sont obtenus avec le polynôme de rétroaction de degré $K = 100$:

$$g(X) = 1 + X^2 + X^7 + X^8 + X^{100}.$$

Il est à noter que le poids du polynôme n'est pas un paramètre déterminant de l'algorithme : il est bien sûr relié à la séquence produite, mais n'a pas d'autre incidence sur le fonctionnement de l'algorithme. Le seul paramètre qui intervient est **le degré** K, puisqu'il nous faut obtenir K formes linéaires indépendantes pour identifier les K premiers bits à l'aide d'une simple inversion matricielle.

Nous avons testé les algorithmes sur trois fonctions :
- la fonction f_1 à cinq entrées ;
- la fonction f_{13} à sept entrées ;
- la fonction f_{15} à huit entrées.

Ces fonctions sont décrites dans l'annexe A. Pour chacune d'entre elles, nous présentons dans le tableau de résultats 4.6 la longueur minimale de séquence observée nécessaire au succès des algorithmes forward avec recherche de combinaisons linéaires et forward-backward intersectant (F.B.I.). Les résultats obtenus ont été moyennés sur cinq mille tirages de séquences. Chaque résultat est associé au nombre moyen de survivants dénombrés, qui figure entre parenthèses.

[2] Il est à noter que ces résultats diffèrent de ceux publiés dans l'article [56]. Les résultats présentés dans cet article n'avaient pas été moyennés sur plusieurs tirages et la séquence testée était une séquence à impulsion. Les résultats alors obtenus étaient nettement moins bons que ceux observés ici. L'explication qui nous semble justifier un tel écart est la suivante : le polynôme de rétroaction étant de poids faible et la séquence d'initialisation étant la séquence à impulsion, la séquence produite par le LFSR variait peu. La sortie de la fonction comportait donc peu de transitions parmi les premiers bits générés, le parcours dans le treillis était assez uniforme et l'ensemble des survivants quasi-constant au début de la séquence produite. D'où un ralentissement de la convergence de l'algorithme.

Fonction utilisée	**Longueur minimale moyenne nécessaire au succès de l'algorithme forward**	**Longueur minimale moyenne nécessaire au succès de l'algorithme F.B.I.**
f_1 à 5 entrées, 1-résiliente	217, 9 (5,8)	174,8 (1,9)
f_{13} à 7 entrées, 3-résiliente	290,5 (9)	168,6 (3,4)
f_{15} à 8 entrées, 2-résiliente	395,6 (15,1)	155,1 (3,29)

Tab. 4.6 – Résultats de simulation obtenus sur un registre de mémoire $K = 100$ filtré par différentes fonctions : longueur minimale nécessaire pour identifier K formes linéaires indépendantes. Entre parenthèses figure le nombre moyen de survivants obtenus.

Quelques commentaires :

1. Comme prévu, quelle que soit la fonction testée, le forward-backward intersectant donne de bien meilleurs résultats qu'un simple algorithme forward. Le nombre de bits nécessaires au succès de l'algorithme est de l'ordre de K (ici $K = 100$), ce qui est très peu.
2. Globalement, le nombre moyen de survivants augmente avec le nombre d'entrées de la fonction, ce qui n'est pas surprenant non plus. Par ailleurs, dans le cas de l'algorithme F.B.I., nous observons que le nombre de survivants est très éloigné de 2^{n-1}, il est même largement inférieur à n. Le nombre de formes linéaires constantes est donc très élevé.
3. En revanche, nous pouvons constater que pour le forward-backward intersectant, la longueur de séquence requise pour le succès de l'algorithme diminue avec le nombre d'entrées (sans varier beaucoup pour autant !) : ceci est probablement lié aux propriétés de la fonction relativement à sa première et/ou sa dernière variable, étudiées par J. Golic (cf. [40]), et au fait que, si nous modélisons la fonction comme un processus aléatoire, le nombre d'états communs aux ensembles de survivants forward et backward sera statistiquement d'autant plus faible que le nombre d'états possibles est élevé.
4. Cet algorithme souligne l'écart fondamental de comportement entre la fonction booléenne filtrant un registre à décalage et un canal binaire symétrique. En effet, comme nous l'avons vu au chapitre 1, si nous considérons non plus une fonction booléenne mais un canal binaire symétrique, le nombre de bits d'observation en deçà duquel nous ne pouvons espérer retrouver la clé sans erreur doit vérifier :

$$N \geq \frac{K}{C(p)} \ ,$$

où $C(p) = 1 - H_2(p)$ est la capacité d'un canal binaire symétrique sans mémoire, fonction de sa probabilité de transition.

Appliquons cette relation aux fonctions que nous avons testées ici : prenons comme valeur de la probabilité de transition la valeur correspondant au maximum du spectre en valeur absolue de chacune des fonctions. Rappelons qu'ici $K = 100$.

(a) *Fonction f_1 :* il s'agit d'une fonction à $n = 5$ entrées, de probabilité de transition équivalente $p = 0.25$. En utilisant la courbe de capacité donnée au chapitre 1 (voir figure 1.4 page 19), nous voyons directement que la capacité associée à un tel canal est $C(p = 0.25) \approx 0.19$.

Alors

$$N \geq \frac{K}{C(p)} \Longrightarrow N \geq 527 \ .$$

(b) *Fonction f_{13} :* il s'agit d'une fonction à $n = 7$ entrées, de probabilité de transition équivalente $p = 0.375$, avec $C(p = 0.375) \approx 0.04$.
N doit alors satisfaire

$$N \geq 2500 \ .$$

(c) *Fonction f_{15} :* il s'agit d'une fonction à $n = 8$ entrées, de probabilité de transition équivalente $p = 0.4375$ ce qui correspond à une capacité $C(p = 0.4375) \approx 0.01$.
N doit alors satisfaire :

$$N \geq 10000 \ .$$

Ainsi, la théorie de l'information nous permet d'évaluer ces bornes sur la longueur d'observation requise pour pouvoir décoder l'initialisation du registre, dans le cas où le canal n'est pas booléen mais binaire symétrique. Nous constatons que les valeurs figurées dans le tableau 4.6 sont nettement inférieures à celles prévues par le théorème de codage : la prise en compte de la structure très particulière du registre filtré par une fonction, notamment de l'effet mémoire intrinsèque au système renforcé par l'espacement régulier des entrées de la fonction, permet d'obtenir de bien meilleurs résultats. La modélisation de la fonction en canal binaire symétrique est donc ici injustifiée.

4.1.7 Espérance du nombre de survivants

Dans cette partie, nous nous proposons d'évaluer le nombre moyen de survivants lors d'une description unique du treillis, par exemple une description forward.
Nous faisons dans cette étude l'hypothèse que la sortie de la fonction est une variable aléatoire prenant les valeurs 0 ou 1 avec la probabilité $p = \frac{1}{2}$ afin de modéliser une fonction aléatoire « équilibrée ».

Sans perdre en généralité et pour des raisons évidentes de lisibilité, nous supposons que le bon chemin est le chemin « tout à zéro ».
Nous supposons que la fonction a n entrées et les états $\sigma_i \quad i \in [0; 2^n - 1]$ sont indexés par leur écriture décimale. Développons alors la probabilité de survie de chaque état en cours d'algorithme : la probabilité qu'un état survive à l'instant t est liée à **la probabilité de survie de ses prédécesseurs** à l'instant $t-1$ et à **la valeur prise en sortie de la fonction** à l'instant t. Rappelons que $pred_0(\sigma_i)$ et $pred_1(\sigma_i)$ sont les prédécesseurs de σ_i finissant respectivement par un 0 et un 1. Afin de simplifier l'écriture, nous les noterons $pred_0(i)$ et $pred_1(i)$.

Alors, la probabilité que le i-ième état survive à l'instant t sera notée $P[i,t]$. Cette quantité vérifie :
$\forall t > 0$

$$\forall i, \ \forall t, \quad P[i,t] \ = \ P\Big[\Big\{ f\left(\sigma_i\right) = z_t \Big\} \bigcap \Big\{ \{pred_0(i) \in \Sigma(t-1)\} \bigcup \{pred_1(i) \in \Sigma(t-1)\} \Big\} \Big] \ . \quad (4.12)$$

En d'autres termes, l'état σ_i survit à l'instant t si et seulement si cet état filtré par f correspond au bit reçu à l'instant t d'une part et d'autre part, l'un de ces deux prédécesseurs au moins a survécu à l'instant $t-1$.

En faisant l'hypothèse que les évènements

$$\Big\{ f\left(\sigma_i\right) = z_t \Big\}$$

états	successeurs	
0	0	4
1	0	4
2	1	5
3	1	5
4	2	6
5	2	6
6	3	7
7	3	7

Tab. 4.7 – Transitions d'états pour une fonction à $n = 3$ entrées.

et

$$\left\{ \ \{pred_0(i) \in \Sigma(t-1)\} \bigcup \{pred_1(i) \in \Sigma(t-1)\} \ \right\}$$

sont indépendants (ce qui n'est pas le cas en réalité, la fonction étant parfaitement définie et fixée), nous pouvons alors écrire :

$$\begin{aligned} P[i,t] \ = \ P\Big[\ \{f(\sigma_i) = z_t\} \ \Big] \ \times \ P\Big[\ \Big\{ \ \{pred_0(i) \in \Sigma(t-1)\} \\ \bigcup \ \{pred_1(i) \in \Sigma(t-1)\} \ \Big\} \ \Big] \end{aligned} \tag{4.13}$$

et donc,

$$\begin{aligned} P[i,t] = \quad \frac{1}{2} \ \times \ \Big(\ P[pred_0(i), t-1] \ + \ P[pred_1(i), t-1] \\ - \ P[pred_0(i), t-1] \times P[pred_1(i), t-1] \ \Big) \ . \end{aligned} \tag{4.14}$$

Nous obtenons alors une relation de récurrence sur la probabilité de survie d'un état. L'espérance du nombre de survivants à l'instant t sera alors donnée par :

$$E\Big[\mathrm{card}\left(\Sigma(t)\right)\Big] = \sum_{i=0}^{2^n-1} P\left[i,t\right]. \tag{4.15}$$

Dérivons le calcul des probabilités $P\left[i,t\right]$ sur un exemple : considérons le cas simple d'une fonction à $n = 3$ entrées. Les transitions du treillis sont synthétisées dans la table 4.7 qui fait correspondre chaque état à ses deux successeurs dans le treillis.

– $t = 0$:

$$\begin{aligned} P[0, t=0] \ &= \ 1 \\ \forall i \neq 0 \ \ P[i, t=0] \ &= \ P_A[t=0] = \frac{2^{n-1}-1}{2^n-1} = \frac{3}{7} \ ; \end{aligned}$$

– $t = 1$: l'état 0 ayant pour successeurs 0 et 4, en appliquant l'équation (4.14) nous obtenons :

$$\begin{aligned} P[0, t=1] &= 1 \\ P[4, t=1] &= \frac{1}{2} \\ \forall i \neq 0,4 \;\; P[i, t=1] &= P_A(t=1) = \frac{1}{2}\Big(2P_A(t=0) - P_A^2(t=0) \Big) ; \end{aligned}$$

– $t = 2$: l'état 4 ayant pour successeurs les états 2 et 6,

$$\begin{aligned} P[0, t=2] &= 1 \\ P[4, t=2] &= \frac{1}{2} \\ P[2, t=2] &= P[6, t=2] = P_B(t=2) = \frac{1}{2}\Big(\frac{1}{2} + P_A(t=1) - \frac{P_A(t=1)}{2} \Big) \\ \forall i \neq 0,2,4,6 \;\; P[i, t=2] &= P_A(t=2) = \frac{1}{2}\Big(2P_A(t=1) - P_A^2(t=1) \Big) ; \end{aligned}$$

– $t = 3$: les états 2 et 6 ayant pour successeurs respectifs 1 et 5, 3 et 7,

$$\begin{aligned} P[0, t=3] &= 1 \\ P[4, t=3] &= \frac{1}{2} \\ \forall i = 2,6 \; : \; P_B(t=3) &= \frac{1}{2}\Big(\frac{1}{2} + P_A(t=2) - \frac{P_A(t=2)}{2} \Big) \\ \forall i = 1,3,5,7 \; : \; P_A(t=3) &= \frac{1}{2}\Big(P_A(t=2) + P_B(t=2) - P_A(t=2)P_B(t=2) \Big) ; \end{aligned}$$

– $t > 3$:

$$\begin{aligned} P[0, t] &= 1 \\ P[4, t] &= \frac{1}{2} \\ \forall i = 2,6 \; : \; P[i, t] &= P_B(t) = \frac{1}{2}(\frac{1}{2} + P_A(t-1) - \frac{1}{2}P_A(t-1)) \\ \forall i = 1,3,5,7 \; : \; P[i, t] &= P_A(t) = \frac{1}{2}\Big(P_A(t-1) + P_B(t-1) - P_A(t-1)P_B(t-1) \Big). \end{aligned}$$

On a alors

$$E\Big[\mathrm{card}\Sigma(t)\Big] = 1 + \frac{1}{2} + 2P_B(t) + 4P_A(t), \tag{4.16}$$

puisque l'évolution de deux états est donnée par P_B et que P_A donne la probabilité de survie de quatre d'entre eux.

Par récurrence, nous pouvons montrer que P_A et P_B, vues comme des fonctions de t, sont décroissantes en vérifiant que :

$$\forall t > 3, H_t \;=\; \Big(P_A(t) \leq P_A(t-1) \text{ et } P_B(t) \leq P_B(t-1)\Big) \implies H_{t+1}.$$

et en vérifiant H_3. Or P_A et P_B sont dans l'intervalle $[0,1]$; ces deux suites étant décroissantes et bornées, elles convergent.
La résolution de ces récurrences imbriquées donne la relation de récurrence qui régit l'évolution de P_A :

$$P_A(t) = \frac{1}{8} \times \left[\, 3P_A(t-1) + P_A(t-2) - P_A(t-1)P_A(t-2) + 1 \, \right] . \tag{4.17}$$

Nous vérifions à l'aide d'une simple résolution d'un système de degré 2 et à l'aide des hypothèses $P_A, P_B \in [0,1]$ que la limite de P_A est :

$$\ell_A \;\; = \;\; -2 + \sqrt{5}$$

et nous en déduisons

$$\ell_B \;\; = \;\; -\frac{1}{2} + \frac{\sqrt{5}}{4} \, .$$

D'où un nombre moyen de survivants :

$$E[\Sigma(t)] \longrightarrow 1 + \frac{1}{2} + 2(-\frac{1}{2} + \frac{\sqrt{5}}{4}) + 4(-2 + \sqrt{5}),$$

soit

$$E[\Sigma(t)] \approx 3.06 \, .$$

En moyennant le nombre de survivants sur l'ensemble des fonctions booléennes à trois entrées équilibrées, nous trouvons $\Sigma_t^{moy} = 2.598$, ce qui est assez proche des résultats prévus par l'expérience.

nombre d'entrées n	nombre moyen de survivants prévu par la théorie	nombre moyen de survivants obtenu par l'expérience
4 entrées	4.28	3.7
5 entrées	5.98	5.38
6 entrées	8.34	7.66
7 entrées	11.66	10.9

Tab. 4.8 – Espérance du nombre de survivants : théorie vs expérience.

La table 4.8 figure les résultats « théoriques » et expérimentaux du nombre moyen de survivants obtenu en appliquant l'algorithme forward.

Les résultats théoriques et expérimentaux sont relativement proches ; il est intéressant de constater que l'écart entre théorie et expérience est quasi-constant, de l'ordre de 0.6 survivant. Nous n'avons aucune explication à ce phénomène ; peut-être un autre modèle serait-il approprié.

4.1.8 Application à d'autres types d'automates

Nous avons jusqu'ici supposé que la structure du générateur de pseudo-aléa était celle d'un LFSR ; l'action de ce générateur peut être décrite par sa matrice génératrice. Comme nous l'avons

évoqué au chapitre 1, si le polynôme générateur associé au LFSR est noté $g(x) = \sum_{i=0}^{K} g_i X^i$, la séquence (a) générée par ce LFSR vérifie la récurrence $a_n = \sum_{i=1}^{K} g_i a_{n-i}$ et sa matrice génératrice s'écrit :

$$\mathbf{G} = \begin{pmatrix} 0 & 0 & \dots & 0 & g_K \\ 1 & 0 & \dots & 0 & g_{K-1} \\ 0 & 1 & & \vdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 1 & g_1 \end{pmatrix}.$$

Les $K-1$ premiers bits ne subissent aucune autre opération que celle d'un simple décalage et le dernier bit est exprimé en fonction de l'état précédent et du polynôme générateur, qui est fixe.
Dans le cas du LFSR, la structure de cette matrice ne varie donc pas dans le temps. Observons ce qui se passe localement, au niveau du vecteur d'entrée de la fonction (*i.e.* un état dans le treillis) entre deux instants consécutifs : un décalage est opéré sur les $n-1$ derniers bits et un bit inconnu entre en première position. Chaque état dans le treillis a donc deux successeurs possibles : le vecteur d'entrée a été influencé par une perturbation, en l'occurrence le bit inconnu entrant en première position.

On peut cependant imaginer que la transition d'un état du générateur entre deux instants consécutifs soit plus complexe : il est, par exemple, tout à fait envisageable que chaque bit constituant l'état à l'instant t soit une fonction linéaire non triviale de l'état du générateur à l'instant $t-1$, fonction qui peut, par ailleurs, dépendre du temps.
Ainsi, deux paramètres modifient le fonctionnement de l'algorithme précédent : la dépendance temporelle de la matrice génératrice d'une part, le nombre de perturbations agissant sur le vecteur d'entrée de la fonction entre deux instants consécutifs d'autre part.

Voyons quelles sont les modifications à apporter à l'algorithme pour que celui-ci soit toujours applicable.

Dépendance temporelle de la matrice de transition

Les variations temporelles de la matrice de transition associée au registre ont des répercussions à deux niveaux :

- d'une part, le treillis est, de fait, non stationnaire : à chaque instant t, il faut donc évaluer les successeurs potentiels des états survivants en fonction de $\mathbf{G}(t)$;
- d'autre part, la phase d'identification de l'état initial du LFSR à partir d'un nombre suffisant de bits décodés, dernière phase de l'algorithme, est également affectée : rappelons que notre algorithme est fondé sur la propriété de pouvoir exprimer linéairement tout bit de la séquence (a) en fonction des K premiers bits $a_0, \dots, a_{K-1}$. Dans le cas d'une matrice de transition invariante en temps, ces calculs se font aisément. Il va nous falloir ici tenir compte des variations temporelles de la matrice génératrice.

Nombre de perturbations

Dans le cas classique du LFSR filtré par une fonction dont les entrées sont régulièrement espacées, une seule perturbation modifie le vecteur d'entrée de la fonction entre deux instants « consécutifs ». D'autres automates ont des transitions plus complexes, notamment les automates cellulaires dont nous donnons un exemple sur la figure 4.4 : l'automate cellulaire représenté est un « 90-150 » (voir annexe B pour plus de détails), l'état de chaque cellule à l'instant $t+1$ est

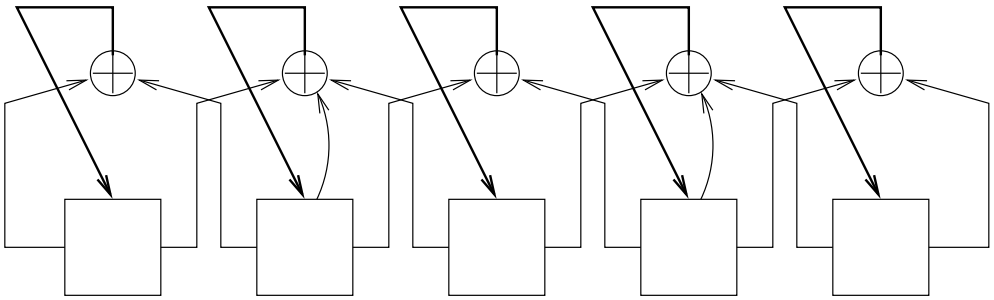


Fig. 4.4 – Exemple d'un automate cellulaire 90-150 à cinq cellules.

conditionné par l'état de ses cellules adjacentes à l'instant t et éventuellement par son propre état.

On peut vérifier que la matrice de transition associée à cet automate a la forme suivante :

$$\mathbf{G} = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

telle que

$$\mathbf{X}(t) = \mathbf{G}\ \mathbf{X}(t-1)\ .$$

Dans le cas général, le vecteur d'entrée de la fonction fera l'objet de deux perturbations entre deux instants consécutifs : chaque état dans le treillis aura donc non plus deux, mais quatre successeurs.

L'avantage d'avoir un nombre limité de successeurs par état réside dans l'élimination successive des survivants ; cela permet de n'obtenir qu'un nombre restreint d'états sur lesquels on peut espérer trouver des formes linéaires constantes, comme dans le cas du LFSR. Malheureusement, dans le cas où le nombre de transitions est supérieur à deux, le nombre de survivants est, de façon générale, trop élevé pour trouver des formes linéaires constantes. Nous avons cependant observé que, dans le cas d'une fonction équilibrée, le nombre de survivants est généralement inférieur à 2^{n-1} : même dans le cas d'un générateur d'aléa tel que chaque état a non plus deux, mais quatre successeurs potentiels, le parcours du treillis permet d'éliminer des survivants. Nous suggérons donc d'appliquer l'algorithme du treillis en cherchant non plus des formes linéaires mais des **formes quadratiques** constantes.
Développons plus avant l'évaluation de ces formes quadratiques.

Reprenons la première des deux approches décrites au paragraphe 4.1.3 : $\Sigma(t) = \{\sigma^1, \ldots, \sigma^{\nu(t)}\}$ étant l'ensemble des survivants à l'instant t, nous allons chercher les solutions $\mathbf{X}$ du système :

$$\mathbf{S_{quad}}\ \mathbf{X} + \mathbf{B} = \mathbf{0}_{\nu(\mathbf{t})}\ ,$$

– $\mathbf{S_{quad}}$ est de la forme :

$$\mathbf{S_{quad}} = \left(\begin{array}{cccc|ccc} \overbrace{\sigma_1^1\sigma_2^1 & \sigma_1^1\sigma_3^1 & \dots & \sigma_{n-1}^1\sigma_n^1}^{\text{termes du second degré}} & \overbrace{\sigma_1^1 & \dots & \sigma_n^1}^{\text{termes du premier degré}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \sigma_1^{\nu(t)}\sigma_2^{\nu(t)} & \sigma_1^{\nu(t)}\sigma_3^{\nu(t)} & \dots & \sigma_{n-1}^{\nu(t)}\sigma_n^{\nu(t)} & \sigma_1^{\nu(t)} & \dots & \sigma_n^{\nu(t)} \end{array} \right) , \quad (4.18)$$

et de taille $\left(\nu(t) \times (\frac{n(n-1)}{2} + n)\right)$,

– $\mathbf{X}$ et $\mathbf{B}$ sont des vecteurs colonne : $\mathbf{X}$ est de taille $(\frac{n(n-1)}{2} + n)$ et $\mathbf{B}$ est le vecteur tout à zéro ou tout à un de taille $\nu(t)$,

puisque la dimension de l'espace des formes quadratiques est égale à $\frac{n(n-1)}{2} + n + 1$.

Nous procédons ensuite de façon identique au cas linéaire. Chaque forme quadratique des n bits qui constituent le vecteur d'entrée est également une forme quadratique des K premiers bits : tout bit de la séquence (a) s'exprime linéairement en fonction des K premiers bits $a_0, \dots, a_{K-1}$ et la composée d'une fonction de degré deux et d'une fonction linéaire est une fonction de degré deux.

Comment exprimer les formes quadratiques du vecteur d'entrée en fonction des K premiers bits de la séquence ?

– Une première façon, assez lourde en terme de calculs, est d'exprimer chaque bit a_t de la séquence en fonction des K premiers bits $a_0, \dots, a_{K-1}$ en utilisant les différentes matrices génératrices associées au générateur d'aléa. On substitue ensuite dans l'expression de la forme quadratique du vecteur d'entrée les bits par leur expression en fonction de $a_0, \dots a_{K-1}$ et on obtient alors l'expression souhaitée.
– Une seconde façon de procéder, un peu moins coûteuse, est la suivante (voir [38] pour plus de détails) : l'état initial du générateur est noté $\mathbf{A}(t = 0)$. Cet état s'exprime de façon linéaire dans une base de taille K, par exemple dans la base canonique, dont le i-ième vecteur sera noté $\mathbf{B_i}(t = 0) = (0, \dots, 0, 1, 0, \dots, 0)$ le 1 étant situé en i-ème position. $\mathbf{B_i}(t)$ représente cet état à l'instant t, obtenu à partir de l'état initial $\mathbf{B_i}(t = 0)$. Ainsi

$$\mathbf{A}(t = 0) = \sum_{i=0}^{K-1} a_i \mathbf{B_i}(t = 0) \ .$$

À chaque instant t, cette relation est vraie :

$$\mathbf{A}(t) = \sum_{i=0}^{K-1} a_i \mathbf{B_i}(t) \ .$$

Or chaque vecteur $\mathbf{B_i}(t)$ s'exprime facilement dans la base canonique :

$$\mathbf{B_i}(t) = \sum_{i=0}^{K-1} b_j(t) \mathbf{B_j}(t = 0) \ ,$$

et

$$\begin{aligned}\mathbf{A}(t) &= \sum_{i=0}^{K-1}\sum_{j=0}^{K-1} a_i b_j(t)\mathbf{B_j}(t=0) \\ &= \sum_{j=0}^{K-1}\Big(b_j(t)\sum_{i=0}^{K-1} a_i\Big)\mathbf{B_j}(t=0)\ . \end{aligned} \tag{4.19}$$

L'expression de toute forme quadratique qui agit sur le vecteur d'entrée de la fonction et donc sur les composantes de $\mathbf{A}(t)$ avec lesquelles il coïncide, sera aisément déduite de l'expression (4.19).
Le procédé ci-dessus nécessite le calcul de $\mathbf{B_j}(t)$ à chaque instant, ce qui est réalisé en évaluant $\mathbf{B_j}(t)$ à l'aide de la matrice génératrice à chaque instant. Il nous faut donc mémoriser les K vecteurs de taille K.

Remarque :

Les algorithmes décrits aux paragraphes 4.1.2, 4.1.3 et 4.1.4 appliqués à un registre filtré classique faisaient apparaître un nombre de survivants de l'ordre de n, $\nu(t) \approx n$. Or, la dimension de la matrice $\mathbf{S_{lin}}$ (cf. équation 4.3), à l'aide de laquelle on peut identifier les formes linéaires constantes, est de taille $\Big(\nu(t) \times n\Big)$. Statistiquement, nous avons donc de grandes chances de trouver des formes linéaires constantes et le succès de nos algorithmes appliqués à un LFSR filtré s'explique ainsi.
En revanche, dans le cas d'un générateur d'aléa plus complexe comme celui que nous avons étudié précédemment, chaque état peut avoir plus de deux successeurs, quatre dans notre exemple. Nous avons observé que le nombre de survivants à chaque instant, s'il diminue, reste néanmoins proche de 2^{n-1}, $\nu(t) \approx 2^{n-1}$. Or, la dimension de la matrice $\mathbf{S_{quad}}$ (cf. équation 4.18) est $\Big(\nu(t) \times (\frac{n(n-1)}{2} + n)\Big)$: statistiquement, ce système aura des solutions lorsque $\nu(t) \leq \frac{n(n-1)}{2} + n$, ou encore :

$$2^{n-1} \leq \frac{n(n-1)}{2} + n$$

soit

$$2^{n-1} \approx \frac{n(n-1)}{2} + n.$$

L'efficacité de l'algorithme du treillis appliqué à un générateur qui introduit deux perturbations dans le vecteur d'entrée de la fonction entre deux instants consécutifs est donc directement reliée au nombre d'entrées n de la fonction : le cas limite est $n = 5$. Au-delà, l'algorithme aura, hors cas particulier, plus de mal à fonctionner.

Nous pouvons généraliser la recherche de fonctions constantes sur les survivants à des **fonctions de degré quelconque**, inférieur à celui de la fonction booléenne néanmoins pour que cette recherche ait un intérêt. Il est à noter que la complexité de cette recherche croît en $\mathcal{O}(n^{\deg(f)})$, où $\deg(f)$ est le degré des fonctions explorées, ce qui devient rapidement prohibitif.

Quelques résultats numériques

Nous présentons quelques résultats de simulation. Ces résultats furent obtenus sur un « 90-150 » de mémoire 64, dont la matrice de transition est donnée dans l'annexe B. Contrairement aux

résultats présentés au paragraphe 4.1.6, les résultats présentés dans le tableau 4.9 n'ont pas été moyennés sur plusieurs tirages : ils ont été obtenus pour une initialisation aléatoire. Nous avons testé plusieurs fonctions et fait varier le nombre d'entrées n afin de mettre en évidence l'importance de ce paramètre sur le succès de l'algorithme.

Pour chaque fonction, nous précisons le nombre d'entrées, la longueur minimale de séquence requise pour identifier la clé secrète. À coté de cette quantité figure entre crochets la longueur minimale obtenue en simulant l'algorithme sur un LFSR de même degré et de polynôme de rétroaction : $g(X) = 1 + X + X^3 + X^4 + X^{64}$, donc de même longueur. Les nombres entre parenthèses dans la dernière colonne donnent pour les algorithmes forward, backward et forward-backward intersectant (F.B.I.) le nombre moyen de survivants associé à chacun d'entre eux.

Fonctions utilisées	**Nombre d'entrées**	**Longueur minimale nécessaire au succès de l'algorithme F.B.I.**	**Nombre de survivants (F, B, F.B.I.)**
f_1	5	544 [114]	(13.3, 13.3, 11.2)
f_6	5	744 [117]	(14.4, 14.1, 12.7)
f_{10}	6	2281 [110]	(26.4, 30.5, 25)
f_9	6	– [125]	(32, 30.4, 30.4)
f_{11}	6	$-^{(*)}$ [118]	(26.8, 31.1, 26.2)
f_{12}	6	4188 [118]	(29.9, 26.3, 24.4)
f_{13}	7	– [84]	(58.7, 64, 58.7)
f_{14}	7	– [124]	(59, 54.4, 50.3)

Tab. 4.9 – Longueur minimale nécessaire à l'identification de l'initialisation d'un automate cellulaire 90-150 de longueur 64 en appliquant l'algorithme forward-backward intersectant afin d'identifier des formes quadratiques constantes de l'état initial. Les quantités entre crochets représentent la longueur minimale requise à l'identification de K combinaisons linéairement indépendantes dans le cas d'un LRSR de même longueur.

Les fonctions f_9, f_{11}, f_{13}, f_{14} qui ne correspondent à aucun nombre sont celles que nous n'avons pas réussi à cryptanalyser, c'est-à-dire celles pour lesquelles il nous a été impossible de trouver des combinaisons des bits de la séquence initiale formant un système de rang plein $r = K + \frac{K(K-1)}{2}$.

Quelques commentaires sur ces résultats : comme prévu par la théorie, l'algorithme se révèle essentiellement efficace pour un nombre d'entrées n inférieur ou égal à 5. Au-delà, le fonctionnement de l'algorithme devient plus erratique :

- pour $n = 6$, quelques cas de convergence peuvent encore être exhibés : la fonction f_{12} en est un exemple ;
- au-delà de $n = 6$, nous avons testé de multiples fonctions et aucune n'a permis de faire converger l'algorithme. Il eut fallu chercher non pas des formes quadratiques constantes sur les états du treillis, mais des formes de degré trois (le parcours dans le treillis permettant tout de même de réduire le cardinal des survivants).

Le cas indexé par une astérisque est particulier : nous n'avons pas réussi à trouver un système de rang r ; en revanche, nous avons pu identifier un système de rang $r - 1$ avec une longueur d'observation $N = 28773$. Les formes quadratiques exhibées dans cette expérience étaient en effet

toujours les mêmes (à des instants différents) et formaient un système dégénéré. Ainsi, par le biais d'un test exhaustif sur la variable dégénérée, ce système pourra quand même être cryptanalysé.

4.1.9 Conclusion

Nous avons présenté un algorithme déterministe de cryptanalyse du registre filtré. Cet algorithme donne d'excellents résultats, puisque lorsque les conditions d'application sont réunies, *i.e.* lorsque les espacements entre les entrées constituent une structure régulière, la longueur de séquence nécessaire à la cryptanalyse d'un registre de longueur K est en $\mathcal{O}(K)$. Nous avons étendu les conditions d'application de cet algorithme aux automates dont l'état interne fait l'objet de deux perturbations entre deux instants consécutifs. L'extension de l'algorithme aux automates dont la transition est fonction du temps a également été étudiée.

Il est à noter que nous nous sommes bornés à la recherche de formes (linéaires ou quadratiques) constantes à chaque instant sur l'ensemble $\Sigma(t)$ de survivants obtenu à cet instant. Rien n'empêche de chercher des formes constantes sur des ensembles de survivants concaténés : par exemple, considérons $\Sigma(t)$ et $\Sigma(t+\delta)$ les ensembles de survivants obtenus aux instants t et $t+\delta$. Nous pouvons chercher des formes constantes sur des états de tailles $n+\delta$ obtenus en concaténant les états de $\Sigma(t)$ et $\Sigma(t+\delta)$ sous réserve qu'il existe un chemin dans le treillis qui les relie : les états obtenus seront donc de la forme $\sigma = (a_t, a_{t+1}, \ldots, a_{t+\delta})$.

Le fonctionnement de notre algorithme repose sur une hypothèse forte qui est que les entrées de la fonction sont connectées de façon régulière dans le générateur de pseudo-aléa. Il sera donc aisé pour qui souhaite empêcher cette attaque de la rendre inapplicable car trop complexe par un choix judicieux des espacements λ_i.

Dans la suite de ce chapitre, nous avons donc tenté d'étendre l'utilisation d'un treillis dans des cas où l'algorithme déterministe que nous venons d'étudier est inapplicable.

4.2 Treillis à métriques souples

Les algorithmes que nous avons présentés jusqu'ici propagent des métriques « dures », au sens où les états sélectionnés le sont avec une probabilité égale à '1' ou '0' : ces algorithmes sont déterministes. Si leur principe permet de retrouver la clé secrète avec une probabilité d'erreur nulle, leur champ d'application est toutefois limité au cas où les espacements entre les entrées de la fonction sont tels que Λ n'est pas trop grand. Rappelons que Λ est le nombre de décalages réguliers à effectuer pour qu'un bit d'entrée coïncidant avec la première composante de la fonction, sorte de la dernière composante. La complexité en mémoire des algorithmes décrits dans le paragraphe 4.1 était en $\mathcal{O}(2^{\Lambda})$. Λ s'exprime en fonction des espacements λ_i selon la formule suivante :

$$\Lambda = 1 + \frac{\sum_{i=0}^{n-2} \lambda_i}{\mathrm{pgcd}(\lambda_i)} \ .$$

Ainsi, le cas où les λ_i sont tous égaux est le plus favorable puisqu'alors $\Lambda = n$. Il paraît cependant très facile de parer à l'attaque du treillis par un choix judicieux de ces paramètres. Nous avons tenté d'étendre l'application de ces algorithmes à des cas où l'algorithme de base, déterministe, est inapplicable. En contrepartie, nous perdons l'avantage des algorithmes déterministes qui identifient la clé avec une probabilité d'erreur nulle (lorsque les conditions sont réunies). Les métriques propagées sur les treillis seront dorénavant des métriques réelles, associées à des probabilités sur chacun des états. Rappelons qu'à partir de K bits linéairement indépendants, l'état initial du générateur peut être identifié à l'aide de l'inversion linéaire d'un système $(K \times K)$

(cf. chapitre 1). Nous allons donc exploiter cette propriété et décoder les K bits les plus fiables, c'est-à-dire ceux dont la probabilité est la plus « éloignée » de $\frac{1}{2}$. Cette façon de procéder peut être reliée à celle mise en oeuvre dans les articles [30, 31, 33, 74].

4.2.1 Cas d'une entrée très « éloignée » des autres : l'algorithme A.T.S.E.

Nous nous intéressons ici au cas où un seul espacement λ_i est très différent des autres, par exemple

$$\begin{aligned} \forall\ i < n-3,\ \lambda_i &= 1 \\ \lambda_{n-2} &\approx\ K-(n-3)\ . \end{aligned} \tag{4.20}$$

Λ est alors presqu'égal à K et l'algorithme du treillis devient inapplicable lorsque K est trop élevé.

Notons cependant que dans le cas où $n-2$ espacements sont égaux (posons $\lambda_i = 1$ pour simplifier), deux vecteurs d'entrée de la fonction consécutifs n'auront plus $n-1$ bits en communs, mais $n-2$. Cette information commune suggère d'exploiter la mémoire de ce nouveau système, en se concentrant sur les $n-1$ entrées régulièrement espacées. Nous proposons donc de construire un treillis ayant 2^{n-1} états, qui correspondent aux $n-1$ premières entrées de la fonction régulièrement espacées. Introduisons les notations suivantes :

- appelons $\sigma' = (\sigma_2, \ldots, \sigma_n)$ un état de taille $n-1$ dans le treillis. Cet état a deux successeurs, $(0, \sigma_2, \ldots, \sigma_{n-1})$ et $(1, \sigma_2, \ldots, \sigma_{n-1})$, et deux prédécesseurs, $(\sigma_3, \ldots, \sigma_n, 0)$ et $(\sigma_3, \ldots, \sigma_n, 1)$;
- notons $\mathrm{chem}^{fwd}[t][\sigma']$, respectivement $\mathrm{chem}^{bwd}[t][\sigma']$, le nombre de chemins dans le sens forward, respectivement backward, qui arrivent à l'état σ'.
  Le nombre d'aller-retours dans le treillis est noté $nbfb^{max}$ et indexé par $nbfb$;
- la concaténation du bit 0 et de l'état σ' de taille $n-1$ sera notée $(0 \mid \sigma')$: le mot résultant est alors de taille n.

L'évaluation du nombre de chemins arrivant à un état σ' correspond en fait au **dénombrement des séquences d'entrée de la fonction** qui sont compatibles avec l'observation et l'effet « mémoire » du système. Le principe de l'algorithme est de propager les probabilités qu'un état soit en entrée de la fonction en utilisant passé et présent dans le treillis.
L'algorithme complet est présenté dans le tableau 4.10.

Quelques remarques :

1. Les états σ' tels que $f(0 \mid \sigma') = f(1 \mid \sigma')$ vont être favorisés le cas échéant : la probabilité que ces états survivent dans le treillis est en effet deux fois supérieure à celle des autres états ; le nombre de chemins incidents à de tels états sera donc pris en compte deux fois.
2. La génération de probabilités à chaque instant se fait par l'intermédiaire d'une normalisation du nombre de chemins arrivant à chaque état par la somme de tous les chemins incidents.
3. La dernière étape peut se faire aisément à partir d'une transformée de Fourier : supposons que nous ayons obtenu à l'instant t les probabilités que chacun des états survive. Notons ces probabilités $P(\sigma')$.

Algorithme du Treillis avec Suppression d'une Entrée (A.T.S.E.)

1. ***Initialisation :***
   - $\forall t,\ \forall \sigma'$
     $\mathrm{chem}^{fwd}[t][\sigma'] = 0$
     $\mathrm{chem}^{bwd}[t][\sigma'] = 1$;
   - $t = 0$:
     $\forall \sigma', \forall b \in \{0,1\}$,
     si $f(b \mid \sigma') = z_0$, $\mathrm{chem}^{fwd}[t][\sigma'] = 1$.
2. ***Déroulement :***
   - Tant que $nbfb < nbfb^{max}$
     (a) *Parcours forward :*
     pour tout $t \in [1, N-1]$,
     $\forall \sigma', \forall\ b$: si $f(b \mid \sigma') = z_t$ et $\mathrm{chem}^{bwd}[t][\sigma'] > 0$

     $$\mathrm{chem}^{fwd}[t][\sigma'] \leftarrow \mathrm{chem}^{fwd}[t][\sigma'] + \mathrm{chem}^{fwd}[t-1][\ pred_b(\sigma')\ ]\ .$$

     $t \leftarrow t+1$
     (b) *Parcours backward :*
     $\mathrm{chem}^{bwd}[N-1][\sigma'] = \mathrm{chem}^{fwd}[N-1][\sigma']$
     pour tout $t \geq 0$,
     $\forall \sigma', \forall\ b$: si $f(b \mid \sigma') = z_t$ et $\mathrm{chem}^{fwd}[t][\sigma'] > 0$

     $$\mathrm{chem}^{bwd}[t][\sigma'] \leftarrow \mathrm{chem}^{bwd}[t][\sigma'] + \mathrm{chem}^{fwd}[t][\ succ_b(\sigma')\ ]\ .$$

     $t \leftarrow t-1$
     (c) $nbfb \leftarrow nbfb + 1$.
3. ***Terminaison :***
   À chaque instant, déterminer la probabilité qu'une forme linéaire d'un état soit constante. Sélectionner les K formes linéaires indépendantes les plus fiables et leur faire subir un décodage à seuil.
   Inverser le système $(K \times K)$ pour retrouver l'état initial.

Tab. 4.10 – Application d'un algorithme aller-retour après suppression d'une entrée : algorithme A.T.S.E.

Alors $\forall \mathbf{u} \in \{0,1\}^{n-1}$,

$$P(\mathbf{u}.\sigma' = 1) = \sum_{\sigma' \text{tel que } \mathbf{u}.\sigma'=1} P(\sigma') \tag{4.21}$$

$$= \frac{1}{2}\big(\widehat{P}(\mathbf{0_{n-1}}) - \widehat{P}(\mathbf{u})\big) \tag{4.22}$$

et de la même façon,

$$P(\mathbf{u}.\sigma' = 0) = \frac{1}{2}\big(\widehat{P}(\mathbf{0_{n-1}}) + \widehat{P}(\mathbf{u})\big) \tag{4.23}$$

où $\forall \mathbf{u} \in 0,1^{n-1}, \widehat{P}(\mathbf{u}) = \sum_{\mathbf{s}} P(\mathbf{s})(-1)^{\mathbf{u}.\mathbf{s}}$.
Illustrons cet algorithme par un exemple.

Exemple 10 *Considérons pour cela une fonction ayant $n = 3$ entrées dont la table des valeurs est donnée par le tableau 4.11.*

vecteur d'entrée	sortie
$0 = 000$	0
$1 = 001$	1
$2 = 010$	0
$3 = 011$	1
$4 = 100$	1
$5 = 101$	1
$6 = 110$	0
$7 = 111$	0

Tab. 4.11 – Table de vérité d'une fonction booléenne à $n = 3$ entrées.

Nous supposons que $\lambda_0 = 1$ et $\lambda_1 = K-2$ où $K = 100$ par exemple, et nous ne normaliserons pas les quantités propagées à chaque instant afin de mieux visualiser le cumul éventuel de ces quantités. L'algorithme du treillis dans sa version initiale est inapplicable et nous entrons dans le cas de l'application de l'algorithme du treillis avec suppression d'une entrée : nous propageons des probabilités sur un plus petit treillis qui ne contient que $2^{n-1} = 4$ états.
Supposons que la séquence (z) observée en sortie de la fonction soit la suivante : 1, 0, 1, 0, 1, 1, ...
La figure 4.5 décrit l'application de l'algorithme à cet exemple.

*À l'instant initial, le bit reçu est un '1'. Or, dans la table des valeurs de la fonction, les deux états du type $(*01)$ appartiennent à l'image réciproque de 1 ; un seul état du type $(*00)$ appartient à l'image réciproque de 1, tout comme les états du type $(*11)$. En revanche, aucun état (*10) ne peut convenir. D'où les quantités $2, 1, 0, 1$ pondérant les états à l'instant initial .*

À l'instant $t = 1$, le bit reçu est un '0'. Les deux états du type $(*10)$ appartiennent à l'image réciproque de '0' : les chemins incidents à l'état (10) sont donc sommés et multipliés par deux. Les états (00) et (11) sont pondérés par la somme des chemins incidents puisqu'un seul des deux états qui leur correspondent appartient à l'image réciproque de 0. Le dernier état (01) ne peut en aucun cas correspondre : il est donc mis à zéro et ainsi de suite.

Nous donnons quelques résultats de simulation dans les tableaux 4.12, 4.13, 4.14 et 4.15. Ces résultats ont été obtenus en faisant des aller-retours dans le treillis, et en initialisant l'état initial du retour par l'état final de l'aller et ainsi de suite. Les espacements entre les entrées de la fonction sont tous égaux à 1, excepté un seul, qui est très élevé. Les probabilités présentées correspondent aux probabilités d'erreur obtenues sur K bits linéairement indépendants décodés. Plus précisément, nous avons sélectionné la probabilité d'erreur minimale sur l'ensemble des combinaisons linéaires des vecteurs d'entrée.

Ces probabilités ont été évaluées sur le dernier chemin « aller » et nous avons moyenné les résultats obtenus sur 100 tirages.

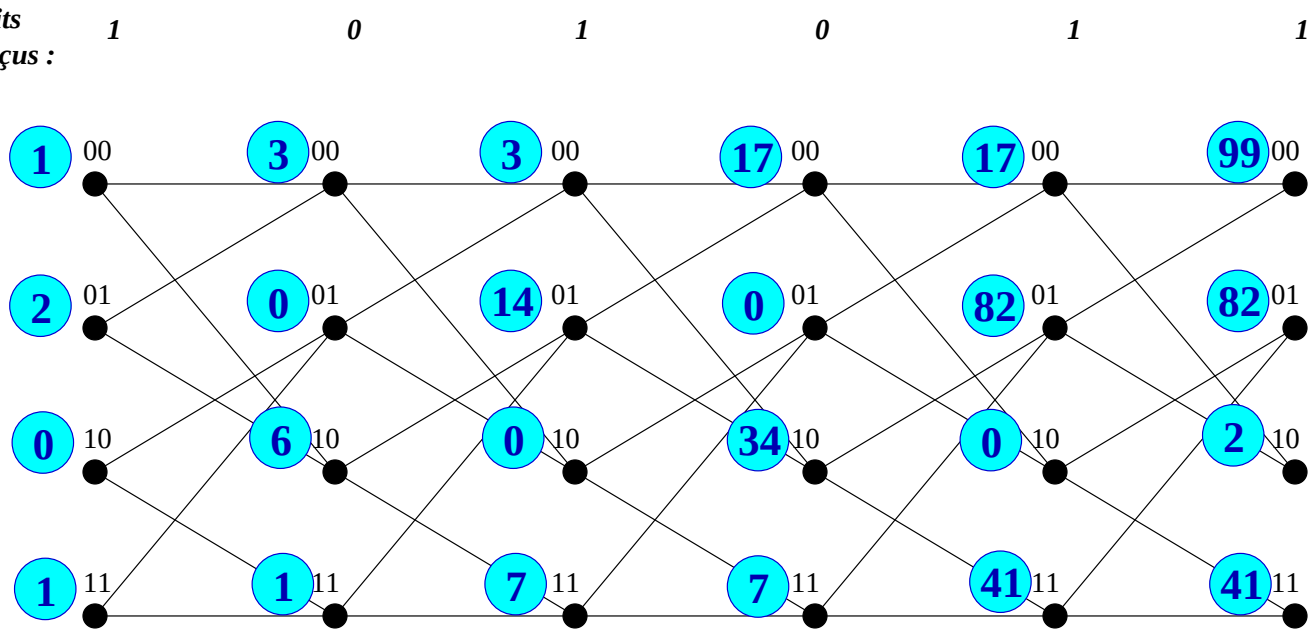


Fig. 4.5 – Déroulement de l'algorithme A.T.S.E sur une fonction à $n = 3$ entrées initialement.

Longueur de la séquence N	Probabilité d'erreur minimale obtenue sur K bits décodés, linéairement indépendants
20000	6.7%
40000	5.5%
60000	4.6%
80000	4.5%
100000	3.6%

Tab. 4.12 – Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_8, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$.

Longueur de la séquence N	Probabilité d'erreur minimale obtenue sur K bits décodés, linéairement indépendants
1000	3.5%
2000	0.6%
3000	$< 10^{-4}$

Tab. 4.13 – Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_{12}, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$.

Les performances présentées ci-dessus sont étonnamment bonnes, puisque rappelons-le, elles ont été obtenues sans utiliser d'équations de parité ; la génération de ces dernières reste un problème « clé » et à ce titre, cet algorithme présente un intérêt notable.

Nous obtenons, sur certaines fonctions (f_1, f_{12}), d'excellents résultats. Cela est lié au com-

Longueur de la séquence N	Probabilité d'erreur minimale obtenue sur K bits décodés, linéairement indépendants
200	10%
300	3.7%
400	0.01%
500	$< 10^{-4}$

Tab. 4.14 – Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 5$ entrées, f_1, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = 1$, $\lambda_3 = K - 4$.

Longueur de la séquence N	Probabilité d'erreur minimale obtenue sur K bits décodés, linéairement indépendants
50000	9.4%
80000	8.2%
120000	8.0%

Tab. 4.15 – Taux d'erreur obtenus en appliquant l'algorithme A.T.S.E. à une fonction à $n = 6$ entrées, f_9, $K = 100$, $\lambda_0 = \lambda_1 = \lambda_2 = \lambda_3 = 1$, $\lambda_4 = K - 5$.

portement des fonctions vis-à-vis de leurs variables d'extrémité, c'est-à-dire de leur première ou dernière variable d'entrée (cf. [40]). Sur la fonction f_1 par exemple, nous constatons que, même sur un treillis réduit, résultant de la suppression d'une entrée, nous avons obtenu des formes linéaires constantes sur certains ensembles de survivants. Cependant, l'algorithme appliqué à la fonction f_9 ne donne pas de très bons résultats.
Les résultats présentés dans ce paragraphe correspondent au cas où les espacements sont presque tous égaux à 1. L'algorithme est également applicable à d'autres systèmes pouvant se ramener via une décimation ou une dégénération artificielle de la fonction au système simulé ci-dessus : par exemple, dans le cas où les espacement sont tous égaux (pas forcément à 1) excepté l'un d'entre eux, ou dans le cas où ils sont légèrement différents, mais tels que le rajout de quelques entrées « artificielles » à la fonction n'entrave pas l'application de l'algorithme.
Par ailleurs, nous pourrions généraliser cet algorithme au cas où $n - j$ entrées sont régulièrement espacées et j entrées ont des espacements premiers entre eux, j pouvant être supérieur à 1. Le treillis construit aura alors 2^{n-j} états. Certaines fonctions, f_1 par exemple, se sont révélées assez robustes devant la suppression d'une entrée, laissant à penser que les performances resteraient acceptables si l'on supprimait une seconde entrée.
Cependant, de façon générale, plus j augmente, plus les performances se dégradent.

4.2.2 Cas où les espacements sont premiers entre eux

Dans le paragraphe précédent, nous avons présenté un algorithme qui propage des métriques souples dans un treillis, dans le cas où le treillis peut ne contenir que 2^{n-1} états. Nous avons cependant voulu étendre le champ d'application du principe du treillis à des cas où les espacements n'ont aucune propriété particulière. Nous supposons dans la suite qu'ils sont premiers entre eux et tels que la valeur de Λ rend tous les algorithmes précédemment étudiés inapplicables.

La première idée, mais qui ne donna aucun résultat, fut la suivante : nous avons considéré un treillis dans lequel chaque état résultait de la concaténation de deux vecteurs d'entrée à

des instants différents, quitte à éventuellement faire coïncider certaines composantes des deux vecteurs d'entrée. Ensuite en opérant des décalages de λ_i puis λ_j et λ_j puis λ_i, c'est-à-dire en parcourant le treillis de deux façons différentes, nous espérions faire diminuer le nombre d'états dans le treillis. Cependant les ensembles de survivants obtenus lors des deux parcours furent identiques et cette piste n'a rien apporté.

Ayant constaté que l'algorithme précédent n'avait aucun effet et souhaitant différencier les états du treillis, nous avons ajouté une contrainte au système en considérant une équation de parité. Supposons par exemple que nous ayons une équation de parité de poids $d = 3$:

$$a_t + a_{t+\theta_0} + a_{t+\theta_1} = 0 \bmod 2. \tag{4.24}$$

D'après le chapitre 1, cette équation est également valable par décalage : si nous décalons l'équation (4.24) de λ_0 puis λ_1, ..., λ_{n-2}, nous obtenons alors l'équation vectorielle :

$$\mathbf{X}(t) + \mathbf{X}(t+\theta_0) + \mathbf{X}(t+\theta_1) = \mathbf{0_n}. \tag{4.25}$$

Nous revenons plus en détails au chapitre 5 sur la génération de telles équations (voir paragraphe 5.1 page 148).

Comment exploiter cette contrainte supplémentaire ?

Un état dans le treillis sera constitué de la concaténation de $\mathbf{X}(t)$ et $\mathbf{X}(t+\theta_0)$ et une condition nécessaire pour que cet état survive est :

$$\begin{aligned} f\Big(\mathbf{X}(t)\Big) &= z_t \\ f\Big(\mathbf{X}(t+\theta_0)\Big) &= z_{t+\theta_0} \\ f\Big(\mathbf{X}(t)+\mathbf{X}(t+\theta_0)\Big) &= z_{t+\theta_1}, \end{aligned} \tag{4.26}$$

c'est-à-dire que la somme de ces deux vecteurs, qui vaut $\mathbf{X}(t+\theta_1)$ soit compatible avec la sortie correspondante observée, $z_{t+\theta_1}$.

Le successeur de l'état $\Big(\mathbf{X}(t)||\mathbf{X}(t+\theta_0)\Big)$ sera donné par $\Big(\mathbf{X}(t+\theta_0)||\mathbf{X}(t+2\theta_0)\Big)$ et ainsi de suite, nous faisons des « bonds » de taille θ_0 dans la séquence. Chaque état $\Big(\mathbf{X}(t)||\mathbf{X}(t+\theta_0)\Big)$ a *a priori* 2^n successeurs qui sont les états du type $\Big(\mathbf{X}(t+\theta_0)||*****\Big)$. Cela laisse peu d'espoir d'avoir un ensemble de survivants suffisamment restreint pour que des formes linéaires soient constantes sur celui-ci.
Le principe de l'algorithme sera ensuite identique à celui de l'algorithme A.T.S.E. décrit dans le paragraphe précédent : sous réserve que l'état en cours satisfasse les conditions du système (4.26), les chemins arrivant à un état sont sommés et convertis en probabilité. Quel que soit l'instant t, dès lors qu'à chacun des états est associée une probabilité de survie, nous évaluons la probabilité qu'une forme linéaire soit constante sur l'ensemble des survivants. Les formes linéaires étant très probablement constantes sont soumises à un décodage à seuil ; une fois que K d'entre elles ont été obtenues et sous réserve qu'elles soient indépendantes, la clé est identifiée par l'inversion d'un système $(K \times K)$.
L'algorithme complet est alors donné dans le tableau 4.16 (cette description omet les phases de normalisation qui permettent de convertir les chemins en probabilité).

Algorithme du treillis souple utilisant une équation de parité

1. ***Initialisation*** :
   à $t = 0$ ne conserver que les états du type $\sigma = (\mathbf{X_1}||\mathbf{X_2})$ vérifiant

$$\begin{aligned} f\Big(\mathbf{X_1}\Big) &= z_0, \\ f\Big(\mathbf{X_2}\Big) &= z_{\theta_0}, \\ f\Big(\mathbf{X_1}+\mathbf{X_2}\Big) &= z_{\theta_1}. \end{aligned}$$

   Les chemins des états vérifiant cette propriété sont mis à '1' ; les autres à zéro.
2. ***Déroulement*** : à t > 0,
   (a) Pour tout état $\sigma = (\mathbf{X_1}||\mathbf{X_2})$, si

$$\begin{aligned} f\Big(\mathbf{X_1}\Big) &= z_t, \\ f\Big(\mathbf{X_2}\Big) &= z_{t+\theta_0}, \\ f\Big(\mathbf{X_1}+\mathbf{X_2}\Big) &= z_{t+\theta_1}, \end{aligned}$$

   alors $chem[\sigma](t) \leftarrow chem[\sigma](t) + chemin[pred(\sigma)](t-1)$.
   (b) Évaluer la probabilité que les différentes formes linéaires calculées sur les survivants soient égales à '1'. S'il existe une forme linéaire dont la probabilité est proche de '1' ou de '0', lui faire subir un décodage à seuil.
3. ***Conclusion*** :
   Une fois que K formes linéaires indépendantes ont été décodées, inverser le système linéaire $(K \times K)$ afin d'identifier l'état initial du LFSR.

Tab. 4.16 – Algorithme du treillis utilisant une équation de parité.

Étude de la complexité :

- *mémoire :* le nombre d'états dans le treillis est de 2^{2n} ;
- *calculs :* à chaque instant, il nous faut évaluer la probabilité qu'une forme linéaire des survivants soit égale à '1' ; cette opération peut être réalisée en utilisant une transformée de Fourier telle que nous l'avons décrite précédemment (voir remarque 3 page 135). La complexité de cette phase est donc en $(Nn \times 2^n)$ où N est la longueur de séquence requise à l'identification de K formes linéaires indépendantes.

L'algorithme exposé ci-dessus peut être généralisé : au lieu de n'utiliser qu'une équation de parité, nous pouvons en utiliser plusieurs (à condition d'en disposer...).
Intéressons-nous par exemple au cas où le cryptanalyste dispose de deux équations de parité.

Chacune de ces équations est associée à un treillis. Nous suggérons deux façons d'exploiter l'existence de ces treillis :

- la première de ces méthodes consiste à **utiliser chacun des treillis de façon indépendante** pour générer de l'information sur les formes linéaires de chacun des vecteurs d'entrée puis à mettre en commun ces informations en fin de déroulement de l'algorithme pour affiner les probabilités obtenues ;
- la seconde méthode est la suite naturelle de ce qui précède : pourquoi ne pas **mettre en commun l'information apportée par chacun des treillis** dans le déroulement même de l'algorithme, en appliquant un **principe « turbo »** ? Nous injectons dans un treillis les informations générées par l'autre et itérons le procédé.

Nous donnons ci-après quelques résultats de simulation. Deux fonctions ont été testées, toutes deux à cinq entrées (ce qui engendre un nombre d'états dans le treillis de 2^{10}) : la première fonction que nous avons considérée est une fonction qui ne présente aucune propriété particulière et dont la table est donnée en annexe (fonction f_7). La seconde fonction testée est la fonction f_1, 1-résiliente.
Le système simulé est associé à un polynôme de rétroaction $g(x) = 1 + x^{37} + x^{100}$ et les équations déduites de ce polynôme le sont par élévations au carré successives (cf. chapitre 2, paragraphe 2.2.1 pour plus de détails). Les espacements entre les entrées de la fonction ont été choisis égaux à $(\lambda_0, \lambda_1, \lambda_2, \lambda_3) = (3, 7, 28, 32)$.

Dans chacun des tableaux 4.17, 4.18, 4.19 et 4.20, nous présentons, pour une longueur de séquence donnée, la probabilité d'erreur minimale obtenue sur les K bits linéairement indépendants les plus fiables (éventuellement issus d'une combinaison linéaire des bits de la séquence initiale). Plus précisément :

- le tableau 4.17 présente les résultats obtenus sur la fonction f_1 en utilisant une seule équation de parité : $a_n + a_{n-37} + a_{n-100} = 0$;
- le tableau 4.18 présente les résultats obtenus sur la fonction f_7 en utilisant une seule équation de parité : $a_n + a_{n-37} + a_{n-100} = 0$;
- le tableau 4.19 présente les résultats obtenus sur la fonction f_1 en utilisant deux équations de parité : $a_n + a_{n-37} + a_{n-100} = 0$ et $a_n + a_{n-74} + a_{n-200} = 0$;
- le tableau 4.20 présente les résultats obtenus sur la fonction f_7 en utilisant deux équations de parité : $a_n + a_{n-37} + a_{n-100} = 0$ et $a_n + a_{n-74} + a_{n-200} = 0$.

Chacun des deux treillis associé à une équation de parité a été utilisé de façon indépendante ; en fin d'algorithme les probabilités issues de chaque treillis ont été multipliées puis normalisées.

Comme les bonds de θ_0 dans le treillis ne fournissent que des probabilités sur les bits dont l'indice est multiple de θ_0, nous avons appliqué cet algorithme $\theta_0 - 1$ fois en incrémentant successivement l'indice du premier bit.

Longueur de séquence	**Taux d'erreur minimal**
5000	13.6 %
8000	12.7 %
10000	13.9 %

Tab. 4.17 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_1, une seule équation de parité est utilisée.

Les derniers résultats présentés dans les tableaux 4.21 et 4.22 ont été obtenus en effectuant un seul aller-retour entre les deux treillis. L'information générée par le premier treillis est injectée

Longueur de séquence	Taux d'erreur minimal
5000	13.5 %
8000	12.3 %
10000	12.2 %

Tab. 4.18 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_7, une seule équation de parité est utilisée.

Longueur de séquence	Taux d'erreur minimal
2000	6.3 %
5000	3.9 %
10000	4 %

Tab. 4.19 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée f_1, deux équations de parité sont utilisées.

Longueur de séquence	Taux d'erreur minimal
2000	8.4 %
5000	6 %
10000	5.8%

Tab. 4.20 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction testée : f_7, deux équations de parité sont utilisées.

Longueur de séquence	Taux d'erreur minimal
2000	2.0%
5000	2.1%
8000	1.5%

Tab. 4.21 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction f_1, les informations générées sur le premier treillis sont injectées dans le second.

Longueur de séquence	Taux d'erreur minimal
2000	3.2%
5000	3.0%
8000	3.0%

Tab. 4.22 – Taux d'erreur obtenus en utilisant un algorithme du treillis propageant des métriques souples sur un aller-retour ; fonction f_7, les informations générées sur le premier treillis sont injectées dans le second.

dans le second de la façon suivante : un état dans un treillis, le premier treillis par exemple, correspond à la concaténation de deux vecteurs d'entrée. Le parcours de ce treillis nous permet de générer de l'information sur chacun des états, laquelle peut ensuite être transformée en

une information sur chaque vecteur d'entrée : considérons l'instant t. Nous indexons par (1) et (2) les probabilités associées respectivement au premier et au second treillis. Par ailleurs nous indexerons par « app » les probabilités *a posteriori* générées par un treillis, et par « ap » les probabilités *a priori* entrées au treillis suivant.

Si σ est un état de taille $2n$ du type

$$\begin{aligned} \sigma &= \Big(\mathbf{X}(t) || \mathbf{X}(t+\theta_0) \Big) \\ &= (\mathbf{X_1} || \mathbf{X_2}) \end{aligned}$$

et si nous disposons des probabilité associées à toutes les valeurs possibles de σ à l'instant t, la probabilité associée au vecteur d'entrée $\mathbf{X}(t)$ sera égale à :

$$P^{(1),\text{app}}\big(\mathbf{X}(t) = (x_0, \ldots, x_{n-1}) \big) = \sum_{\sigma \text{ tel que } \mathbf{X_1}=(x_0,\ldots,x_{n-1})} P(\sigma) \ . \tag{4.27}$$

Nous générons ainsi des probabilités sur tous les vecteurs d'entrée de la fonction $\mathbf{X}(t)$ à partir du premier treillis.

Les états du second treillis sont également constitués de la concaténation de deux vecteurs d'entrée, mais prélevés à des instants différents du premier : les états sont, pour ce qui est du second treillis, du type : $\sigma = \Big(\mathbf{X}(t) || \mathbf{X}(t+\theta_1) \Big)$. La connaissance des probabilités $P^{(1),\text{app}}\Big(\mathbf{X}(t) = (x_0, \ldots, x_{n-1}) \Big)$ issues du premier treillis nous permet de calculer les probabilités associées aux états du second treillis en procédant de la façon suivante :

$$P^{(2),\text{ap}}\Big(\big(\mathbf{X}(t) || \mathbf{X}(t+\theta_1) \big) = (\mathbf{X'_1} || \mathbf{X'_2}) \Big) = \frac{P^{(1),\text{app}}\big(\mathbf{X}(t) = \mathbf{X'_1} \big) \ . \ P^{(1),\text{app}}\big(\mathbf{X}(t+\theta_1) = \mathbf{X'_2} \big)}{\sum_{(x_0,\ldots,x_{n-1}),(y_0,\ldots,y_{n-1})} P^{(1),\text{app}}\big(\mathbf{X}(t) = (x_0, \ldots, x_{n-1}) \big) \ . \ P^{(1),\text{app}}\big(\mathbf{X}(t+\theta_1) = (y_0, \ldots, y_{n-1}) \big)} \tag{4.28}$$

Ces probabilités initialisent le parcours du second treillis. Par ailleurs, en cours de fonctionnement, la probabilité associée à un état déduite du parcours du treillis, *i.e.* des probabilités associées aux états incidents, est multipliée par la probabilité *a posteriori* déduite du premier treillis. Ces quantités sont ensuite normalisées par leur somme, et le procédé est itéré.

En pratique, nous n'avons effectué qu'un seul aller-retour : nous avons en effet observé que les résultats se dégradaient avec le nombre d'aller-retours effectués. Une explication à ce phénomène peut être la suivante : faire plusieurs aller-retours entre les deux treillis correspond à reboucler de l'information déjà utilisée, ce qui introduit des cycles, néfastes au bon fonctionnement des algorithmes itératifs (voir chapitre 3 paragraphe 3.2.2 et chapitre 5 paragraphe 5.3.3).
Les résultats obtenus sont néanmoins assez bons, puisque qu'avec deux équations de parité seulement et des espacements quelconques, nous parvenons à atteindre une probabilité d'erreur de l'ordre de quelques pour-cent. Remarquons toutefois que le cas que nous avons étudié est très favorable dans la mesure où les équations considérées sont de poids $d = 3$. L'algorithme est généralisable à des équations de poids supérieur, mais n'oublions pas que la complexité en mémoire du treillis est en $\mathcal{O}(2^{(d-1)})$, les d vecteurs concaténés étant liés par une équation.
Enfin, nous avons considéré dans ce paragraphe le cas d'une ou deux équations seulement. Une généralisation immédiate de ces algorithmes est de considérer autant d'équations que possible et d'itérer entre tous les treillis qui leur sont associés.

4.3 Conclusion

Dans ce chapitre, nous avons étudié des algorithmes fondés sur l'utilisation de treillis. Dans un premier temps, nous nous sommes placés dans une situation « favorable » : les espacements entre les entrées de la fonction étaient tous identiques, par le biais éventuel d'une transformation du système initial. Les algorithmes déterministes que nous avons présentés se sont avérés extrêmement efficaces, la longueur de séquence nécessaire à la cryptanalyse sans erreur du registre filtré étant de l'ordre de la longueur K du registre. Nous avons ensuite étendu le champ d'application de ces algorithmes à des systèmes ne satisfaisant pas les hypothèses initiales : dans un premier temps, nous avons considéré le cas où une seule entrée est très éloignée des autres. Puis, nous avons envisagé le cas le plus général où les espacements entre les entrées étaient quelconques. Dans ce dernier cas cependant, il nous a fallu introduire une contrainte supplémentaire : nous avons testé l'utilisation d'équations de parité. Ces algorithmes ont tous eu pour principe d'évaluer le nombre de chemins du treillis, c'est-à-dire le nombre de séquences d'entrée cohérentes avec l'observation et la ou les équations vectorielles considérées. Notons que ce principe est assez proche de celui du SOJA décrit au chapitre 5. Ces algorithmes « souples » se sont révélés efficaces : les probabilités d'erreur obtenues sont généralement assez faibles pour pouvoir, le cas échéant, compléter cet algorithme par un test exhaustif permettant finalement de caractériser la clé entière. Cependant, comme l'ont suggéré A. Canteaut et J-P. Tillich [14], les performances de nos algorithmes pourraient être améliorées en utilisant non pas un simple décodage à seuil sur les K bits les plus fiables, mais un autre type de décodeur à entrée souple présentant de meilleures performances.
Par ailleurs, aucune étude théorique des performances des algorithmes de type « treillis souples » n'a été présentée mais il pourrait être intéressant d'approfondir ce point.

Chapitre 5

Le SOJA et ses dérivés

Les différents algorithmes de cryptanalyse exposés dans ce chapitre se caractérisent par une utilisation conjointe de la connaissance de la fonction booléenne et de la structure du registre à décalage.

À partir d'équations scalaires satisfaites par la séquence PN, et en exploitant la structure particulière du registre filtré, le cryptanalyste peut générer des équations vectorielles dont les variables sont les vecteurs d'entrée de la fonction. Les algorithmes de type « SOJA » (Soft Output Joint Algorithm) que nous allons présenter utilisent simultanément les contraintes linéaires sur les vecteurs d'entrée et l'image de ces derniers par la fonction booléenne pour inférer la clé secrète.

Essayons de situer le principe « SOJA » parmi les travaux exposés dans les chapitres précédents (chapitres 3 et 4) :

- rappelons que l'algorithme IDBF (voir chapitre 3), qu'il soit inclus dans un schéma de décodage plus complet ou non, utilise séquentiellement la connaissance de la fonction booléenne et celle du LFSR. La connaissance de la fonction booléenne permet à l'IDBF de générer des probabilités sur les bits de la séquence PN. La connaissance du registre filtré est ensuite utilisée :
  - soit par l'intermédiaire de la récurrence linéaire à laquelle il est associé : les probabilités en sortie de l'IDBF sont soumises à un décodage à seuil. K bits linéairement indépendants étant décodés, la clé est déduite de la résolution d'un système linéaire qui est établi en utilisant la récurrence du LFSR (voir paragraphe 3.2.2 page 101),
  - soit pour générer des équations de poids faibles, dont l'intérêt est de pouvoir intégrer l'IDBF dans une structure « turbo » qui comprend un décodeur de Gallager (voir paragraphe 3.2.3 page 104) ;
- l'algorithme du treillis de type « hard » (cf. chapitre 4, paragraphe 4.1, page 109) identifie également K combinaisons linéaires indépendantes de la séquence d'entrée et, en utilisant la récurrence linéaire du registre, en déduit l'état initial de ce dernier ;
- les algorithmes treillis de type « souple » (cf. chapitre 4, paragraphe 4.2, page 134) sont les plus proches du travail présenté dans ce chapitre dans la mesure où ils font un usage conjoint de la table de la fonction et des équations vectorielles. Un point les distingue cependant des algorithmes « SOJA » : rappelons que le principe des algorithmes de type treillis était d'évaluer le nombre de séquences d'entrée compatibles avec une ou deux équations vectorielles et l'observation en sortie de la fonction. Le principe du « SOJA » sera, pour un vecteur donné, d'évaluer sa distribution statistique en dénombrant les vecteurs qui satisfont toutes les équations auxquelles il appartient et dont l'image est cohérente avec la séquence observée.

Ce chapitre débute par une présentation des hypothèses de travail ; nous dérivons ensuite

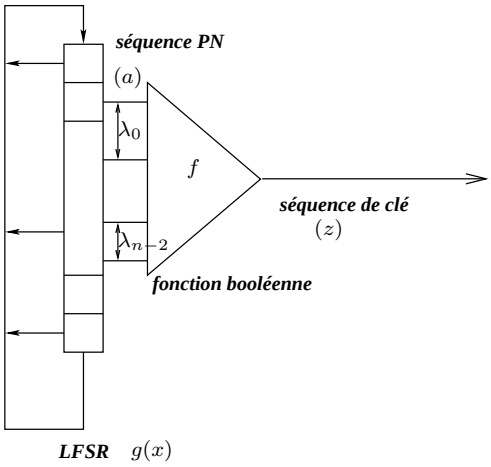


Fig. 5.1 – Registre filtré par une fonction booléenne.

l'algorithme du SOJA (Soft Output Joint Algorithm) et suggérons deux méthodes de décodage à partir des probabilités qu'il génère. Nous entrons ensuite dans le détail du fonctionnement du SOJA : nous en présentons deux versions, l'une vectorielle (SOJA-1), l'autre scalaire (SOJA-2) et les résultats de simulation associés à chacune de ces versions, dans chacune des structures de décodage proposées. Nous sommes ensuite naturellement amenés à présenter une version itérative de cet algorithme qui rappellera fortement les algorithmes de type Belief Propagation présentés au chapitre 3 ; les variables seront non plus des bits, mais des vecteurs. Nous exposerons en dernier lieu un algorithme que nous avons baptisé « Probability-matching » : cet algorithme s'inspire à la fois du SOJA et de la cryptanalyse linéaire [64]. Le principe en est le suivant : la connaissance de la fonction permet souvent d'exhiber des combinaisons linéaires de la séquence d'entrée satisfaites avec une certaine probabilité. Ces combinaisons linéaires peuvent être identifiées en utilisant le principe « SOJA ».

5.1 Hypothèses de travail - Fondement du principe SOJA

Considérons le système cryptographique du registre filtré représenté une nouvelle fois sur la figure 5.1 ; supposons par ailleurs que nous disposions d'un ensemble d'équations de parité de poids d générées à l'aide d'un des algorithmes présentés dans le chapitre 2, paragraphe 2.2.1. Écrivons la m-ième équation de la façon suivante :

$$a_t + a_{t+\theta_{m,1}} + \ldots + a_{t+\theta_{m,d-1}} = 0 \mod 2 \ . \tag{5.1}$$

Rappelons que le vecteur d'entrée de la fonction à l'instant t est noté :

$$\mathbf{X}(t) = (a_t, a_{t+\lambda_0}, a_{t+\lambda_0+\lambda_1}, \ldots, a_{t+\sum_{i=0}^{n-2}\lambda_i}) \ .$$

Il vérifie $f\Big(\mathbf{X}(t)\Big) = z_t$.

De par la structure cyclique du LFSR (voir chapitre 1), l'équation (5.1) est vérifiée quelle que soit la valeur de l'indice t. En particulier, si l'on choisit d'examiner les équations obtenues en décalant l'équation ci-dessus de λ_0, $\lambda_0 + \lambda_1$, …, $\sum_i \lambda_i$ successivement, nous obtenons le système

suivant :

$$\begin{cases} a_t + a_{t+\theta_{m,1}} + \ldots + a_{t+\theta_{m,d-1}} = 0 & \text{aucun décalage, } i.e. \text{ décalage nul} \\ a_{t+\lambda_0} + a_{t+\lambda_0+\theta_{m,1}} + \ldots + a_{t+\lambda_0+\theta_{m,d-1}} = 0 & \text{décalage de } \lambda_0 \\ \ldots & \ldots \\ a_{t+\sum_i \lambda_i} + a_{t+\sum_i \lambda_i+\theta_{m,1}} + \ldots + a_{t+\sum_i \lambda_i+\theta_{m,d-1}} = 0 & \text{décalage de } \sum_i \lambda_i \end{cases}$$

qui peut se réécrire vectoriellement :

$$\mathbf{X}(t) + \mathbf{X}(t+\theta_{m,1}) + \ldots + \mathbf{X}(t+\theta_{m,d-1}) = \mathbf{0_n} \ . \tag{5.2}$$

Nous avons ainsi généré une équation vectorielle à partir d'une équation scalaire et toute équation scalaire peut subir le même sort.
Notons $\mathcal{E}^v(t)$ l'ensemble d'équations vectorielles vérifiées par $\mathbf{X}(t)$ [1] et soit μ_v son cardinal. Nous supposerons pour simplifier que chaque vecteur appartient au même nombre d'équations vectorielles, c'est-à-dire que μ_v est indépendant de t.

L'écriture vectorielle de ces équations apparaît particulièrement intéressante ; en effet, tout vecteur $\mathbf{X}(t)$ se doit maintenant de vérifier deux contraintes conjointes : il doit vérifier les équations vectorielles que nous venons d'exhiber tout en appartenant à l'image réciproque du bit qui lui correspond en sortie de la fonction, comme le montre le système (5.3).

Système d'équations du SOJA

$$\forall m \in [1, \mu_v] \quad \begin{cases} f\Big(\mathbf{X}(t)\Big) = z_t \\ f\Big(\mathbf{X}\left(t+\theta_{m,1}\right)\Big) = z_{t+\theta_{m,1}} \\ \ldots \\ f\Big(\mathbf{X}\left(t+\theta_{m,d-1}\right)\Big) = z_{t+\theta_{m,d-1}} \end{cases} \tag{5.3}$$

$$\text{et} \qquad \mathbf{X}(t) + \mathbf{X}(t+\theta_{m,1}) + \ldots + \mathbf{X}(t+\theta_{m,d-1}) = \mathbf{0_n} \ .$$

[1] $\mathcal{E}^v(t)$ contient l'équation (5.2) mais également toutes les équations du type

$$\mathbf{X}(t-\theta_{m',i}) + \mathbf{X}(t-\theta_{m',i}+\theta_{m',1}) + \ldots + \mathbf{X}(t) + \ldots + \mathbf{X}(t-\theta_{m',i}+\theta_{m',d-1}) = \mathbf{0_n}$$

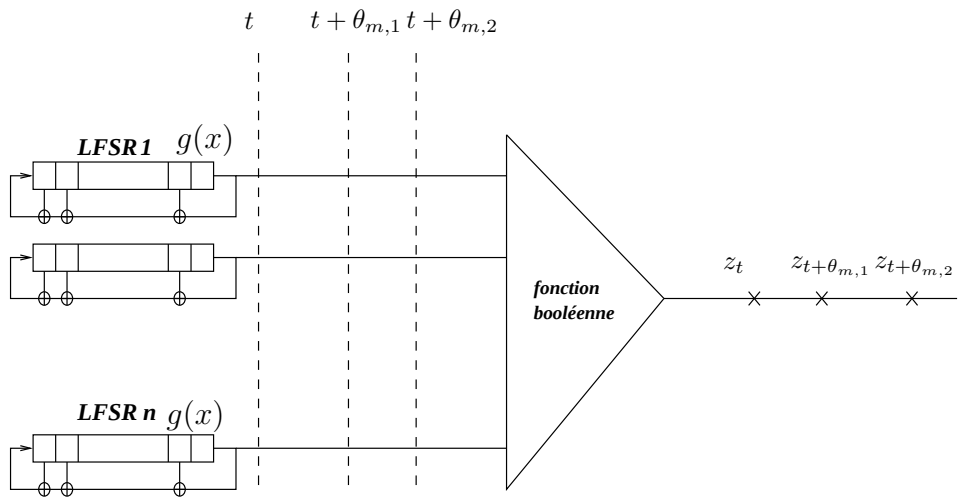


Fig. 5.2 – Mise en évidence d'une équation vectorielle de poids $d = 3$ sur un registre filtré représenté de façon équivalente sous la forme d'un générateur par combinaison.

Tous les travaux présentés dans ce chapitre reposent sur ce système d'équations.

Illustrons notre propos à l'aide de la figure 5.2 : afin de faire apparaître visuellement les contraintes du système (5.3), nous avons remplacé le registre filtré par un générateur à combinaison de registres. Cette structure comprend n LFSRs qui sont tous associés au polynôme de rétroaction $g(x)$ utilisé dans le système étudié (voir figure 5.1). L'initialisation de ces registres est choisie de telle sorte qu'à chaque instant t les vecteurs d'entrée de la fonction $\mathbf{X}(t)$ soient identiques à ceux obtenus dans le cas d'un unique registre filtré. Il suffit pour cela d'initialiser le premier registre par les K premiers bits de la séquence (a), et le i-ième registre $(i > 0)$ par les K premiers bits de la décalée $(\sum_{j=0}^{i-1} \lambda_j)$-ième de la séquence initiale (a).
Cette structure est équivalente à celle du registre filtré et permet une représentation plus claire des contraintes vérifiées par les vecteurs d'entrée. Comme nous le voyons sur cette figure, si nous considérons une équation de poids $d = 3$, les vecteurs d'entrée doivent vérifier les relations :

$$\begin{aligned} &\mathbf{X}(t) + \mathbf{X}(t+\theta_{m,1}) + \mathbf{X}(t+\theta_{m,2}) = \mathbf{0_n} \ , \\ &f\Big(\mathbf{X}(t+j)\Big) = z_{t+j}, \quad \forall j \in \{\ 0, \theta_{m,1}, \theta_{m,2}\ \} \ . \end{aligned}$$

Supposons que la fonction f soit équilibrée : il y a alors *a priori* 2^{n-1} vecteurs d'entrée qui peuvent générer chacun des bits observés. Les équations vectorielles permettent de réduire le cardinal de l'image réciproque de chacun des bits, puisqu'elles constituent une contrainte supplémentaire : seuls certains vecteurs pourront satisfaire conjointement les contraintes du système. Les algorithmes décrits dans ce chapitre sont fondés sur cette remarque.

Nous introduisons deux notations supplémentaires qui nous seront utiles dans la suite du chapitre :

– z^E désigne l'ensemble des bits reçus dont les antécédents sont liés par l'équation vectorielle $E \in \mathcal{E}^v(t)$. Si nous considérons une équation

$$(E) \quad : \mathbf{X}(t) + \mathbf{X}(t+\theta_{m,1}) + \ldots + X(t+\theta_{m,d-1}) = \mathbf{0_n}$$

alors

$$z^E = (z_t, z_{t+\theta_{m,1}} \ldots, z_{t+\theta_{m,d-1}}) \ . \tag{5.4}$$

– $z^{\mathcal{E}}$ est la réunion des z^E où $E \in \mathcal{E}^v(t)$:

$$z^{\mathcal{E}} = \bigcup_{E \in \mathcal{E}^v(t)} z^E \ . \tag{5.5}$$

5.2 L'algorithme SOJA : Soft Output Joint Algorithm

L'algorithme du SOJA, présenté dans ce paragraphe, consiste essentiellement à dénombrer l'ensemble des vecteurs d'entrée qui satisfont les contraintes du système (5.3), afin d'en déduire des probabilités sur les vecteurs d'entrée (SOJA-1) ou sur les bits (SOJA-2). Dans un premier temps, nous allons exposer l'algorithme complet de cryptanalyse ; les calculs des probabilités seront détaillés par la suite, ainsi que l'effet des fonctions « plateau » sur les deux versions du SOJA. Nous présenterons ensuite des résultats de simulation.

5.2.1 Algorithme général

Afin d'avoir une vue globale sur le fonctionnement du SOJA, nous décrivons ci-après son principe général sans entrer, pour l'instant, dans le détail du calcul des probabilités.
Nous introduisons la notation suivante :

$$\Xi_{\ell_{\mathbf{u}}}(t) = P_{app}(a_t^{\ell_{\mathbf{u}}}) \tag{5.6}$$

$$= P\Big(a_t^{\ell_{\mathbf{u}}} = 1 \mid (z), f, \mathcal{E}^v(t)\Big) \ , \tag{5.7}$$

qui représente la probabilité *a posteriori* du t-ième bit de la séquence que nous cherchons à retrouver ; comme dans le cas du décodage itératif (voir chapitre 3) il pourra être intéressant, selon les propriétés de la fonction, de décoder non pas la séquence (a) directement, mais une séquence dérivée de (a). Cette séquence dérivée est obtenue en appliquant à (a) une forme linéaire, $\ell_{\mathbf{u}}$: la séquence résultante est notée $(a^{\ell_{\mathbf{u}}})$.

L'algorithme général est alors donné dans le tableau 5.1.

L'étape (2a) consiste à générer des probabilités sur la séquence d'entrée, du moins sur une séquence du type $(a^{\ell_{\mathbf{u}}})$, à l'aide de l'une des deux versions du SOJA, le SOJA-1 ou le SOJA-2. Dans l'étape (2b), nous suggérons deux algorithmes de décodage qui utilisent les probabilités générées par le SOJA pour cryptanalyser le système ; décrivons plus en détail ces méthodes que nous avons appelées « SOJA-Gallager » et « SOJA-threshold ».

– **SOJA-Gallager** : cette méthode consiste à appliquer la version δ de l'algorithme Belief Propagation (voir chapitre 3, paragraphe 3.1.5) en utilisant comme « observations » les probabilités $\Xi_{\ell_{\mathbf{u}}}(t)$ générées par le SOJA. Les contraintes utilisées sont les équations de parité dont nous disposons. L'algorithme est appliqué jusqu'à ce qu'un mot de code soit obtenu, *i.e.*, jusqu'à ce que le mot en sortie vérifie toutes les équations de parité (le nombre maximal d'itérations permises est néanmoins borné).

Algorithme général du SOJA

1. ***Entrées*** **:**
   $g(X)$, f, $\{\mathcal{E}^v(t)\}$, $\Big(\{\mathcal{E}^b(t)\}\Big)$, (z).
2. ***Déroulement :***
   (a) Générer les probabilités $\Xi_{\phi_{\mathbf{u}}}(t)$ en utilisant l'une des deux versions du SOJA : SOJA-1 ou SOJA-2.
   (b) À partir des probabilités obtenues, cryptanalyser le système en utilisant le SOJA-Gallager ou le SOJA-threshold.
3. ***Sortie*** **:**
   état initial du générateur PN.

Tab. 5.1 – Algorithme général du SOJA.

– **SOJA-threshold** : cette méthode consiste à classer les bits par ordre de fiabilité, en utilisant les probabilités générées par le SOJA. Le bit $a_t^{\ell_{\mathbf{u}}}$ peut être considéré comme d'autant plus fiable que la probabilité $\Xi_{\ell_{\mathbf{u}}}(t)$ qui lui est associée est proche de 0 ou de 1 ; nous classons donc les $\Xi_{\ell_{\mathbf{u}}}(t)$ de façon décroissante (du bit le plus fiable au bit le moins fiable) :

$$|\Xi_{\ell_{\mathbf{u}}}(t_{i_1}) - 1/2| \geq |\Xi_{\ell_{\mathbf{u}}}(t_{i_2}) - 1/2| \geq \ldots \geq |\Xi_{\ell_{\mathbf{u}}}(t_{i_N}) - 1/2|.$$

les i_j étant deux à deux distincts.
Nous considérons ensuite les K bits les plus fiables qui sont linéairement indépendants (cf. chapitre 4 paragraphe 4.2) et leur appliquons un décodage à seuil, lequel se résume à la procédure suivante :

$$\Xi_{\ell_{\mathbf{u}}}(t) > 1/2 \Longrightarrow a_t^{\ell_{\mathbf{u}}} \text{est décodé en '1'},$$
$$\Xi_{\ell_{\mathbf{u}}}(t) < 1/2 \Longrightarrow a_t^{\ell_{\mathbf{u}}} \text{est décodé en '0'}.$$

Rappelons que tout bit de la séquence $(a^{\ell_{\mathbf{u}}})$ peut s'exprimer linéairement en fonction des K premiers bits d'initialisation $a_0^{\ell_{\mathbf{u}}}, \ldots, a_{K-1}^{\ell_{\mathbf{u}}}$ (voir chapitre 1). Ainsi, après avoir obtenu K bits linéairement indépendants, il suffit d'inverser un système linéaire $(K \times K)$ pour retrouver les K premiers bits de clé.

Nous suggérons d'appliquer l'un ou l'autre de ces décodages ; les résultats de simulation que nous avons obtenus semblent démontrer que la méthode de « seuillage » (SOJA-threshold) est néanmoins la plus efficace, tout en étant la moins complexe.

5.2.2 SOJA-1

Nous détaillons dans un premier lieu l'algorithme du SOJA-1. Cet algorithme est plus précis, « exact » que l'algorithme du SOJA-2, mais il est aussi plus complexe.

Entrée	Sortie
(0,0,0)	0
(0,0,1)	1
(0,1,0)	1
(0,1,1)	1
(1,0,0)	0
(1,0,1)	1
(1,1,0)	0
(1,1,1)	0

Tab. 5.2 – Table de vérité d'une fonction à $n = 3$ entrées.

L'algorithme du SOJA a pour dessein d'évaluer des probabilités sur les bits de la séquence (a) ; ces probabilités constituent ensuite l'entrée « souple » du SOJA-Gallager ou du SOJA-Threshold.

Le principe de l'algorithme du SOJA-1 est d'évaluer dans un premier temps des probabilités sur les vecteurs d'entrée de la fonction booléenne, afin d'en déduire des probabilités sur la séquence (a). La première étape du SOJA-1 consiste donc à dénombrer l'ensemble des vecteurs qui satisfont conjointement les deux conditions du système (5.3) : pour tout d-uplet $(z_t, \ldots, z_{t+\theta_{m,d-1}})$, nous énumérons dans la table de vérité de la fonction, les vecteurs qui correspondent au d-uplet reçu et dont la somme est le vecteur nul.
Illustrons notre propos sur un exemple simple.

Exemple 11 *Considérons une fonction à $n = 3$ entrées, dont la table de vérité est donnée dans le tableau 5.2. Supposons que nous disposions d'un ensemble d'équations de poids $d = 3$, et que pour l'une d'entre elles,*

$$\mathbf{X}(t) + \mathbf{X}(t + \theta_{m,1}) + \mathbf{X}(t + \theta_{m,2}) = \mathbf{0_n} \ , \tag{5.8}$$

nous ayons reçu : $z_t = 1, z_{t+\theta_{m,1}} = 0, z_{t+\theta_{m,2}} = 1$. Intéressons-nous, par exemple, à $\mathbf{X}(t)$.
Sans utiliser d'équations vectorielles, les quatre triplets appartenant à l'image réciproque du bit reçu $z_t = 1$ sont équiprobables, i.e. :

$$\begin{aligned} P\Big(\mathbf{X}(t) = (0,0,1)\Big) &= P\Big(\mathbf{X}(t) = (0,1,0)\Big) \\ &= P\Big(\mathbf{X}(t) = (0,1,1)\Big) \\ &= P\Big(\mathbf{X}(t) = (1,0,1)\Big) = 1/2^{n-1} = 1/4 \ , \end{aligned}$$

puisque la fonction est équilibrée et que le bit correspondant à $\mathbf{X}(t)$ en sortie de la fonction est égal à '1'. En revanche, en utilisant les équations vectorielles, nous allons pouvoir affiner la connaissance de $\mathbf{X}(t)$. La première phase du SOJA-1 consiste à trouver une probabilité sur les vecteurs d'entrée ; considérons $\mathbf{X}(t)$ et déterminons la distribution de probabilité que lui confère l'équation (5.8).

Pour cela, énumérons l'ensemble des configurations possibles des vecteurs d'entrée :

$\mathbf{X}(t)$:	*(0,0,1)*	*(0,0,1)*	*(0,1,0)*	*(0,1,0)*	*(0,1,1)*
$\mathbf{X}(t+\theta_{m,1})$:	*(0,0,0)*	*(1,0,0)*	*(0,0,0)*	*(1,1,1)*	*(0,0,0)*
$\mathbf{X}(t+\theta_{m,2})$:	*(0,0,1)*	*(1,0,1)*	*(0,1,0)*	*(1,0,1)*	*(0,1,1)*

$\mathbf{X}(t)$:	*(0,1,1)*	*(1,0,1)*	*(1,0,1)*	*(1,0,1)*	*(1,0,1)*
$\mathbf{X}(t+\theta_{m,1})$:	*(1,1,0)*	*(0,0,0)*	*(1,0,0)*	*(1,1,0)*	*(1,1,1)*
$\mathbf{X}(t+\theta_{m,2})$:	*(1,0,1)*	*(1,0,1)*	*(0,0,1)*	*(0,1,1)*	*(0,1,0)*

Il y a dix triplets de vecteurs d'entrée dont la somme est le vecteur nul et qui correspondent à l'observation $(1,0,1)$. *Énumérant les différentes valeurs possibles de* $\mathbf{X}(t)$ *sur ces dix configurations, nous pouvons en déduire que la distribution de probabilité du vecteur* $\mathbf{X}(t)$ *est la suivante :*

$$\begin{aligned} P\Big(\mathbf{X}(t)=(0,0,1)\Big) &= P\Big(\mathbf{X}(t)=(0,1,0)\Big) = P\Big(\mathbf{X}(t)=(0,1,1)\Big) = \frac{2}{10} = \frac{1}{5} \\ P\Big(\mathbf{X}(t)=(1,0,1)\Big) &= \frac{4}{10} = \frac{2}{5} \end{aligned}$$

où ces probabilités sont conditionnées par l'existence de l'équation vectorielle (5.8). Nous constatons que cette seule équation (5.8) a permis d'accroître la connaissance de $\mathbf{X}(t)$, *sa distribution de probabilité étant moins « lisse » qu'avant. En utilisant l'ensemble des équations auxquelles* $\mathbf{X}(t)$ *appartient, nous pourrons espérer avoir une information encore plus précise sur la valeur de* $\mathbf{X}(t)$.

Décrivons à présent le calcul des probabilités sur les vecteurs d'entrée dans un cadre plus général.

Introduisons la quantité :

$$\forall \mathbf{x} \in \{0,1\}^n \quad \Gamma^t(\mathbf{x}) = P\Big(\mathbf{X}(t)=\mathbf{x} \text{ dans } E \mid z^E, f\Big) \tag{5.9}$$

qui, pour une équation vectorielle E donnée, représente la proportion de vecteurs $\mathbf{X}(t)$ égaux à $\mathbf{x}$ et qui satisfont les deux conditions souhaitées. $\mathbf{X}(t)$ appartient généralement à plusieurs équations vectorielles qui apportent toutes une information du type de Γ^t.
Comment intégrer toutes ces informations ?

Supposons que les informations apportées par chacune des équations $E \in \mathcal{E}^v(t)$ soient indépendantes ; cette hypothèse apparaît légitime lorsque les équations de $\mathcal{E}^v(t)$ ne présentent aucun cycle de taille quatre, *i.e.* lorsque l'intersection des supports des équations de $\mathcal{E}^v(t)$ se réduit au seul vecteur $\mathbf{X}(t)$. Nous pouvons alors écrire :

$$\forall t,\ P\Big(\mathbf{X}(t)=\mathbf{x} \mid z^{\mathcal{E}}, f\Big) = \frac{\prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{x})}{\sum_{\mathbf{y}} \prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{y})}\ , \tag{5.10}$$

qui représente la probabilité *a posteriori* totale du vecteur $\mathbf{X}(t)$ qu'il est possible d'évaluer et qui s'exprime de façon simple en fonction des Γ^t. Dans l'exemple 11, Γ^t a été évalué en dénombrant dans la table de vérité de la fonction les vecteurs qui satisfont les conditions souhaitées. Il existe cependant une approche de calculs beaucoup moins naïve qui utilise la transformée de Fourier, et développée dans la proposition suivante. Cette proposition évalue l'expression de $\Gamma^t(\mathbf{x})$ pour

tout $\mathbf{x}$ et tout d-uplet de sortie possible. Nous pourrons ainsi calculer ces quantités et les stocker dans une table : le calcul de la probabilité totale sur $\mathbf{X}(t)$ décrit dans la formule (5.10) en sera facilité.

Afin de simplifier l'écriture de $\Gamma^t(\mathbf{x})$, nous omettrons les indices temporels ; le lecteur doit cependant garder à l'esprit que cette expression doit être rapportée à un vecteur d'entrée $\mathbf{X}(t)$ dans l'une de ses équations vectorielles $E \in \mathcal{E}^v(t)$.

Proposition 22 *Considérons une fonction booléenne f ayant n entrées et supposons que nous disposions d'un ensemble d'équations vectorielles de poids d. Si la fonction est équilibrée, alors, pour tout d-uplet $(z_0, \ldots, z_{d-1}) \in \{0,1\}^d$, $\forall \mathbf{x} \in \{0,1\}^n$, Γ, défini dans l'équation (5.9), s'exprime de la façon suivante :*

$$\Gamma(\mathbf{x}) = 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{2^{(d-1)n+1} + (-1)^{\sum_{i=1}^{d-1} z_i}\ 2.\widehat{b}(\mathbf{x})}{2^{dn} + (-1)^{\sum_{i=0}^{d} z_i} \widehat{h}(\mathbf{0_n})}\ , \tag{5.11}$$

où $b = (\widehat{f_\chi})^{d-1}$ et $h = (\widehat{f_\chi})^d$.

Preuve :
Posons

$$\begin{aligned} \gamma_{\mathbf{z}}(\mathbf{x}) \ &= \ \text{card} \Big\{ (\mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}}) \text{ tels que} \\ &\qquad f(\mathbf{x}) = z_0,\ f(\mathbf{Y^1}) = z_1,\ \ldots, f(\mathbf{x} + \sum_{i=1}^{d-2} \mathbf{Y^i}) = z_{d-1} \Big\}\ . \end{aligned} \tag{5.12}$$

Nous supposons que $(z_0, \ldots, z_{d-1})$ est un d-uplet observable en sortie de la fonction. Il existe alors au moins une configuration possible dans la table de vérité de la fonction : la configuration qui a réellement lieu, celle que l'on cherche à retrouver. Ainsi, $\sum_{\mathbf{y}} \gamma_{\mathbf{z}}(\mathbf{y}) > 0$.
On peut donc écrire :

$$\Gamma(\mathbf{x}) = \frac{\gamma_{\mathbf{z}}(\mathbf{x})}{\sum_{\mathbf{y}} \gamma_{\mathbf{z}}(\mathbf{y})}\ .$$

Développons le calcul de $\gamma_{\mathbf{z}}(\mathbf{x})$:

$$\begin{aligned} \gamma_{\mathbf{z}}(\mathbf{x}) \ &= \ 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \sum_{\mathbf{Y^1}} \cdots \sum_{\mathbf{Y^{d-2}}} 1\!\!1_{\{f(\mathbf{Y^1})=z_1\}}(\mathbf{Y^1}) \times \\ &\qquad \ldots \times 1\!\!1_{\{f(\mathbf{Y^{d-2}})=z_{d-2}\}}\left(\mathbf{Y^{d-2}}\right) \times \\ &\qquad 1\!\!1_{\{f(\mathbf{x}+\mathbf{Y^1}+\ldots+\mathbf{Y^{d-2}})=z_{d-1}\}}\left(\mathbf{x} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-2}}\right). \end{aligned}$$

En posant $1\!\!1_{\{f(\mathbf{Y^i})=z_i\}}(\mathbf{Y^i}) = \left[\frac{1+(-1)^{f(\mathbf{Y^i})+z_i}}{2}\right]$, on obtient :

$$\begin{aligned} \gamma_{\mathbf{z}}(\mathbf{x}) \ &= \ \frac{1}{2^d}\left[1 + (-1)^{f(\mathbf{x})+z_0}\right] \times \\ &\qquad \sum_{\mathbf{Y^1}} \cdots \sum_{\mathbf{Y^{d-2}}} \left[1 + (-1)^{f(\mathbf{Y^1})+z_1}\right] \ldots \left[1 + (-1)^{f(\mathbf{Y^{d-2}})+z_{d-2}}\right] \times \\ &\qquad \left[1 + (-1)^{f(\mathbf{x}+\sum_i \mathbf{Y^i})+z_{d-1}}\right]. \end{aligned}$$

Puis, en posant :

$$\mathbf{T} = \mathbf{x} + \sum_{i=1}^{d-1} \mathbf{Y^i}$$

et en utilisant la relation

$$\forall\ \mathbf{W},\ \mathbf{S}\ \in \{0,1\}^n\ \ \sum_{\mathbf{S}} (-1)^{\mathbf{S}.\mathbf{W}} = 2^n \times 1\!\!1_{\{\mathbf{W}=\mathbf{0_n}\}}(\mathbf{W})\ ,$$

on obtient :

$$\begin{aligned}
\gamma_{\mathbf{z}}(\mathbf{x}) &= \frac{1}{2^d}\left[1+(-1)^{f(\mathbf{x})+z_0}\right] \times \\
&\sum_{\mathbf{Y^1}} \cdots \sum_{\mathbf{Y^{d-2}}} \sum_{\mathbf{T}} \left[1+(-1)^{f(\mathbf{Y^1})+z_1}\right] \cdots \left[1+(-1)^{f(\mathbf{Y^{d-2}})+z_{d-2}}\right] \times \\
&\left[1+(-1)^{f(\mathbf{T})+z_{d-1}}\right] \frac{1}{2^n} \sum_{\mathbf{S}} (-1)^{\mathbf{S}.(\mathbf{T}+\mathbf{x}+\mathbf{Y^1}+\ldots+\mathbf{Y^{d-1}})}\ .
\end{aligned}$$

En développant l'expression ci-dessus, on obtient alors :

$$\begin{aligned}
\gamma_{\mathbf{z}}(\mathbf{x}) &= \frac{1}{2^{n+d}}\left[1+(-1)^{f(\mathbf{x})+z_0}\right] \times \\
&\sum_{\mathbf{S}} \prod_{i=1}^{d-1} \left[2^n 1\!\!1_{\{\mathbf{S}=\mathbf{0_n}\}}(\mathbf{S}) + (-1)^{z_i} \widehat{f_\chi}(\mathbf{S})\right] (-1)^{\mathbf{S}.\mathbf{x}}\ .
\end{aligned}$$

Si la fonction est équilibrée, alors

$$\begin{aligned}
\widehat{f_\chi}(\mathbf{0_n}) &= \sum_{\mathbf{y}} (-1)^{f(\mathbf{y})} \\
&= \mathrm{card}\{\mathbf{y} \mid f(\mathbf{y})=0\} - \mathrm{card}\{\mathbf{y} \mid f(\mathbf{y})=1\} \\
&= 0
\end{aligned}$$

et l'expression ci-dessus se réduit à :

$$\gamma_{\mathbf{z}}(\mathbf{x}) = \frac{1}{2^{n+d}}\left[1+(-1)^{f(\mathbf{x})+z_0}\right]\left[2^{(d-1)n} + (-1)^{\sum_1^{d-1} z_i} \sum_{\mathbf{S}} \widehat{f_\chi}^{d-1}(\mathbf{S})(-1)^{\mathbf{S}.\mathbf{x}}\right].$$

Il vient alors

$$\sum_{\mathbf{y}} \gamma_{\mathbf{z}}(\mathbf{y}) = \frac{1}{2^{n+d}}\left[2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \sum_{\mathbf{S}} \widehat{f_\chi}^{d}(\mathbf{S})\right].$$

Finalement, en posant $b = (\widehat{f_\chi})^{d-1}$ $h = (\widehat{f_\chi})^d$ on obtient

$$\begin{aligned}
\Gamma(\mathbf{x}) &= \frac{\gamma_{\mathbf{z}}(\mathbf{x})}{\sum_{\mathbf{y}} \gamma_{\mathbf{z}}(\mathbf{y})} \\
&= \left[1+(-1)^{f(\mathbf{x})+z_0}\right] \frac{2^{(d-1)n} + (-1)^{\sum_{i=1}^{d-1} z_i} \widehat{b}(\mathbf{x})}{2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \widehat{h}(\mathbf{0_n})}\ ,
\end{aligned}$$

ce qui coïncide avec le résultat énoncé dans la proposition. ∎

La proposition ci-dessus nous permet d'évaluer efficacement l'information apportée par une équation vectorielle et le d-uplet de bits de sortie correspondant à chacun de ses vecteurs.
À l'aide de la formule (5.10), nous obtenons la probabilité globale sur $\mathbf{X}(t)$ qui tient compte de toutes les équations vectorielles auxquelles $\mathbf{X}(t)$ appartient. Nous devons alors convertir ces informations en probabilités sur les bits de la séquence (a) ou sur une séquence dérivée $(a^{\ell_{\mathbf{u}}})$, la forme linéaire $\ell_{\mathbf{u}}$ étant choisie en fonction des propriétés de la fonction booléenne. Les probabilités sur les bits de la séquence que l'on cherche à décoder sont calculées de la façon suivante :
$\forall\ \mathbf{u}\ \in \{0,1\}^n$

$$\Xi_{\ell_{\mathbf{u}}}(t) = P\,(\ \mathbf{u}.\mathbf{X}(t) = 1 \mid z, f\ ) = \sum_{\mathbf{x}/\ell_{\mathbf{u}}(\mathbf{x})=1} \frac{\prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{x})}{\sum_{\mathbf{y}} \prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{y})}\ . \tag{5.13}$$

En particulier, si l'on cherche à décoder la séquence (a), $\mathbf{u}$ sera choisi de la forme

$$\mathbf{u} = (0, 0, \ldots, 1)\ .$$

Il reste ensuite à appliquer l'une des deux techniques de décodage suggérées dans le paragraphe 5.2.1 afin de retrouver la clé $(a_0, \ldots, a_{K-1})$.

Quelques remarques :

1. Nous constatons que lorsque la fonction f est équilibrée, à $\mathbf{x}$ fixé (et donc $z_0 = f(\mathbf{x})$ fixé), la quantité $\gamma_{\mathbf{z}}(\mathbf{x})$ ne dépend que de la parité de la somme : $\sum_{i=1}^{d-1} z_i$. Ce résultat peut paraître surprenant. Essayons de l'expliquer sans utiliser la formule explicite de cette quantité.

   Nous constatons dans un premier temps qu'il est normal que $\gamma_{\mathbf{z}}(\mathbf{x})$ ne dépende que du poids du $(d-1)$-uplet $(z_1, \ldots, z_{d-1})$ et ce, que la fonction soit équilibrée ou non. En effet, si un ensemble de vecteur $(\mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}})$ est dénombré comme une configuration valide, toute version permutée de cet ensemble de vecteurs est également valide, pour un $\mathbf{z'}$ qui correspondra à cette même permutation appliquée aux composantes de $\mathbf{z}$.

   Par ailleurs, supposons que le $(d-1)$-uplet $(\mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}})$ soit dénombré par $\gamma_{\mathbf{z}}(\mathbf{x})$. Considérons deux vecteurs quelconques ayant même image par f, par exemple supposons que $f(\mathbf{Y^1}) = f(\mathbf{Y^2})$. Cela est possible puisque la dépendance de $\gamma_{\mathbf{z}}$ par rapport au seul poids de $\mathbf{z}$ nous autorise à classer les vecteurs $\mathbf{Y^i}$ comme nous le souhaitons.
   Alors, afin de montrer que si f est équilibrée, $\gamma_{\mathbf{z}}(\mathbf{x})$ ne dépend que de la parité du poids de $\mathbf{z}$, il nous suffit de montrer que ces deux vecteurs $\mathbf{Y}^1$ et $\mathbf{Y}^2$ sont en correspondance directe, en bijection, avec deux autres vecteurs $\mathbf{Y'^1}$, $\mathbf{Y'^2}$ qui satisfont l'équation vectorielle et tels que $f(\mathbf{Y'^1}) = f(\mathbf{Y'^2}) = \overline{f(\mathbf{Y^1})}$.
   Soit $\alpha = \mathbf{Y^1} + \mathbf{Y^2}$. Nous allons en fait chercher à montrer que :

   $$\begin{aligned} & \mathrm{card}\Big\{(\mathbf{Y^1}, \mathbf{Y^2}) \text{ tel que } \mathbf{Y^1} + \mathbf{Y^2} = \alpha \text{ et } f(\mathbf{Y^1}) = f(\mathbf{Y^2}) = 0\Big\} \\ = \quad & \mathrm{card}\Big\{(\mathbf{Y^1}, \mathbf{Y^2}) \text{ tel que } \mathbf{Y^1} + \mathbf{Y^2} = \alpha \text{ et } f(\mathbf{Y^1}) = f(\mathbf{Y^2}) = 1\Big\}\ . \end{aligned}$$

   Appelons $A_{0,0}$ le premier ensemble, $A_{1,1}$ le second. Par analogie nous définissons également $A_{0,1}$ comme :

   $$A_{0,1} = \Big\{(\mathbf{Y^1}, \mathbf{Y^2}) \text{ tel que } \mathbf{Y^1} + \mathbf{Y^2} = \alpha \text{ et } f(\mathbf{Y^1}) = 0,\ = f(\mathbf{Y^2}) = 1\Big\}\ ,$$

$$A_{1,0} = \Big\{ (\mathbf{Y^1}, \mathbf{Y^2}) \text{ tel que } \mathbf{Y^1} + \mathbf{Y^2} = \alpha \text{ et } f(\mathbf{Y^1}) = 1, \ = f(\mathbf{Y^2}) = 0 \Big\} \ .$$

Les cardinaux de ces derniers ensembles sont égaux :

$$\mathrm{card} A_{0,1} = \mathrm{card} A_{1,0}$$

puisque si $(\mathbf{Y^1}, \mathbf{Y^2})$ appartient à $A_{0,1}$, $(\mathbf{Y^2}, \mathbf{Y^1})$ appartient à $A_{1,0}$.
Par ailleurs,

$$\begin{aligned} \mathrm{card} A_{0,0} + \mathrm{card} A_{0,1} &= \mathrm{card} \Big\{ \mathbf{Y^1} \text{ tel que } f(\mathbf{Y^1}) = 0 \Big\} \\ &= 2^{n-1} \end{aligned}$$

car la fonction est équilibrée.
De même,

$$\mathrm{card} A_{1,1} + \mathrm{card} A_{1,0} = 2^{n-1}.$$

De ces trois égalités, nous déduisons que

$$\mathrm{card} A_{0,0} = \mathrm{card} A_{1,1} \ .$$

Ainsi, si le poids du vecteur $(z_1, \ldots, z_{d-1})$ varie d'un multiple de 2, la valeur de $\gamma_{\mathbf{z}}(\mathbf{x})$ est inchangée. Cela montre que $\gamma_{\mathbf{z}}(\mathbf{x})$ ne dépend que de la parité de la somme $\sum_{i=1}^{d-1} z_i$.

2. Rappelons que les indices temporels ont été omis dans l'équation (5.11). En réalité, cette quantité est évaluée pour une équation vectorielle (E) : $\mathbf{X}(t) + \ldots + \mathbf{X}(t + \theta_{m,d-1}) = \mathbf{0_n}$, où les vecteurs $\mathbf{X}(t)$,..., $\mathbf{X}(t + \theta_{m,d-1})$ correspondent respectivement aux bits de sortie $z_t, \ldots, z_{t+\theta_{m,d-1}}$. $\mathbf{X}$ peut donc être assimilé à $\mathbf{X}(t)$ et z_0 à z_t.
Par ailleurs, la proposition 22, qui évalue $\Gamma = \Gamma^t = P\Big(\mathbf{X}(t) = \mathbf{x} \mid z^E, f\Big)$ pour le vecteur $\mathbf{X}$, antécédent de z_0, reste valable quel que soit le vecteur de (E) que l'on considère, $\mathbf{X}(t), \ldots, \mathbf{X}(t + \theta_{m,d-1})$, à une permutation des indices près.

3. Nous avons supposé dans la proposition 22 que la fonction booléenne f était équilibrée, l'expression de Γ s'en trouvant alors simplifiée. En utilisant le même procédé de calculs que celui utilisé dans la démonstration de la proposition 22, nous pouvons montrer que, dans le cas général, Γ s'exprime de la façon suivante :

$$\Gamma(\mathbf{x}) = 1\!\!1_{\{f(\mathbf{x}) = z_0\}}(\mathbf{x}) \times \ \frac{2^{(d-1)n+1} + (-1)^{\sum_{i=1}^{d-1} z_i} \ 2.\widehat{b}(\mathbf{x}) + \alpha}{2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \widehat{h}(\mathbf{0_n}) + 2^n \alpha} \tag{5.14}$$

$$\text{avec } \alpha = \sum_{(0,\ldots,0) \prec \mathbf{t} \prec (1,\ldots,1)} 2^{n(d-1-w_H(\mathbf{t}))} \left(\widehat{f_\chi}(\mathbf{0}_n) \right)^{w_H(\mathbf{t})} (-1)^{\mathbf{z}.\mathbf{t}} \tag{5.15}$$

où $w_H(\mathbf{t})$ est le poids de Hamming du vecteur $\mathbf{t}$, et où $\prec$ désigne la relation suivante : $\forall \ \mathbf{t}, \mathbf{s} \in \{0,1\}^n, \mathbf{t} \prec \mathbf{s} \Longleftrightarrow \forall \ 0 \leq i \leq n-1, \ \ t_i < s_i$.
Le terme α se factorisant par $\widehat{f_\chi}(\mathbf{0_n})$, il s'annule lorsque la fonction est équilibrée, et l'on vérifie que l'équation (5.14) se résume dans ce cas à l'équation (5.11).

4. La démonstration de la proposition 22 prouve que

$$\forall d > 0, \sum_{\mathbf{y}} \gamma(\mathbf{y}) > 0 \ .$$

Or il est démontré dans cette même proposition que

$$\sum_{\mathbf{y}} \gamma(\mathbf{y}) = 2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \sum_{\mathbf{S}} (\widehat{f_\chi})^d(\mathbf{S}) \ .$$

Faisons l'hypothèse que les équations sont de poids $d = 2$. Ce cas de figure est peu probable puisqu'il signifie que la période du registre a été décrite au moins une fois. Quoiqu'il en soit, si les équations sont de poids deux, alors l'expression ci-dessus devient

$$\begin{aligned} \sum_{\mathbf{y}} \gamma(\mathbf{y}) &= 2^{2n} + (-1)^{\sum_{i=0}^{1} z_i} \sum_{\mathbf{S}} (\widehat{f_\chi})^2(\mathbf{S}) \\ &= 2^{2n+1} \ , \end{aligned}$$

car d'après l'égalité de Parseval, $\sum_{\mathbf{S}} (\widehat{f_\chi})^2(\mathbf{S}) = 2^{2n}$; de plus, on a toujours l'égalité $z_0 + z_1 = 0 \mod 2$ car si l'équation est de poids $d = 2$, les deux vecteurs d'entrée de la fonction sont égaux.
Lorsque les équations sont de poids strictement supérieur à deux, alors $(-1)^{\sum_{i=0}^{d-1} z_i}$ peut prendre les valeurs +1 et -1. En particulier, lorsque le résultat est -1, nous venons de montrer de façon simple que :

$$\forall d \geq 3, \quad \sum_{\mathbf{S}} (\widehat{f_\chi})^d(\mathbf{S}) < 2^{dn} \ . \tag{5.16}$$

5. Supposons que les équations soient de poids $d = 3$. Γ s'exprime directement en fonction de l'auto-corrélation de la fonction booléenne :

$$\Gamma(\mathbf{x}) \propto \left[2^n + (-1)^{z_1 + z_2} r_f(\mathbf{x})\right] .$$

De plus, le cas d'équations de poids $d = 3$ est extrêmement favorable lorsque l'on raisonne vectoriellement : en effet, cette situation nous permet de détecter très facilement les vecteurs tout à zéro.
Supposons que l'on ait $\mathbf{X}(t) = \mathbf{0_n}$.
Alors, pour tout $m \in [1, \mu_v]$,

$$\begin{aligned} \mathbf{X}(t) + \mathbf{X}(t + \theta_{m,1}) + \mathbf{X}(t + \theta_{m,2}) = 0_n \quad &\Longrightarrow \quad \mathbf{X}(t + \theta_{m,1}) = \mathbf{X}(t + \theta_{m,2}) \\ &\Longrightarrow \quad f\Big(\mathbf{X}(t + \theta_{m,1})\Big) = f\Big(\mathbf{X}(t + \theta_{m,2})\Big) \\ &\Longrightarrow \quad z_{t+\theta_{m,1}} = z_{t+\theta_{m,2}} \ . \end{aligned} \tag{5.17}$$

Toutes les observations des vecteurs qui interviennent dans une équation avec $\mathbf{X}(t)$ seront donc deux à deux égales, comme nous l'avons illustré sur la figure 5.3. Il résulte de cette propriété que les probabilités générées sur les vecteurs nuls vont être particulièrement fines, et que ces vecteurs vont pouvoir être repérés, identifiés très facilement. En effet si la fonction ne possède aucune propriété particulière, exceptée celle d'être équilibrée, nous pouvons modéliser sa sortie en une variable aléatoire prenant les valeurs '0' et '1' avec une probabilité égale à 1/2. Alors, la probabilité que les deux bits de sortie soient

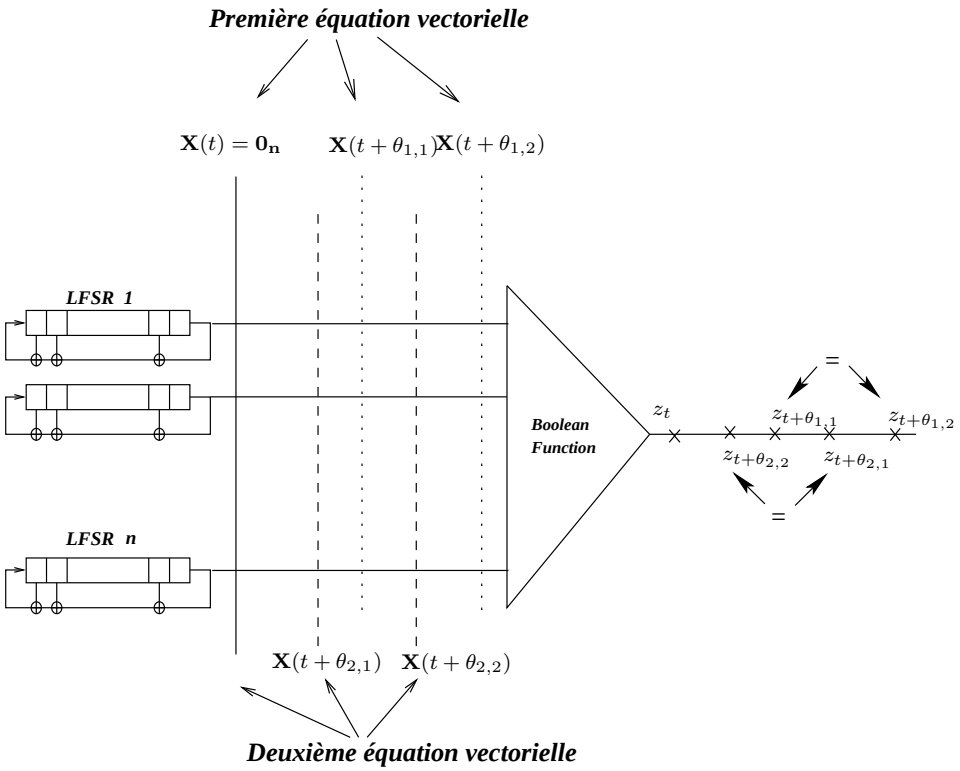


Fig. 5.3 – Détection du vecteur nul en utilisant le SOJA-1 sur des équations vectorielles de poids $d = 3$ que satisfont les vecteurs d'entrée de la fonction booléenne.

constamment égaux (cf. équation (5.17)) lorsque le vecteur est quelconque est en $\frac{1}{2^{\mu_v}}$. Si μ_v est suffisamment grand, la probabilité de ce dernier évènement est donc proche de zéro, à moins que le vecteur ne soit nul.

Nous reviendrons ultérieurement sur cette propriété (page 182), point de départ d'un algorithme dérivé du SOJA qui sera décrit dans le paragraphe 5.4.

6. L'information générée sur les vecteurs d'entrée à l'aide de la formule (5.11) est convertie en probabilités sur les bits d'une séquence $(a^{\ell_{\mathbf{u}}})$ à l'aide de l'équation (5.13). La forme linéaire $\ell_{\mathbf{u}}$ de la séquence initiale est choisie en fonction des propriétés de la fonction. Très souvent, les formes linéaires qui sont privilégiées sont celles de poids 1, car si les espacements λ_i entre les entrées de la fonction sont premiers entre eux, toute forme linéaire non triviale (de poids supérieur à 1) ne correspond qu'à un seul bit de sortie. Les bits issus de formes linéaires de poids 1, en revanche, apparaissent dans la génération de n bits de sortie : notre algorithme génère n valeurs de Ξ par bit. Cette solution s'avère souvent la plus efficace puisqu'elle tient compte de la mémoire intrinsèque au système.

Complexité

Évaluons dorénavant la complexité de calcul et de mémoire du SOJA-1 à l'aune de l'expression (5.11). D'après cette expression, Γ doit être évalué en tout point $\mathbf{x}$ et pour une parité donnée de $z_1 + \ldots + z_{d-1}$, z_0 étant déterminé par la valeur de $\mathbf{x}$. Nous allons donc construire une table qui, pour chaque valeur possible de $(-1)^{z_1+\cdots+z_{d-1}}$, répertorie les valeurs de Γ évalué en chaque

$\mathbf{x}$; lors du déroulement de l'algorithme, il nous suffira alors, pour un vecteur d'entrée donné, de prélever dans la table la valeur appropriée.

La complexité de l'algorithme peut être décomposée selon deux axes principaux :

- **la construction de la table :**
  la table est de taille maximale 2×2^n : pour chacune des deux valeurs de z_0, on va obtenir 2^{n-1} valeurs potentiellement non nulles de Γ, et ce, pour chacune des parités possibles de $\sum_{i=1}^{d-1} z_i$. De plus le calcul de Γ requiert le calcul de la transformée de Fourier de f, pour évaluer les fonctions b et h, et finalement les transformées de Fourier de ces dernières.
  La complexité associée à la phase de construction de la table est donc en $\mathcal{O}(2^n)$ en terme de mémoire et $\mathcal{O}(n2^n)$ en terme de calculs ;
- **le calcul des probabilités $\Xi_{\ell_\mathbf{u}}$:**
  d'après l'expression (5.13), le calcul de $\Xi_{\ell_\mathbf{u}}(t)$ se décompose en trois étapes : tout d'abord, l'évaluation de $\sum z_i \pmod 2$; vient ensuite le calcul du produit des quantités Γ^t sur les μ_v équations auxquelles $\mathbf{X}(t)$ appartient. Finalement, nous sommons sur les $\mathbf{x}$ tels que $\ell_\mathbf{u}(\mathbf{x}) = 1$, qui sont au nombre de 2^{n-1}. La complexité totale de l'évaluation de $\Xi_{\ell_\mathbf{u}}$ est donc $\mathcal{O}(\mu_v d 2^{n-1})$ pour chaque vecteur d'entrée. Le calcul de tous les $\Xi_{\ell_\mathbf{u}}$ sera donc en $\mathcal{O}(N\mu_v d 2^{n-1})$. De plus, pour chaque vecteur $\mathbf{X}(t)$ il nous faut mémoriser les 2^{n-1} valeurs de Γ^t.
  Ainsi, le calcul des distributions de probabilité des vecteurs d'entrée de la fonction booléenne requiert $\mathcal{O}(Nd\mu_v 2^{n-1})$ calculs et a une complexité en mémoire de $\mathcal{O}(N2^{n-1})$.

La complexité en mémoire totale de l'algorithme SOJA-1 est donc

$$\mathcal{O}(N2^{n-1}) \ . \tag{5.18}$$

La complexité en terme de calculs est en

$$\mathcal{O}(Nd\mu_v 2^{n-1}) \ . \tag{5.19}$$

Remarquons que la phase de construction de la table est négligeable devant celle du calcul des probabilités.

Dans le paragraphe suivant, nous proposons une autre version du SOJA. Le principe fondamental est identique : nous utilisons conjointement la connaissance de la fonction et celle des équations de parité. Seul le calcul des probabilités diffère.

5.2.3 SOJA-2

Le principe général de cette version alternative du SOJA est identique à celui du SOJA-1, décrit au paragraphe 5.2.2 : le dénombrement des vecteurs d'entrée de la fonction qui satisfont les conditions du système (5.3) permet de générer des probabilités sur les bits d'une séquence $(a^{\ell_\mathbf{u}})$. Cependant, contrairement au SOJA-1, cette version baptisée SOJA-2 génère directement une probabilité sur les bits de la séquence, sans chercher à évaluer dans un premier temps une information sur les vecteurs d'entrée. Il en résulte un algorithme moins coûteux en terme de calculs et de mémoire, mais aussi moins précis.

Détaillons maintenant le principe du SOJA-2. Considérons une équation vectorielle :

$$\mathbf{X}(t) + \mathbf{X}(t+\theta_{m,1}) + \ldots + \mathbf{X}(t+\theta_{m,d-1}) = \mathbf{0_n} \ .$$

Chacun des vecteurs $\mathbf{X}(t)$, $\mathbf{X}(t+\theta_{m,1})$, ..., $\mathbf{X}(t+\theta_{m,d-1})$ correspond respectivement au bit $z_t, z_{t+\theta_{m,1}}, \ldots, z_{t+\theta_{m,d-1}}$ en sortie de la fonction. Comme dans le cas du SOJA-1, nous énumérons les vecteurs qui se somment à zéro d'une part, appartiennent à l'image réciproque des bits reçus d'autre part. Lors de l'énumération de ces vecteurs, nous évaluons la proportion de '1' et de '0' sur chacune des composantes du vecteur considéré, $\mathbf{X}(t)$ par exemple.

Reprenons l'exemple 11 traité au paragraphe 5.2.2 et déterminons la probabilité générée par le SOJA-2.

Exemple 12 *Nous reprenons exactement le même contexte expérimental que celui traité dans l'exemple 11 page 154 et considérons à nouveau l'ensemble des triplets possibles. Supposons que nous nous intéressions à la troisième coordonnée ($i = 3$) du vecteur $\mathbf{X}(t)$. Dénombrons alors la proportion de '1' sur cette coordonnée : le lecteur peut vérifier qu'il apparaît huit '1' contre deux '0'. Le SOJA-2 nous donne alors une probabilité que le troisième bit soit égal à '1' dans cette équation :*

$$P\Big(X_3(t) = 1 \mid z^E, f\Big) = \frac{8}{10} = \frac{4}{5}$$

Introduisons alors la quantité [2] :

$$\Omega_i^t = P\Big(X_i(t) = 1 \text{ dans } E \ \Big| \ z^E, f\Big) \tag{5.20}$$

où $X_i(t)$ désigne la i-ième coordonnée du vecteur $\mathbf{X}(t)$.
Ω_i^t représente la proportion de vecteurs d'entrée qui satisfont l'équation E, appartiennent à l'image réciproque de z_t et contiennent un '1' sur leur i-ième composante. En comparaison avec le SOJA-1 précédemment décrit, le SOJA-2 consiste à projeter l'information Γ évaluée par le SOJA-1 sur chaque composante des vecteurs d'entrée. La complexité de cette version de l'algorithme est donc moindre.
Tout comme dans le cas du SOJA-1, nous pouvons évaluer ces probabilités de façon efficace par l'intermédiaire de la transformée de Fourier. L'expression de Ω_i^t est détaillée dans la proposition suivante :

Proposition 23 *Considérons une fonction booléenne f ayant n entrées, et supposons que nous disposions d'équations vectorielles de poids d.*
Posons $\mathbf{v^i} = (0, \ldots, 0, v_i = 1, 0, \ldots) \in \{0,1\}^n$, i.e., $\mathbf{v^i}$ est le i-ème vecteur canonique.
Alors, pour tout d-uplet $(z_0, \ldots, z_{d-1}) \in \{0,1\}^d$, si la fonction est équilibrée Ω_i s'exprime de la façon suivante :

$$\Omega_i \quad = \quad \frac{1}{2} - (-1)^{z_1+\ldots z_{d-1}} \times \frac{2^n b(\mathbf{v^i}) + (-1)^{z_0}\Big(b * \widehat{f_\chi}\Big)(\mathbf{v^i})}{2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i}\widehat{h}(\mathbf{0_n})}. \tag{5.21}$$

[2] Remarque : les notations introduites ici diffèrent de celles utilisées dans [57]. Les notations que nous avions alors introduites ne nous satisfaisant plus, nous avons décidé de les modifier et d'introduire de nouvelles notations en accord avec celles du SOJA-1.

Si nous supposons par ailleurs que f est résiliente, l'expression (5.21) se réduit à :

$$\Omega_i = \frac{1}{2} - (-1)^{z_0+\ldots+z_{d-1}} \times \frac{\left(b * \widehat{f_\chi}\right)(\mathbf{v^i})}{2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \widehat{h}(\mathbf{0_n})} \tag{5.22}$$

où $b = (\widehat{f_\chi})^{d-1}$, $h = (\widehat{f_\chi})^d$.

Preuve :
Ω_i peut être calculé en utilisant les mêmes astuces de calcul que pour l'évaluation de Γ.
Nous constatons également que Ω_i et Γ sont liés par la relation suivante :

$$\begin{aligned} \Omega_i &= \sum_{\mathbf{x}/x_i=1} \Gamma(\mathbf{x}) \\ &= \frac{1 - \sum_{\mathbf{x}} (-1)^{\mathbf{v^i}.\mathbf{x}}\, \Gamma(\mathbf{x})}{2}\ . \end{aligned}$$

Posons $B(\mathbf{x}) = \sum_{\mathbf{x}} (-1)^{\mathbf{v^i}.\mathbf{x}}\, \gamma(\mathbf{x})$, où γ a été défini page 155 dans l'équation (5.12), et évaluons la quantité $B(\mathbf{x})$, $\forall\ \mathbf{x}$. Notons dès à présent que

$$\sum_{\mathbf{x}} (-1)^{\mathbf{v^i}.\mathbf{x}}\, \Gamma(\mathbf{x}) = \frac{B(\mathbf{x})}{\sum_{\mathbf{y}} \gamma(\mathbf{y})} = \frac{B(\mathbf{x})}{2^{dn} + \widehat{h}(\mathbf{0_n})}\ . \tag{5.23}$$

D'après la définition de B, on a :

$$B(\mathbf{x}) = \sum_{\mathbf{x}} [1 + (-1)^{f(\mathbf{x})+z_0}] \times \left(2^{(d-1)n+1} + (-1)^{z_1+\ldots+z_{d-1}} 2\widehat{b}(\mathbf{x})\right) (-1)^{\mathbf{v^i}.\mathbf{x}}.$$

En développant cette expression, et en utilisant le fait que :

$$\begin{aligned} \sum_{\mathbf{x}} (-1)^{\mathbf{v^i x}} &= 0, \\ \sum_{\mathbf{x}} (-1)^{f(\mathbf{x})} &= 0 \text{ car } f \text{ est équilibrée,} \end{aligned}$$

nous obtenons

$$B(\mathbf{x}) = 2\left(\sum_{\mathbf{x}} \widehat{b}(\mathbf{x}) (-1)^{\mathbf{v^i}.\mathbf{x}} + \sum_{\mathbf{x}} \widehat{b}(\mathbf{x}) (-1)^{f(\mathbf{x})+\mathbf{v^i}.\mathbf{x}}\right). \tag{5.24}$$

Comme $b(\mathbf{x}) = \widehat{f_\chi}^{d-1}(\mathbf{x})$, nous pouvons écrire

$$\begin{aligned} \sum_{\mathbf{x}} \widehat{b}(\mathbf{x}) (-1)^{\mathbf{v^i}.\mathbf{x}} &= \sum_{\mathbf{x}} \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s}) (-1)^{\mathbf{s}.\mathbf{x}} (-1)^{\mathbf{v^i}.\mathbf{x}} \\ &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s}) \sum_{\mathbf{x}} (-1)^{(\mathbf{v^i}+\mathbf{s}).\mathbf{x}} \\ &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s}) 2^n \mathbb{1}_{\{\mathbf{s}=\mathbf{v^i}\}}(\mathbf{s}) \\ &= 2^n \widehat{f_\chi}^{d-1}(\mathbf{v^i}) \end{aligned} \tag{5.25}$$

et de la même façon,

$$\begin{aligned}\sum_{\mathbf{x}} \widehat{b}(\mathbf{x})(-1)^{\mathbf{v^i}.\mathbf{x}+f(\mathbf{x})} &= \sum_{\mathbf{x}}\sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s})(-1)^{\mathbf{s}.\mathbf{x}}(-1)^{\mathbf{v^i}.\mathbf{x}+f(\mathbf{x})} \\ &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s}) \sum_{\mathbf{x}} (-1)^{(\mathbf{v^i}+\mathbf{s}).\mathbf{x}+f(\mathbf{x})} \\ &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s}) \widehat{f_\chi}(\mathbf{v^i}+\mathbf{s}) \ . \end{aligned} \tag{5.26}$$

Finalement, de par les équations (5.23), (5.24), (5.25) et (5.26), Ω_i prend la forme suivante : :

$$\Omega_i = \frac{1}{2} - (-1)^{z_1+\ldots z_{d-1}} \times \frac{2^n \widehat{f_\chi}^{d-1}(\mathbf{v_i}) + (-1)^{z_0} \sum_{\mathbf{s}} \widehat{f_\chi}(\mathbf{v^i}+\mathbf{s}) \widehat{f_\chi}^{d-1}(\mathbf{s})}{2^{dn} + (-1)^{z_0+\ldots+z_{d-1}} \widehat{h}(\mathbf{0_n})}$$

d'où la formule (5.21).

La formule (5.22) obtenue dans le cas d'une fonction résiliente se déduit immédiatement de l'équation (5.21). Rappelons en effet que $\mathbf{v^i}$ est de poids de Hamming $w_H(\mathbf{v^i}) = 1$. Or, si la fonction est t-résiliente, $\forall$ $\mathbf{v}$ t. q $w_H(\mathbf{v}) \leq t$, $\widehat{f_\chi}(\mathbf{v}) = 0$. En particulier cette relation sera vraie sur le vecteur $\mathbf{v^i}$ dès que l'ordre de résilience est non nul. D'où la relation (5.22). ∎

Afin de tenir compte de toutes les équations vectorielles contenues dans $\mathcal{E}^v(t)$ et en faisant l'approximation que les informations déduites de ces équations sont statistiquement indépendantes, nous calculons une probabilité globale sur chacun des bits de la séquence. La probabilité totale $\Xi_i(t)$ de la i-ième coordonnée du vecteur $\mathbf{X}(t)$ se calcule de la façon suivante :

$$\Xi_i(t) = P\Big(X_i(t) = 1 \mid z^{\mathcal{E}}, f\Big) \tag{5.27}$$

$$= \frac{\prod_{E\in\mathcal{E}^v(t)} \Omega_i^t}{\prod_{E\in\mathcal{E}^v(t)} \Omega_i^t + \prod_{E\in\mathcal{E}^v(t)} (1-\Omega_i^t)} \ . \tag{5.28}$$

Chaque équation vectorielle appartenant à $\mathcal{E}^v(t)$ apporte une information Ω_i^t sur la i-ième composante de $\mathbf{X}(t)$, $X_i(t)$. L'hypothèse d'indépendance entre ces quantités qui sous-tend la formule (5.28) est fausse : la i-ième composante du vecteur $\mathbf{X}(t)$, $X_i(t)$, est liée aux autres composantes $\Big(X_j(t)\Big)_{j\neq i}$ du vecteur, quelle que soit l'équation vectorielle élément de $\mathcal{E}^v(t)$ considérée.

Quelques remarques :

1. Comme dans le cas du SOJA-1, la proposition 23 concerne la j-ième composante de l'antécédent de z_0. Cette expression reste valable pour tous les antécédents des bits de sortie, après permutation des indices.
2. La proposition 23 suppose que la fonction est équilibrée. Si ce n'est pas le cas, en reprenant la démonstration de cette dernière proposition, nous constatons qu'un terme supplémentaire

apparaît, puisque dorénavant, $\widehat{f_\chi}(\mathbf{0_n}) \neq 0$. L'expression de Ω_i est alors la suivante :

$$\Omega_i = \frac{1}{2} - (-1)^{z_1+\dots z_{d-1}} \times \frac{2^n b(\mathbf{v^i}) + (-1)^{z_0} b * f(\mathbf{v^i}) + (-1)^{z_0+\dots+z_{d-1}} 2^{(d-1)n} \widehat{f_\chi}(\mathbf{0_n})}{2^{dn} + (-1)^{z_0+\dots+z_{d-1}} \widehat{h}(\mathbf{0_n})}. \tag{5.29}$$

3. Soulignons l'absence d'indexation des quantités ci-dessus par une forme linéaire de la séquence initiale, contrairement à ce qui fut fait dans le cas du SOJA-1 (voir expression de $\Xi_{\ell_\mathbf{u}}$ page 157). La forme linéaire était déterminée par les propriétés de la fonction. Dans le cas du SOJA-2, les hypothèses d'indépendance entre les quantités Ω_i^t sont assez grossières, et les résultats obtenus en considérant des formes linéaires ne sont pas satisfaisants.

4. La remarque qui suit est à rapprocher de la remarque 6 faite dans le cas du SOJA-1 (voir page 160). Considérons le bit a_t ; ce bit appartient à n vecteurs d'entrée différents, et est donc successivement première composante du vecteur $\mathbf{X}(t)$, seconde composante du vecteur $\mathbf{X}(t-\lambda_0)$, ..., n-ième composante du vecteur $\mathbf{X}(t-\sum \lambda_i)$. n informations du type $\Xi_i(a_t)$ sont donc générées, une pour chaque valeur de $0 < i \leq n$. La probabilité *a posteriori* totale qui est générée sur chacun des bits de la séquence (a) est donc :

$$P_{app}(a_t) \propto \prod_{i,h \mid X_i(h)=a_t} \Xi_i(h) \ .^3 \tag{5.30}$$

Complexité

À l'aide de la proposition 23, nous pouvons évaluer la complexité du calcul des probabilités qu'effectue le SOJA-2. Pour un i donné, et pour chacune des deux valeurs de z_0, nous allons obtenir deux valeurs de Ω_i, correspondant chacune à une parité de $\sum_{j=1}^{d-1} z_j$. La table sera donc de taille $4n$. Le calcul des transformées de Fourier ($\widehat{f_\chi}$ et $\widehat{h}$) se fait en $\mathcal{O}(n2^n)$ et le calcul de h et b en $\mathcal{O}(d2^n)$ (la mémorisation des fonctions requiert un espace mémoire en $\mathcal{O}(2^n)$). Pour chaque valeur de i, il faudra ensuite calculer le produit de convolution entre b et $\widehat{f_\chi}$ qui se fait en $\mathcal{O}(2^n)$ opérations.
La complexité en mémoire du SOJA-2 est

$$\mathcal{O}(4n + 2^n) \approx \mathcal{O}(2^n) \tag{5.31}$$

La complexité en terme de calculs est de

$$\mathcal{O}(n2^n + d2^n + n2^n + Nd\mu_v) \approx \mathcal{O}(Nd\mu_v) \tag{5.32}$$

où le terme $Nd\mu_v$ provient du calcul de $P_{app} \propto \prod_{i,h \mid X_i(h)=a_t} \Xi_i(h)$ (cf. formule (5.28) et (5.30)).

[3] Le signe proportionnel signifie « à une normalisation près », comme cela est couramment pratiqué en décodage itératif. Il ne signifie pas que toutes les quantités $P_{app}(a_t)$ sont proportionnelles à $\prod_{i,h \mid X_i(h)=a_t} \Xi_i(h)$ avec le même coefficient de proportionnalité.

5.2.4 Cas particulier des fonctions « plateau»

Pourquoi nous sommes-nous intéressés à ce type de fonctions booléennes ?
Rappelons tout d'abord ce qu'est une fonction « plateau » :
Définition : Une fonction booléenne est dite « plateau » si et seulement si

$$\exists\, r \in \mathbb{N} \ \text{ tel que } \ \forall \mathbf{u},\ \widehat{f_\chi}^2(\mathbf{u}) = 0 \ \text{ ou } \ 2^{2n-r}.$$

Les termes de la transformée de Fourier d'une fonction booléenne plateau sont donc soit nuls, soit égaux en valeur absolue. D'après la définition ci-dessus et celle de la transformée de Fourier à valeurs dans $\mathbb{Z}$, le paramètre r qui caractérise la fonction plateau se doit d'être pair.
De plus, en vertu de l'égalité de Parseval (voir chapitre 1 , paragraphe 1.4, équation 1.18 page 14),

$$\sum_{\mathbf{u}} \widehat{f_\chi}^2(\mathbf{u}) = 2^{2n}.$$

La transformée de Fourier d'une fonction « plateau » comporte donc 2^r valeurs non nulles, qui sont toutes égales à $2^{n-\frac{r}{2}}$ en valeur absolue.

Revenons alors à la modélisation de la fonction booléenne en canal binaire symétrique (cf. chapitre 1 et chapitre 3 équation 1.50). La probabilité de transition du canal équivalent est déterminée à partir du spectre de la fonction :

$$\begin{aligned} p_u &= \frac{1}{2} + \frac{\widehat{f}(\mathbf{u})}{2^n} \\ &= \frac{1}{2} - \frac{\widehat{f_\chi}(\mathbf{u})}{2^{n+1}}, \end{aligned}$$

où $\mathbf{x} \longrightarrow \mathbf{u}.\mathbf{x}$ est la forme linéaire de la séquence d'entrée que l'on cherche à décoder, et qui est choisie en fonction des caractéristiques de la fonction. Les seules formes linéaires de la séquence d'entrée qui sont décodables, sont celles qui vérifient $\widehat{f}(\mathbf{u}) \neq 0$, *i.e.* telles que $p_u \neq \frac{1}{2}$.
Supposons maintenant que la fonction f soit « plateau » : les formes linéaires qui vérifient $\widehat{f}(\mathbf{u}) \neq 0$, vérifient par définition $\widehat{f_\chi}^2(\mathbf{u}) = 2^{2n-r}$ et correspondent à des valeurs de la transformée de Fourier égales en valeur absolue.
Il existe donc une valeur p_0 telle que toutes les probabilités de transition des formes linéaires décodables sont soit de la forme $p_u = p_0$ soit de la forme $p_u = 1 - p_0$. Ces deux cas de figure sont, en terme de décodage, des problèmes identiques, une séquence bruitée avec une probabilité p_u se déduisant d'une séquence bruitée avec une probabilité $1 - p_u$ par simple complémentation. Plaçons-nous dans un contexte de décodage itératif de ces formes linéaires : il existe 2^r formes linéaires équidistantes de la séquence reçue (ou de son inverse). Il existe donc 2^r solutions au décodage, aucune forme linéaire n'est favorisée et la convergence de l'algorithme en est affectée.

Les fonctions « plateau » apparaissent donc comme difficilement cryptanalysables, et c'est pourquoi nous nous sommes intéressés à ces fonctions.

Effet des fonctions « plateau » sur le SOJA-1

Proposition 24 *Si la fonction booléenne f est plateau et si les équations de parité sont de poids impair, alors*

$$\Gamma(\mathbf{x}) = [1 + (-1)^{f(\mathbf{x})+z_0}] \frac{2^{(d-1)n} + (-1)^{\sum_1^{d-1} z_i} \times 2^{(2n-r)(p-1)} r_f(\mathbf{x})}{2^{dn} + (-1)^{\sum_0^{d-1} z_i} \times 2^{(2n-r)p+n} f_\chi(\mathbf{0_n})}$$

où $d = 2p + 1$, *et* $\{\widehat{f_\chi}(\mathbf{u})\} = \{0, \pm\, 2^{n-\frac{r}{2}}\}$.

Preuve :
Rappelons l'expression générale de $\Gamma(\mathbf{x})$:

$$\Gamma(\mathbf{x}) = \left[1 + (-1)^{f(\mathbf{x})+z_0}\right] \frac{2^{(d-1)n} + (-1)^{\sum_1^{d-1} z_i} \widehat{b}(\mathbf{x})}{2^{dn} + (-1)^{\sum_{i=0}^{d-1} z_i} \widehat{h}(\mathbf{0_n})}$$

où $b = \widehat{f_\chi}^{d-1}$ et $h = \widehat{f_\chi}^{d}$.
Si l'on suppose que les équations de parités sont de poids d impair, c'est-à-dire qu'il existe un entier p tel que $d = 2p + 1$, alors

$$\begin{aligned}
\widehat{b}(\mathbf{x}) &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d-1}(\mathbf{s})(-1)^{\mathbf{s.x}} \\
&= \sum_{\mathbf{s}} \widehat{f_\chi}^{2p}(\mathbf{s})(-1)^{\mathbf{s.x}} \\
&= [2^{2n-r}]^{p-1} r_f(\mathbf{x})
\end{aligned}$$

où r_f est la fonction d'auto-corrélation associée à la fonction f définie au chapitre 1 (voir paragraphe 1.4, équation 1.36, page 17). De plus,

$$\begin{aligned}
\widehat{h}(\mathbf{0_n}) &= \sum_{\mathbf{s}} \widehat{f_\chi}^{d}(\mathbf{s})(-1)^{\mathbf{s.x}} \\
&= \sum_{\mathbf{s}} \widehat{f_\chi}^{2p+1}(\mathbf{s})(-1)^{\mathbf{s.x}} \\
&= [2^{2n-r}]^{p} \times 2^n f_\chi(\mathbf{0_n}).
\end{aligned}$$

On a donc

$$\Gamma(\mathbf{x}) = [1 + (-1)^{f(\mathbf{x})+z_0}] \frac{2^{(d-1)n} + (-1)^{\sum_1^{d-1} z_i} \times 2^{(2n-r)(p-1)} r_f(\mathbf{x})}{2^{dn} + (-1)^{\sum_0^{d-1} z_i} \times 2^{(2n-r)p+n} f_\chi(\mathbf{0_n})}.$$

∎

Proposition 25 *Si la fonction est « plateau » et si les équations de parité sont de poids pair, le SOJA-1 ne permet aucun gain par rapport à l'initialisation classique utilisant les corrélations non nulles de la fonction.*

Preuve :
Si le poids d des équations est pair, $d = 2p$, en utilisant les mêmes astuces de calcul que celles utilisées ci-dessus, et puisque

$$\begin{aligned}
\Gamma(\mathbf{x}) \neq 0 &\iff f(\mathbf{x}) = z_0 \\
&\iff f_\chi(\mathbf{x}) = (-1)^{z_0}
\end{aligned}$$

on montre que

$$\begin{aligned}\Gamma(\mathbf{x}) &= 0 \text{ si } f(\mathbf{x}) \neq z_0 \\ &= \frac{1}{2^{n-1}} \text{ sinon.}\end{aligned}$$

Supposons alors que $f\Big(\mathbf{X}(t)\Big) = 1$ et rappelons que la fonction booléenne est équilibrée.

$$\begin{aligned}\Xi_{\ell_{\mathbf{u}}}(t) &= P(\mathbf{u}.\mathbf{X}(t) = 1|z, f) = \sum_{\mathbf{x}/\ell_{\mathbf{u}}(\mathbf{x})=1} \frac{\prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{x})}{\sum_{\mathbf{y}} \prod_{E\in\mathcal{E}^v(t)} \Gamma^t(\mathbf{y})} \\ &= \sum_{\mathbf{x}/\mathbf{u}.\mathbf{x}=1} \frac{(\frac{1}{2^{n-1}})^{\mu_v} f(\mathbf{x})}{2^{n-1} \times (\frac{1}{2^{n-1}})^{\mu_v}} \\ &= \frac{1}{2^{n-1}} \sum_{\mathbf{x}/\mathbf{u}.\mathbf{x}=1} f(\mathbf{x}) \\ &= \frac{1}{2} - \frac{\widehat{f}(\mathbf{u})}{2^n} \\ &= 1 - p_u\end{aligned}$$

où p_u est la probabilité de transition associée au canal binaire symétrique modélisant le bruitage de la forme linéaire $\mathbf{u} \longrightarrow \mathbf{u}.\mathbf{x}$. La valeur de $\Xi_{\ell_{\mathbf{u}}}(t)$ est donc, dans ce cas, strictement identique à la valeur d'initialisation « classique», c'est-à-dire, utilisant les propriétés du canal de l'algorithme de Gallager (voir chapitre 3, paragraphe 3.1.8, équation 1.50 page 22). On peut donc conclure que lorsque la fonction est « plateau » et que les équations de parité sont de poids pair, le SOJA-1 n'est d'aucune utilité. ∎

Effet des fonctions « plateau » sur le SOJA-2

Proposition 26 *Pour toute fonction « plateau » f,*

$$\forall (p,q) \in \mathbb{N}^2, \quad \forall\ \mathbf{v} \neq \mathbf{0_n} \in \{0,1\}^n, \quad \sum_{\mathbf{s}} \widehat{f_\chi}^{2p+1}(\mathbf{v}+\mathbf{s})\widehat{f_\chi}^{2q+1}(\mathbf{s}) = 0\ . \tag{5.33}$$

Preuve :
Supposons que $p = 0$, $q = 0$. Alors,

$$\begin{aligned}\sum_{\mathbf{s}} \widehat{f_\chi}(\mathbf{v}+\mathbf{s})\widehat{f_\chi}(\mathbf{s}) &= \sum_{\mathbf{s}}\sum_{\mathbf{x}}\sum_{\mathbf{y}} (-1)^{f(\mathbf{x})+f(\mathbf{y})+\mathbf{s}.(\mathbf{x}+\mathbf{y})+\mathbf{v}.\mathbf{x}} \\ &= \sum_{\mathbf{x}}\sum_{\mathbf{y}} (-1)^{f(\mathbf{x})+f(\mathbf{y})+\mathbf{v}.\mathbf{x}} \times 2^n \mathbb{1}_{\{\mathbf{x}=\mathbf{y}\}}(\mathbf{x}) \\ &= 2^{2n}\delta_{\mathbf{0_n}}(\mathbf{v}) \\ &= 0\end{aligned}$$

puisque $\mathbf{v}$ est un vecteur non nul par hypothèse.
f étant une fonction plateau, par définition $\{\widehat{f_\chi}(\mathbf{u})\} = \{0, \pm A\}$, $A = 2^{n-\frac{r}{2}}$.

Posons

$$\begin{aligned} \tau_{\mathbf{v}} \ : \{0,1\}^n &\longrightarrow \{0,1\}^n \\ \mathbf{x} &\longrightarrow \mathbf{x}+\mathbf{v} \ ; \end{aligned}$$

$\tau_{\mathbf{v}}$ est la fonction de translation de vecteur $\mathbf{v}$.

Recalculons la somme précédente en utilisant les propriétés des fonctions « plateau».

$$\begin{aligned} \sum_{\mathbf{s}} \widehat{f_\chi}(\mathbf{v}+\mathbf{s})\widehat{f_\chi}(\mathbf{s}) &= \sum_{\mathbf{s}\in Supp(\widehat{f_\chi})\cap Supp(\widehat{f_\chi} o\tau_{\mathbf{v}})} (-1)^{sgn\left(\widehat{f_\chi}(\mathbf{s})\right)+sgn\left(\widehat{f_\chi}(\mathbf{s}+\mathbf{v})\right)} \times A^2 \\ &= A^2 \sum_{\mathbf{s}\in Supp(\widehat{f_\chi})\cap Supp(\widehat{f_\chi} o\tau_{\mathbf{v}})} (-1)^{sgn\left(\widehat{f_\chi}(\mathbf{s})\right)+sgn\left(\widehat{f_\chi}(\mathbf{s}+\mathbf{v})\right)} \\ &= 0 \ . \end{aligned}$$

Alors, soit p et q deux entiers positifs quelconques,

$$\begin{aligned} & \sum_{\mathbf{s}} \widehat{f_\chi}^{2p+1}(\mathbf{v}+\mathbf{s})\widehat{f_\chi}^{2q+1}(\mathbf{s}) \\ = \ & A^{2(p+q+1)} \times \sum_{\mathbf{s}\in Supp(\widehat{f_\chi}^{2p+1})\cap Supp(\widehat{f_\chi}^{2q+1} o\tau_{\mathbf{v}})} (-1)^{sgn\left(\widehat{f_\chi}^{2p+1}(\mathbf{s})\right)+sgn\left(\widehat{f_\chi}^{2q+1}(\mathbf{s}+\mathbf{v})\right)} \\ = \ & 0 \end{aligned}$$

parce que

$$\begin{aligned} Supp(\widehat{f_\chi}^{2p+1}) &= Supp(\widehat{f_\chi}) \\ Supp(\widehat{f_\chi}^{2q+1} o\tau_{\mathbf{v}}) &= Supp(\widehat{f_\chi} o\tau_{\mathbf{v}}) \\ \forall \mathbf{s}, \ sgn\left(\widehat{f_\chi}^{2p+1}(\mathbf{s})\right) &= sgn\left(\widehat{f_\chi}(\mathbf{s})\right) \ . \end{aligned}$$

∎

Corollaire 2 *Si la fonction booléenne f est plateau et résiliente, le SOJA-2 ne peut être appliqué que dans le cas où les équations de parité sont de poids impair.*

Preuve :
Si la fonction est résiliente, les valeurs de Ω_i sont données par l'équation (5.22) (cf. page 163). Si le poids d de l'équation est pair, il existe un entier p tel que $d = 2p$. Alors, en posant $q = 0$ dans l'équation (5.33) et en se référant à l'équation (5.22), nous constatons que les valeurs de Ω_i sont, dans ce cas, toutes égales à $\frac{1}{2}$.

∎

5.2.5 Résultats de simulation, commentaires

Description des systèmes étudiés

Nous avons appliqué les algorithmes décrits dans ce chapitre à deux systèmes différents, chacun contenant des fonctions « plateau » résilientes.

Pour chacun des systèmes, nous avons comparé les performances des algorithmes suivants :

1. La version δ de l'algorithme Belief Propagation (voir chapitre 3) appliquée à la séquence (z) résultant du filtrage de la séquence PN par la fonction f ; la probabilité de transition du canal binaire équivalent (qui, dans ce cas, ne sert qu'à initialiser l'algorithme de Gallager) est évaluée en fonction du spectre de f.
2. À titre de référence, nous avons appliqué le même algorithme à une séquence (z) résultant non plus du filtrage de la séquence PN par la fonction, mais du bruitage de cette séquence par un canal binaire symétrique ; nous avons choisi la probabilité de transition du canal de telle sorte qu'elle soit égale à celle qui a été précédemment déduite du spectre de la fonction.
3. Les algorithmes liés au SOJA-1 :
   (a) le (SOJA-1)-Gallager ;
   (b) le (SOJA-1)-threshold .
4. Les algorithmes liés au SOJA-2 :
   (a) le (SOJA-2)-Gallager ;
   (b) le (SOJA-2)-threshold.

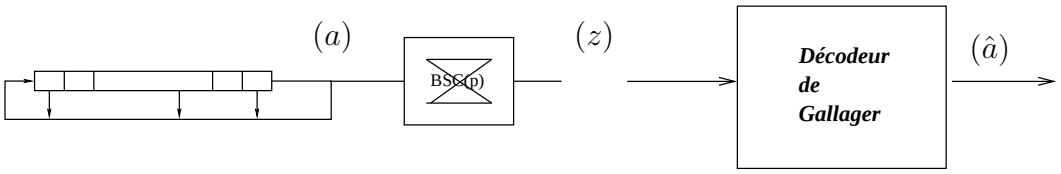
Ces différentes stratégies sont résumées sur la figure 5.4, où $\widehat{a}$ désigne la séquence décodée.

Lorsqu'une fonction booléenne est utilisée, les espacements entre ses entrées, *i.e.* les valeurs des λ_i si l'on se réfère à la figure 5.1, sont choisis premiers entre eux et de telle sorte qu'ils soient répartis sur toute la mémoire du registre à décalage.

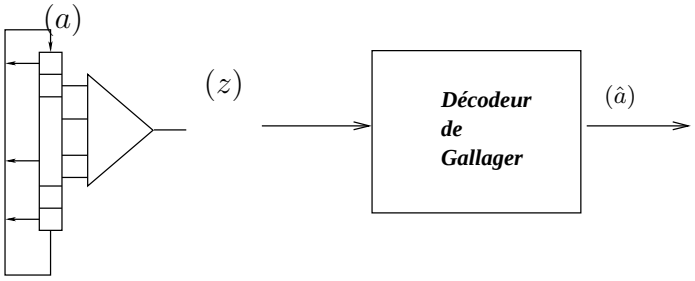
De plus, dans le cas du SOJA-1 comme dans celui du SOJA-2, nous avons choisi de décoder les formes linéaires triviales de poids 1, c'est-à-dire les bits d'entrée de la fonction : comme nous l'avons exposé plus haut (cf. remarques 6 et 4 aux pages 160 et 165 respectivement), cela permet d'obtenir plus d'information sur les bits de la séquence que l'on souhaite décoder, puisqu'un bit de la séquence PN est impliqué dans la génération de n bits de sortie, alors qu'une forme linéaire n'apparaît qu'une seule fois. Par exemple, a_t est une composante des vecteurs $\mathbf{X}(t), \mathbf{X}(t-\lambda_0), \ldots, \mathbf{X}(t-\sum_{i=0}^{n-2}\lambda_i)$, images réciproques de $z_t, z_{t-\lambda_0}, \ldots z_{t-\sum_{i=0}^{n-2}\lambda_i}$ respectivement.

Les résultats sont figurés dans les tableaux 5.3 et 5.4. Nous présentons les performances obtenues pour les différentes stratégies de décodage utilisées en fonction de la longueur d'observation N.
La deuxième ligne contient les résultats obtenus en utilisant l'algorithme de Gallager sur une séquence bruitée par un vrai canal binaire symétrique (stratégie 2). La troisième ligne contient quant à elle les résultats obtenus en appliquant la stratégie 1 sur la fonction booléenne étudiée. Pour ces deux stratégies, les données du tableau représentent le taux de succès des algorithmes, c'est-à-dire la proportion de fois où l'algorithme de Gallager a convergé.
Les quatre dernières lignes présentent les résultats obtenus avec nos algorithmes : la quatrième

Stratégie 1 : modèle du canal binaire symétrique.



Stratégie 2 : la fonction booléenne est considérée comme un BSC



Stratégies 3(a), 3(b) − 4(a), 4(b) : utilisation des APP générées par le SOJA

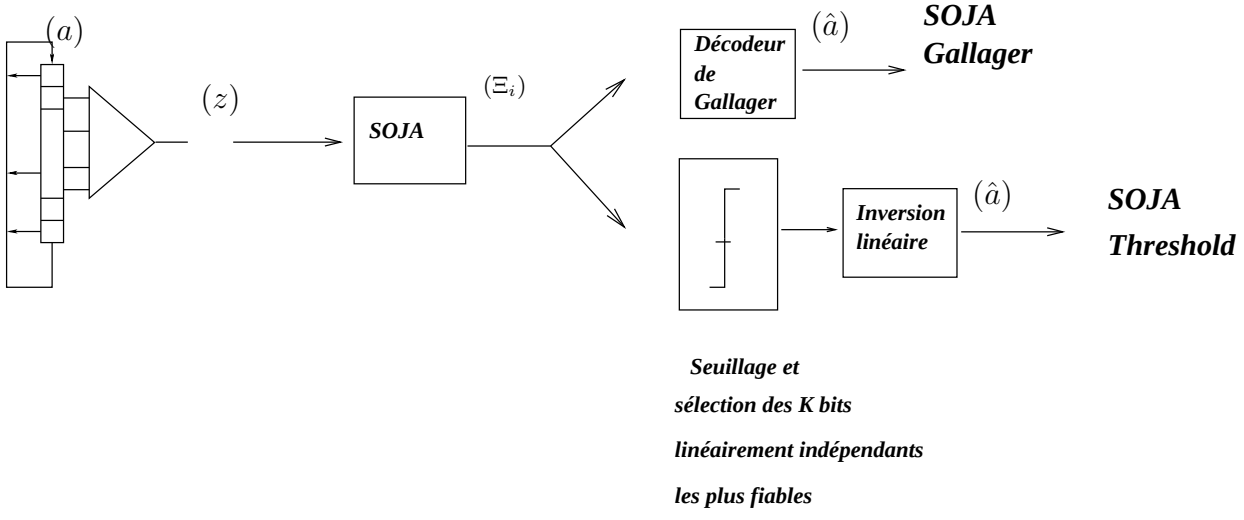


Fig. 5.4 – Représentation des différentes stratégies de décodage comparées : algorithme BP appliqué sur une séquence bruitée par un BSC ou par une fonction booléenne et algorithmes du SOJA suivis d'un décodage BP ou d'une détection à seuil.

ligne et la cinquième ligne contiennent les taux de succès des algorithmes du (SOJA-1)-Gallager, (SOJA-2)-Gallager respectivement (stratégies 3a, 4a).
Les deux dernières lignes présentent le taux d'erreur moyen, ρ_{moy}, obtenu sur les K bits linéairement indépendants, lorsque l'on applique les algorithmes de seuillage SOJA-threshold, plus précisément les stratégies 4a et 4b.

Premier système

Les résultats présentés ici sont obtenus sur le système suivant :
- la séquence PN est produite par un registre de longueur $K = 40$, associé à un polynôme de rétroaction

$$g(x) = 1 + X + X^3 + X^5 + X^9 + X^{11} + X^{12} + X^{17} + X^{19} + X^{21} + X^{25} + X^{27} + X^{29} + X^{32} + X^{33} + X^{38} + X^{40};$$

- les équations de parité utilisées sont de poids $d = 5$; elles sont générées en utilisant l'algorithme de Canteaut-Trabbia [10]. Pour une longueur de séquence donnée, nous utilisons la totalité des équations générées par cet algorithme ;
- la fonction booléenne est la fonction f_{13}, à $n = 7$ entrées. Il s'agit d'une fonction plateau 3-résiliente, de paramètre $r = 4$; la probabilité de transition d'un canal binaire symétrique « équivalent » serait alors $p = \frac{1}{2} + \frac{\widehat{f}(\mathbf{u})}{2^7} = 0.375$;
- les espacements entre les entrées de la fonction sont premiers entre eux.

Le tableau 5.3 présente les résultats obtenus.

Stratégies de décodage	**N= 11000**	**N= 17000**
vrai BSC ($p = 0.375$) + Gallager	pas de convergence	85%
f_{13} vue comme un BSC + Gallager	pas de convergence	19%
f_{13} + (SOJA-2)-Gallager	pas de convergence	36%
f_{13} + (SOJA-1)-Gallager	pas de convergence	100%
f_{13} + (SOJA-2)-threshold	$\rho_{moy} = 0.226$	$\rho_{moy} = 0.02$
f_{13} + (SOJA-1)-threshold	$\rho_{moy} = 0.096$	$\rho_{moy} < 10^{-4}$

Tab. 5.3 – Comparaison de différentes stratégies de cryptanalyse incluant le SOJA. Paramètres du système testé : $K = 40$, équations de poids $d = 5$, fonction à $n = 7$ entrées, 3-résiliente, « plateau ».

Deuxième système

Nous nous intéressons ici au système suivant :
- la séquence PN est produite par un registre de longueur $K = 100$, associé au polynôme de rétroaction

$$g(x) = 1 + X^{37} + X^{100};$$

- les équations de parité utilisées sont de poids $d = 3$; elles sont générées par élévations au carré successives du polynôme de rétroaction comme suggéré dans [66]. Pour une longueur de séquence donnée, nous utilisons la totalité des équations générées par cet algorithme ;

- la fonction booléenne est la fonction f_{15} donnée en annexe. Il s'agit d'une fonction plateau à $n = 8$ entrées, 2-résiliente, de paramètre $r = 6$; la probabilité de transition d'un canal binaire symétrique « équivalent » serait alors $p = \frac{1}{2} + \frac{\widehat{f}(\mathbf{u})}{2^8} = 0.4375$;
- les espacements entre les entrées de la fonction sont premiers entre eux.

Le tableau 5.4 présente les résultats obtenus.

Stratégies de décodage	**N = 5000**	**N= 11000**	**N= 30000**
vrai BSC (p=0.4375) **+ Gallager**	pas de convergence	pas de convergence	pas de convergence
f_{15} **vue comme BSC + Gallager**	pas de convergence	pas de convergence	pas de convergence
f_{15} + **(SOJA-2)-Gallager**	pas de convergence	pas de convergence	pas de convergence
f_{15} + **(SOJA-1)-Gallager**	1%	42%	86%
f_{15} + **(SOJA-2)-threshold**	$\rho_{moy} = 0.245$	$\rho_{moy} = 0.165$	$\rho_{moy} = 0.08$
f_{15} + **(SOJA-1)-threshold**	$\rho_{moy} = 0.004$	$\rho_{moy} < 10^{-4}$	$\rho_{moy} < 10^{-4}$

Tab. 5.4 – Comparaison de différentes stratégies de cryptanalyse incluant le SOJA ; paramètres du système testé : $K = 100$, équations de poids $d = 3$, fonction à $n = 8$ entrées, 2-résiliente, « plateau ».

Quelques commentaires sur les différentes utilisations faites du décodage de Gallager

Nous avons présenté les résultats obtenus en appliquant la version δ de l'algorithme Belief Propagation à divers systèmes : tout d'abord, nous avons considéré un vrai canal binaire symétrique et initialisé l'algorithme à l'aide de sa probabilité de transition. Ensuite, nous avons filtré la séquence par une fonction booléenne, puis nous avons procédé comme s'il s'était agi d'un canal. Enfin, l'initialisation de l'algorithme de Gallager a été prise en sortie des algorithmes du SOJA-1 et SOJA-2.

Les résultats présentés dans le tableau 5.3 montrent que la convergence de l'algorithme de Gallager sur un canal BSC n'est pas identique à celle obtenue sur une fonction booléenne : le cas du BSC est plus favorable que celui de la fonction comme nous l'avions déjà observé au paragraphe 3.1.8 du chapitre 3. Notons toutefois que la fonction ici utilisée est une fonction résiliente plateau qui a précisément été choisie parce que sa structure handicape la convergence de l'algorithme de Gallager, comme nous l'avons expliqué au paragraphe 5.2.4 (voir page 166). La différence de comportement de l'algorithme est donc accrue par le choix d'une telle fonction.
Par ailleurs, soulignons que l'algorithme du SOJA-2 (version approchée du SOJA) présente de meilleures performances que l'algorithme BP : il converge dans 36% des cas, alors que l'algorithme BP appliqué à une fonction ne converge que dans 19% des cas.

Dans le deuxième exemple qui a été traité (voir tableau 5.4), il apparaît clairement que l'algorithme de Gallager classique ne permet pas de cryptanalyser le système. Nous avons même essayé des valeurs plus importantes de la longueur d'observation N (jusqu'à $N = 200000$ bits), sans observer de convergence. Rappelons que dans cet exemple, les équations de parité ont été générées

par élévations au carré successives du polynôme de rétroaction (cf. chapitre 2, paragraphe 2.2.1) et sont donc peu nombreuses, en quantité insuffisante pour permettre à l'algorithme BP de converger. L'utilité du SOJA est, dans ce cas, à souligner.

Enfin, nous constatons que la version approchée du SOJA, le SOJA-2, a des performances moins bonnes que celles obtenues avec le SOJA-1. Si, dans le premier exemple, une légère augmentation de la longueur de séquence N peut justifier l'utilisation du SOJA-2, cette version du SOJA ne peut concurrencer le SOJA-1 dans le deuxième exemple, surtout dans le cas du SOJA-Gallager.

Si les performances du SOJA-2 sont moins bonnes que celles du SOJA-1, la complexité qui lui est associée est nettement inférieure et on pourrait penser qu'il est préférable d'utiliser le SOJA-2 sur une longueur de séquence plus importante. Toutefois, sur les simulations que nous avons effectuées, nous avons constaté que le SOJA-2 donnait d'assez mauvais résultats en général, les meilleurs résultats ayant généralement été obtenus sur des fonctions résilientes. Une explication à ce phénomène pourrait être la suivante : le SOJA-2 est fondé sur une hypothèse, fausse, qui est que les bits d'un même vecteur sont indépendants. Cependant, l'hypothèse d'indépendance entre les entrées de la fonction est « moins fausse » lorsque la fonction est résiliente.
Il faut alors trouver un compromis entre les paramètres du système et la complexité de calculs que l'on s'autorise.

Efficacité du SOJA-threshold

Quel que soit l'exemple considéré, le SOJA-threshold apparaît comme l'algorithme le plus performant et le plus efficace. Dans certains cas, l'algorithme de Gallager ne permet pas du tout de cryptanalyser le système et le décodage à seuil est l'unique solution viable.

Le résultat le plus spectaculaire est obtenu dans le deuxième exemple (tableau 5.4) : le SOJA-threshold donne des résultats nettement meilleurs que les autres algorithmes, le nombre d'équations étant relativement faible. Sa complexité est également très réduite, puisqu'il s'agit d'opérer un simple décodage à seuil sur des bits « fiables » et indépendants, puis d'inverser un système linéaire ($K \times K$). Un test exhaustif sur les bits décodés apparaît de plus quasi-inutile puisque, d'après les résultats observés, une légère augmentation de la longueur de séquence permet de réduire le taux d'erreur de façon significative. De tels résultats s'expliquent par le fait que les équations de parité, bien que peu nombreuses, sont de poids faible, égal à $d = 3$. Comme nous l'avons vu au paragraphe 5.2.2 (voir page 159), ce cas est très favorable. Nous serons amenés à observer de nouveau cette propriété dans le paragraphe 5.4.

Exemples de temps de calcul

À titre indicatif, nous indiquons le temps nécessaire au calcul des probabilités sur les bits de la séquence (a), en utilisant soit le SOJA-1 soit le SOJA-2. Le fréquence du processeur utilisée était de 700 MHz, et nos programmes n'ont pas fait l'objet d'une optimisation particulière. Par exemple, dans le cas de la fonction f_{13} et d'équations de poids $d = 5$ (résultats présentés dans le tableau 5.3), les calculs pour une longueur de séquence de $N = 11000$ bits ont duré :

- 83 secondes dans le cas du SOJA-2 ;
- 104 secondes dans le cas du SOJA-1.

Remarquons que ces temps de calcul sont relativement courts (d'autant plus que nos programmes n'ont pas été optimisés). D'autre part, nous observons comme prévu que le SOJA-1 est plus complexe que le SOJA-2.

L'évaluation des probabilités par le SOJA-1 prenant une durée raisonnable, nous pouvons considérer une version itérée de ce calcul : le paragraphe qui suit présente un algorithme du SOJA itératif qui, à une itération donnée, réévalue les probabilités sur les vecteurs d'entrée en fonction des probabilités calculées à l'itération précédente.

5.3 Version itérée de l'algorithme du SOJA : le SOJA itératif

Dans cette partie, nous nous intéressons à une version itérée de l'algorithme du SOJA-1. Rappelons que le SOJA-1 consiste à évaluer des probabilités sur les vecteurs d'entrée dans un premier temps, d'en déduire des probabilités sur les bits ensuite.

L'algorithme du SOJA-1 se prolonge naturellement en un algorithme itératif : tout comme dans le décodage Belief Propagation (cf. chapitre 3), l'idée est d'utiliser les probabilités *a posteriori* obtenues sur les vecteurs d'entrée de la fonction comme probabilités *a priori* à l'itération suivante.

Nous allons dans un premier temps décrire l'algorithme du SOJA itératif et présenter un moyen efficace de calculer les probabilités. L'étude des performances montrera dans une deuxième partie, que si l'algorithme permet d'obtenir de bons résultats au cours des deux premières itérations, il diverge rapidement par la suite. Nous nous efforcerons donc en dernier lieu de donner une explication au comportement de cet algorithme.

5.3.1 Description de l'algorithme itératif

Notons $\Gamma^{(i),t}$ la quantité $\Gamma^t = P\Big(\mathbf{X}(t) = \mathbf{x} \mid z^E, f\Big)$ évaluée à l'itération i.

Nous supposons que la fonction booléenne f est équilibrée. L'algorithme est décrit dans le tableau 5.5.

Comment allons-nous procéder pour appliquer l'étape 2a de l'algorithme ci-dessus ?
Cette étape consiste à calculer la quantité

$$\Gamma^{(i),t} = P^{(i)}\Big(\mathbf{X}(t) = \mathbf{x} \mid z^E, f\Big)$$

en fonction des probabilités $P^{(i-1)}\Big(\mathbf{X}(t') = \mathbf{x} \mid z^{\mathcal{E}^v(t')}, f\Big)$ calculées à l'itération précédente.

Considérons un élément E de $\mathcal{E}^v(t)$:

$$(E) \ : \mathbf{X}(t) + \mathbf{X}(t+\theta_1) + \ldots + \mathbf{X}(t+\theta_{d-1}) = \mathbf{0}_n,$$

où l'indice m de l'équation est omis afin d'en simplifier l'écriture.

Algorithme SOJA-itératif

1. ***Initialisation :***
   $i = 0 : \forall t \in [0, N-1]$

$$\forall\ \mathbf{x},\ \ P^{(0)}\Big(\mathbf{X}(t) = \mathbf{x} \mid z,\ f\Big) = \frac{1}{2^{n-1}} \text{ si } f(\mathbf{x}) = z_t$$
$$= 0 \text{ sinon.}$$

2. ***Déroulement :***
   $i > 0$:
   (a) $\forall t,\ \forall E \in \mathcal{E}^v(t)$, calculer $\Gamma^{(i),t}$ en fonction de $P^{(i-1)}\left(\mathbf{X}(t) = \mathbf{x} \mid z^{\mathcal{E}^v(t)},\ f\right)$.
   (b) $\forall t,\ P^{(i)}\Big(\mathbf{X}(t) = \mathbf{x} \mid z^{\mathcal{E}^v(t)}, f\Big) = \frac{\prod_{E\in\mathcal{E}^v(t)} \Gamma^{(i),t}(\mathbf{x})}{\sum_{\mathbf{y}} \prod_{E\in\mathcal{E}^v(t)} \Gamma^{(i),t}(\mathbf{y})}$.
3. ***Terminaison :***
   $i = i_{max}$:
   (a) $\Xi_{\ell_{\mathbf{u}}}(t) = P\Big(\mathbf{u}.\mathbf{X}(t) = 1 \mid z, f\Big) = \sum_{x/\ell_{\mathbf{u}}(\mathbf{x})=1} \frac{\prod_{E\in\mathcal{E}^v(t)} \Gamma^{(i_{max}),t}(\mathbf{x})}{\sum_{\mathbf{y}} \prod_{E\in\mathcal{E}^v(t)} \Gamma^{(i_{max}),t}(\mathbf{y})}$.
   (b)

$$\Xi_{\ell_{\mathbf{u}}}(t) > 1/2 \implies a_t^{\ell_{\mathbf{u}}} = 1,$$
$$\Xi_{\ell_{\mathbf{u}}}(t) < 1/2 \implies a_t^{\ell_{\mathbf{u}}} = 0.$$

Tab. 5.5 – Algorithme du SOJA itératif.

La probabilité que $\mathbf{X}(t)$ soit égal à $\mathbf{x}$ dans cette équation est donnée par :

$$P\Big(\mathbf{X}(t) = \mathbf{x} \text{ dans } E\Big) = \sum_{\mathbf{y}_1} \dots \sum_{\mathbf{y}_{d-2}} P\Big(\mathbf{X}\left(t+\theta_1\right) = \mathbf{y}_1\Big) \times \dots \times P\Big(\mathbf{X}\left(t+\theta_{d-2}\right) = \mathbf{y}_{d-2}\Big)$$
$$\times P\Big(\mathbf{X}\left(t+\theta_{d-1}\right) = \mathbf{x} + \mathbf{y}_1 + \dots + \mathbf{y}_{d-2}\Big)\ . \tag{5.34}$$

Or, cette quantité est un produit de convolution. En utilisant la propriété donnée dans le chapitre 1 au paragraphe 1.4 (proposition 19 page 17) selon laquelle la transformée de Fourier d'un produit de convolution est un produit simple de transformées de Fourier, on obtient :
$\forall \mathbf{u} \in \{0,1\}^n$

$$\widehat{P}_{\mathbf{X}(t)}(\mathbf{u}) = \prod_{j=1}^{d-1} \widehat{P}_{\mathbf{X}(t+\theta_j)}(\mathbf{u})\ . \tag{5.35}$$

Le calcul des probabilités se fait donc de façon aisée dans le domaine fréquentiel. La figure 5.5 résume le calcul qui va être fait sur les probabilités.

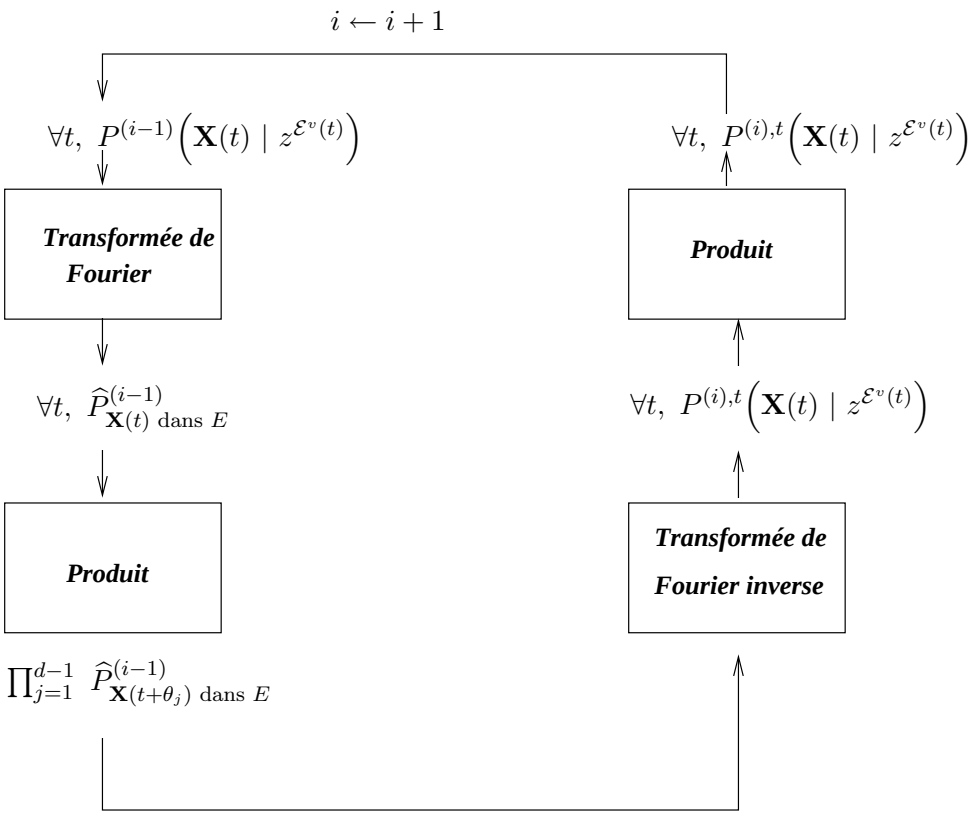


Fig. 5.5 – Différentes étapes du calcul des probabilités à l'itération i dans l'algorithme SOJA-itératif.

L'algorithme du SOJA-1 présenté dans le paragraphe précédent apparaît comme la première itération du SOJA-itératif : le SOJA-1 génère des informations en supposant que tous les vecteurs d'entrée sont équiprobables. Si la fonction est équilibrée, cela revient à initialiser l'algorithme itératif par

$$\begin{aligned} P\Big(\mathbf{X}(t) = \mathbf{x} \mid z_t\Big) &= \frac{1}{2^{n-1}} \text{ si } f\Big(\mathbf{X}(t)\Big) = z_t \\ P\Big(\mathbf{X}(t) = \mathbf{x} \mid z_t\Big) &= 0 \text{ sinon.} \end{aligned}$$

Les résultats obtenus après une itération de l'algorithme SOJA itératif sont strictement identiques à ceux qui seraient obtenus en appliquant le SOJA-1.

Finalement, nous constatons que cet algorithme est une généralisation de l'algorithme Belief Propagation décrit au chapitre 3 : reprenons la figure 3.4 du chapitre 3, page 70. Les noeuds carrés représentent une équation vectorielle, les noeuds ronds représentent les vecteurs d'entrée. Une branche relie une équation à un vecteur si ce dernier intervient dans l'équation. L'initialisation de l'algorithme est effectuée en fonction de la séquence reçue et du « canal » : la fonction booléenne. Dans le cas scalaire, l'initialisation était du type p ou $1-p$ puisque chaque bit ne pouvait prendre que deux valeurs. Dans le cas vectoriel, il nous faut calculer 2^n initialisations puisque les vecteurs d'entrée peuvent prendre 2^n valeurs. Cependant, l'initialisation est « plus forte», au sens où l'entrée et la sortie de la fonction étant liées de façon déterministe, un grand nombre de pro-

babilités sont initialisées à 0 (la moitié d'entre elles exactement lorsque la fonction est équilibrée).

Une question se pose alors naturellement : pourquoi ne pas avoir propagé de probabilités partielles sur les vecteurs comme dans la version α de l'algorithme Belief Propagation (voir paragraphe 3.1.2) ? La réponse est quasiment identique à la justification de l'application de l'algorithme BP version δ dans le chapitre 3 : d'une part, la complexité est plus importante lorsque l'on propage des informations partielles, d'autre part, nous avons simulé ce type de décodage et n'avons constaté aucune différence majeure avec l'algorithme approché décrit dans le cadre 5.5.

5.3.2 Résultats de simulation

Nous présentons ci-après les résultats obtenus en utilisant différentes fonctions. La procédure expérimentale fut la suivante : nous avons moyenné les résultats obtenus sur cent tirages, pour une longueur de séquence donnée. À chaque itération, nous avons évalué le taux d'erreur obtenu pour toutes les formes linéaires possibles de la séquence d'entrée et avons calculé le minimum.

La première fonction sur laquelle nous avons testé l'algorithme du SOJA-itératif est une fonction 1-résiliente à cinq entrées (fonction f_1). La seconde fonction considérée est une fonction à six entrées n'ayant pas de propriétés particulières, et dont la forme est donnée en annexe (fonction f_9). La dernière fonction testée est la fonction f_{15} à huit entrées, plateau et 2-résiliente, sur laquelle nous avons déjà testé les performances du SOJA précédemment (voir tableau 5.4).

Les taux d'erreurs minimaux sur toutes les itérations ont été obtenus pour la forme linéaire triviale, *i.e.* de poids 1, décalée sur toutes les positions. Nous reproduisons sur chacune des courbes 5.6, 5.7 et 5.8, le taux d'erreur obtenu en fonction du nombre d'itérations pour différentes longueurs de la séquence de clé disponible.

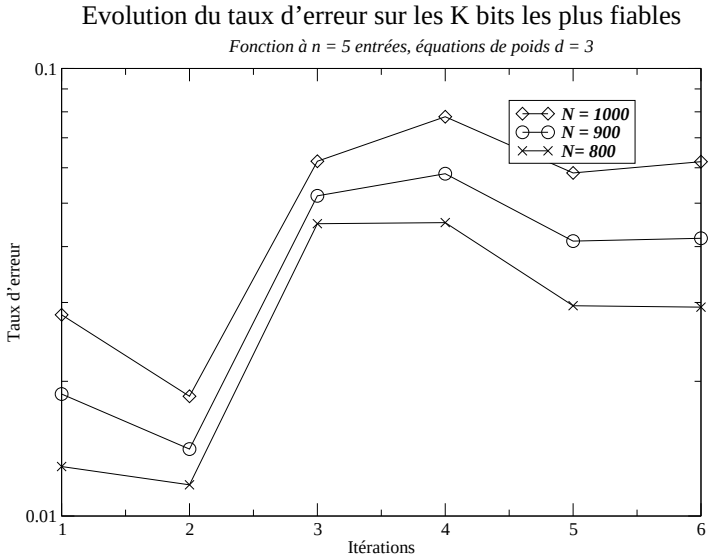


Fig. 5.6 – Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_1 en utilisant des équations de parité de poids $d = 3$.

Ces trois figures montrent nettement que l'algorithme du SOJA-itératif ne présente un intérêt

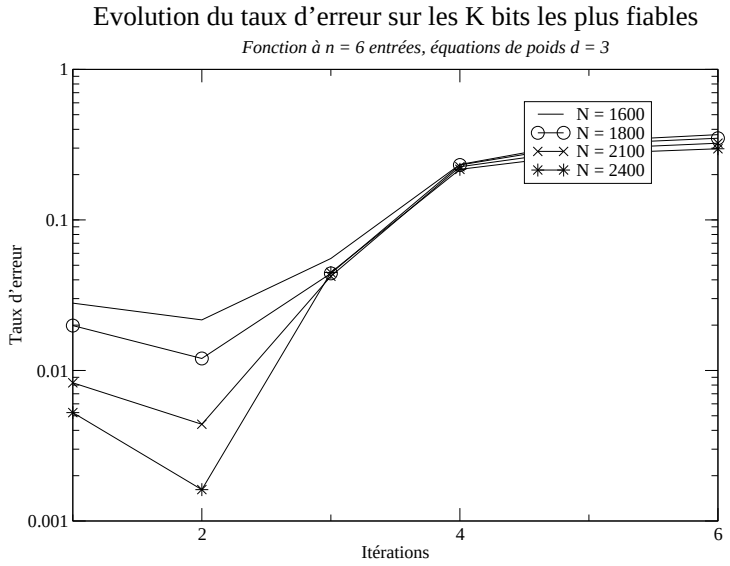


Fig. 5.7 – Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_9 en utilisant des équations de parité de poids $d = 3$.

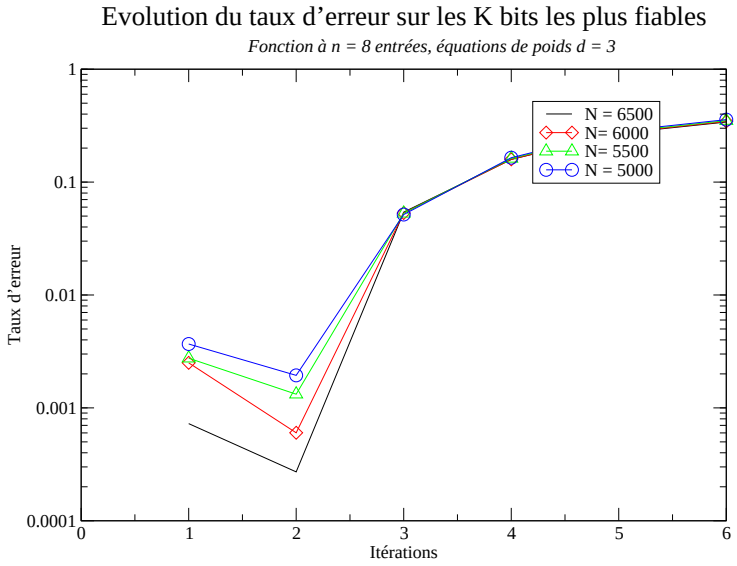


Fig. 5.8 – Taux d'erreur obtenu en appliquant le SOJA-itératif à la fonction f_{15} en utilisant des équations de parité de poids $d = 3$.

que sur les deux premières itérations : dès la troisième itération, il diverge et le taux d'erreur augmente. Nous suggérons une explication à ce phénomène dans le paragraphe qui suit.

5.3.3 Les cycles de commutativité

Les résultats présentés dans le paragraphe précédent sont, à première vue, inattendus : l'algorithme itératif ne permet pas d'améliorer grandement le taux d'erreur, seule une itération (en

plus du SOJA-1) est souvent utile. Nous avons cependant observé qu'une nette augmentation de la longueur d'observation permettait la convergence de l'algorithme. Comment peut-on expliquer un tel phénomène ?

L'explication qui nous semble la plus plausible est celle d'existence de cycles, dûs aux décalages effectués sur les vecteurs et à la commutativité de l'addition dans $\mathbb{N}$! Ces cycles ne sont pas sans rappeler ceux que nous avions déjà rencontrés dans l'algorithme IDBF (voir chapitre 3, page 97). Nous les appellerons « cycles de commutativité » par la suite.
Pour illustrer notre propos, considérons une équation de poids $d = 3$ écrite de la façon suivante :

$$(E) \ : \mathbf{X}(t) + \mathbf{X}(t + \theta_1) + \mathbf{X}(t + \theta_2) = \mathbf{0_n}.$$

Intéressons-nous à deux décalées particulières de cette équation, l'une obtenue en décalant E de θ_1, l'autre, en décalant E de θ_2. Le tableau ci-après figure les indices des vecteurs qui interviennent alors dans les trois équations :

$$\begin{array}{ccc} t & t+\theta_1 & t+\theta_2 \\ t+\theta_1 & t+2\theta_1 & t+\theta_1+\theta_2 \\ t+\theta_2 & t+\theta_1+\theta_2 & t+2\theta_2 \ . \end{array}$$

Lors de la première itération, $\mathbf{X}(t + \theta_1)$ reçoit de l'information des vecteurs $\mathbf{X}(t + 2\theta_1)$ et $\mathbf{X}(t + \theta_1 + \theta_2)$; le vecteur $\mathbf{X}(t + \theta_2)$ reçoit quant à lui de l'information des vecteurs $\mathbf{X}(t + \theta_1 + \theta_2)$ et $\mathbf{X}(t + 2\theta_2)$.
Dès la seconde itération, un cycle apparaît : $\mathbf{X}(t + \theta_1)$ et $\mathbf{X}(t + \theta_2)$ transmettent l'information générée précédemment à $\mathbf{X}(t)$, lequel reçoit en double l'information du vecteur $\mathbf{X}(t + \theta_1 + \theta_2)$.
La figure 5.9 illustre l'apparition des cycles et permet de comprendre l'allure générale des courbes de taux d'erreur observées. Cette figure présente le graphe de dépendance de la variable $\mathbf{X}(t)$ au cours des différentes itérations. Comme nous le constatons sur cette figure, le premier cycle apparaît à la deuxième itération, et la situation ne fait qu'empirer par la suite : à l'itération (i), $(i - 1)$ cycles apparaissent sur des nouveaux noeuds vecteurs ; seuls deux nouveaux vecteurs ne sont pas inclus dans un cycle. Ainsi, dès la troisième itération, la proportion de vecteurs inclus dans des cycles est supérieure à celle de vecteurs « seuls », et l'effet des cycles est décuplé au fur et à mesure des itérations.
Tout vecteur $\mathbf{X}(t)$ a donc une arborescence qui contient un « arbre parasite » rempli de cycles qui freinent la convergence, et ce, à partir de la deuxième itération essentiellement.
En revanche, lorsque l'on augmente la quantité d'observation, on augmente en même temps le nombre d'équations, et l'algorithme converge : l'influence de chaque arbre parasite est alors moindre devant le nombre d'équations.

5.4 Algorithme «Probability-matching» : un algorithme à mi-chemin entre le SOJA et la cryptanalyse linéaire

Dans cette partie, nous présentons un nouvel algorithme de cryptanalyse du registre filtré : s'il utilise les mêmes hypothèses que celles faites dans le SOJA, il se rapproche dans l'esprit de la cryptanalyse linéaire. Cet algorithme a donné lieu à la rédaction d'un article (cf. [58]).

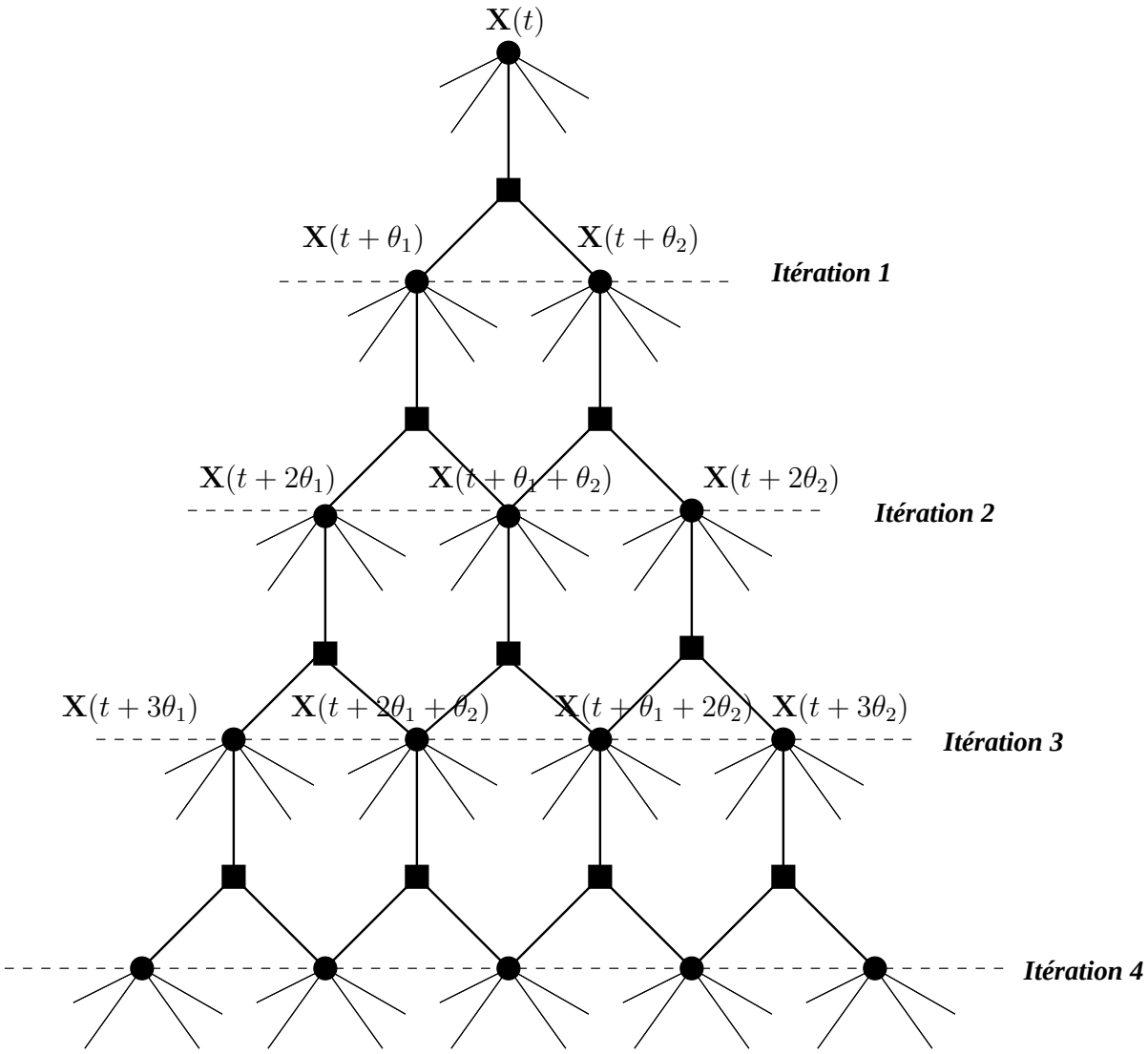


Fig. 5.9 – Cycles de commutativité dans le graphe des vecteurs d'entrée lors de l'application du SOJA-itératif.

5.4.1 Première approche de l'algorithme ; les équations de poids $d = 3$: un cas extrêmement favorable

Reprenons la remarque faite dans le paragraphe 5.2.2, page 159, et considérons par exemple des équations de parité de poids $d = 3$:

$$\mathbf{X}(t) + \mathbf{X}(t + \theta_{m,1}) + \mathbf{X}(t + \theta_{m,2}) = \mathbf{0}_n \ .$$

– Supposons que $\mathbf{X}(t) = \mathbf{0}_n$ alors

$$\forall m \in [1, \mu_v], \quad \mathbf{X}(t + \theta_{m,1}) = \mathbf{X}(t + \theta_{m,2}) \ ,$$

ce qui après filtrage par la fonction donne l'égalité :

$$\forall m \in [1, \mu_v], \quad z_{t+\theta_{m,1}} = z_{t+\theta_{m,2}} \ .$$

Une telle égalité entre les observations est extrêmement rare si le vecteur $\mathbf{X}(t)$ n'est pas le vecteur nul et si la fonction est quelconque : en considérant que la sortie de cette fonction est une variable aléatoire qui prend les valeurs 0 et 1 avec une probabilité égale à 1/2, nous pouvons écrire :

$$\forall m \in [1, \mu_v] \quad P\left(z_{t+\theta_{m,1}} = z_{t+\theta_{m,2}}\right) = \frac{1}{2^{\mu_v}} \ ;$$

– supposons maintenant que nous cherchions à cryptanalyser ce système ; d'après la remarque qui précède, si nous observons constamment que les bits de sortie sont égaux entre eux, *i.e.* :

$$\forall m \in [1, \mu_v], \ z_{t+\theta_{m,1}} = z_{t+\theta_{m,2}} \ , \tag{5.36}$$

nous pouvons en conclure que le vecteur $\mathbf{X}(t)$ est nul avec une probabilité d'erreur qui décroît avec la valeur de μ_v. Sous de telles hypothèses, un algorithme de cryptanalyse immédiat consiste à repérer les vecteurs tels que la propriété décrite dans l'équation (5.36) soit satisfaite. Une fois que l'on a réussi à décoder suffisamment de vecteurs, la simple inversion linéaire d'un système $(K \times K)$ nous procure l'état initial du générateur. Notons que dans ce cas précis, un test exhaustif sera nécessaire : les vecteurs d'entrée que l'on identifie étant tous nuls, le système obtenu est dégénéré, non inversible.

À première vue, un tel algorithme est peu « portable», puisqu'il est intrinsèquement lié au poids des équations de parité : dès que le poids des équations n'est plus égal à $d = 3$, il nous sera impossible de détecter le vecteur tout à zéro et l'application de l'algorithme paraît alors compromise.
Nous allons cependant montrer dans la suite, qu'en faisant de légères modifications dans l'algorithme pré-cité, nous allons pouvoir étendre son champ d'application à des équations de poids $d > 3$.

5.4.2 Principe de l'algorithme « Probability-matching »

Détaillons davantage l'attaque et pour cela, considérons une équation vectorielle :

$$(E) \quad : \ \mathbf{X}(t) + \mathbf{X}(t + \theta_{m,1}) + \ldots + \mathbf{X}(t + \theta_{m,d-1}) = \mathbf{0_n} \ .$$

Rappelons qu'après filtrage par la fonction, les vecteurs $\mathbf{X}(t)$, $\mathbf{X}(t + \theta_{m,1})$, …, $\mathbf{X}(t + \theta_{m,d-1})$, correspondent aux bits z_t, $z_{t+\theta_{m,1}}$, …, $z_{t+\theta_{m,d-1}}$, observés par le cryptanalyste. Comme nous

l'avons vu dans l'exemple donné dans le paragraphe 5.4.1 qui précède, certaines valeurs du vecteur d'entrée $\mathbf{X}(t)$ peuvent influencer la valeur du $(d-1)$-uplet $(z_{t+\theta_{m,1}}, \ldots, z_{t+\theta_{m,d-1}})$; rappelons également que tout vecteur d'entrée détermine la valeur du bit en sortie, z_t.

L'algorithme consiste alors à détecter ces vecteurs en fonction des valeurs prises par l'ensemble des $(z_{t+\theta_{m,1}}, \ldots, z_{t+\theta_{m,d-1}})$, et en fonction de z_t. L'influence théorique de la valeur du vecteur $\mathbf{X}(t)$ peut être évaluée à partir de la table des valeurs de la fonction. Une valeur expérimentale sera, quant à elle, calculée à partir des μ_v d-uplets dont nous disposons en sortie de la fonction pour chaque vecteur $\mathbf{X}(t)$: $\forall m \in [1, \mu_v]$, $(z_t, z_{t+\theta_{m,1}}, \ldots, z_{t+\theta_{m,d-1}})$. Nous allons alors constater que :

- certaines valeurs $\mathbf{x}$ du vecteur $\mathbf{X}(t)$ vont effectivement influencer le d-uplet

$$(z_t, z_{t+\theta_{m,1}}, \ldots, z_{t+\theta_{m,d-1}}) ,$$

  mais par l'intermédiaire de **la valeur de z_t et de la valeur de la somme $\sum_{i=1}^{d-1} z_{t+\theta_{m,i}}$ uniquement** (résultat qui avait déjà été observé lors du calcul des probabilités associées aux deux versions du SOJA) ;
- la plupart du temps et contrairement à ce qui a été présenté dans l'exemple, l'étude de ces $(d-1)$-uplets ne permettra pas de déterminer de façon unique le vecteur d'entrée. En revanche, il permettra d'identifier non pas un vecteur mais un ensemble de vecteurs, sur lesquels nous pourrons espérer trouver des formes linéaires constantes. Ces formes linéaires sur les vecteurs d'entrée sont « convertibles » en combinaisons linéaires des K premiers bits de la séquence (a). Ainsi, si l'on réussit à trouver K formes linéaires indépendantes, on pourra retrouver la clé secrète en inversant un système linéaire $(K \times K)$.

Dans la suite, nous avons choisi de présenter en premier lieu la partie de calculs des probabilités, puisque les résultats obtenus influencent la façon de procéder dans l'algorithme. Nous illustrerons le principe général de l'algorithme à l'aide d'un exemple simple. Enfin, nous donnerons sa forme générale.

5.4.3 Calcul des probabilités associées au « Probability-matching »

Dans cette partie, nous détaillons le calcul des probabilités sur lesquelles l'algorithme repose. Pour des raisons de simplicité, nous écrivons une équation vectorielle impliquant le vecteur $\mathbf{X}(t)$ sous la forme suivante :

$$\mathbf{X} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n} .$$

$(\mathbf{X}, \mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}})$ est un d-uplet de vecteurs liés par une équation vectorielle. Du point de vue du cryptanalyste, ces vecteurs sont des variables aléatoires uniformément distribuées. Nous introduisons donc les notations suivantes : $Z_0 = f(\mathbf{X})$ et $\forall i \geq 1$, $Z_i = f(\mathbf{Y^i})$.
$\forall\ \mathbf{x} \in \{0,1\}^n$, $\forall\ \mathbf{z} = (z_0, \ldots, z_{d-1})$, nous voulons évaluer la quantité suivante :

$$\Delta_{\mathbf{z}}(\mathbf{x}) = P\left[(Z_0, \ldots, Z_{d-1}) = (z_0, \ldots, z_{d-1}) \mid \mathbf{X} = \mathbf{x},\ f(\mathbf{X}),\ \mathbf{X} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n}\right] \quad (5.37)$$

De façon assez surprenante et comme nous l'avions déjà observé dans le cas du SOJA (voir la remarque 1 page 157), si la quantité $\Delta_{\mathbf{z}}$ est liée au $(d-1)$-uplet $(z_1, \ldots, z_{d-1})$, elle ne l'est que par la parité de la somme $\sum_{i=1}^{d-1} z_i$. La proposition ci-dessous illustre cette propriété en donnant l'expression de $\Delta_{\mathbf{z}}$.

Proposition 27 *Soit $f : \{0,1\}^n \longrightarrow \{0,1\}$ une fonction booléenne équilibrée. Alors, quelle que soit la valeur de $\mathbf{x} \in \{0,1\}^n$ et $\mathbf{z} = (z_0, z_1, \ldots, z_{d-1})$, la dépendance de $\Delta_{\mathbf{z}}$ par rapport à*

$(z_1, \ldots, z_{d-1})$ se résume à la valeur de la parité de $\sum_{i=1}^{d-1} z_i$. Plus précisément, $\Delta_{\mathbf{z}}$ s'exprime de la façon suivante :

$$\Delta_{\mathbf{z}}(\mathbf{x}) = \frac{\mathbb{1}_{\{f(\mathbf{x})=z_0\}}(\mathbf{x})}{2^{(n+1)(d-1)}} \left[\, 2^{(d-1)n} + (-1)^{z_1+\ldots+z_{d-1}} \, \widehat{b}(\mathbf{x}) \, \right] \tag{5.38}$$

où la probabilité $\Delta_{\mathbf{z}}$ a été définie dans l'équation (5.37), $b = (\widehat{f_\chi})^{d-1}$ et $f_\chi = (-1)^f$.

Preuve :
Reprenons la quantité précédemment introduite dans l'équation (5.12) (voir page 155)

$$\gamma_{\mathbf{z}}(\mathbf{x}) = \text{card} \left\{ \, (\mathbf{Y^1}, \ldots, \mathbf{Y^{d-1}}) \mid \quad f(\mathbf{x}) \; = z_0, \; f(\mathbf{Y^1}) = z_1, \ldots, \; f(\mathbf{Y^{d-1}}) = z_{d-1} \right.$$
$$\left. \text{et} \;\; \mathbf{x} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n} \right\} .$$

$\gamma_{\mathbf{z}}$ et $\Delta_{\mathbf{z}}$ sont liés par l'équation suivante :

$$\Delta_{\mathbf{z}}(\mathbf{x}) = \frac{\gamma_{\mathbf{z}}(\mathbf{x})}{\sum_{\mathbf{z}'} \gamma_{\mathbf{z}'}(\mathbf{x})} = \frac{\gamma_{\mathbf{z}}(\mathbf{x})}{2^{(d-2)n}} \; . \tag{5.39}$$

Dans le cas du SOJA, la quantité Γ était également liée à $\gamma_{\mathbf{z}}$ mais le coefficient de normalisation était différent :

$$\Gamma(\mathbf{x}) = \frac{\gamma_{\mathbf{z}}(\mathbf{x})}{\sum_{\mathbf{y}} \gamma_{\mathbf{z}}(\mathbf{y})} \; .$$

Le calcul de $\gamma_{\mathbf{z}}$ ayant été détaillé dans le paragraphe 5.2.2, nous ne revenons par sur ce calcul ; sous l'hypothèse que la fonction est équilibrée, nous avons obtenu l'expression suivante :

$$\begin{aligned} \gamma_{\mathbf{z}}(\mathbf{x}) \;\; &= \;\; \mathbb{1}_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \sum_{\mathbf{Y^1}} \cdots \sum_{\mathbf{Y^{d-2}}} \mathbb{1}_{\{f(\mathbf{Y^1})=z_1\}}(\mathbf{Y^1}) \ldots \mathbb{1}_{\{f(\mathbf{Y^{d-3}})=z_{d-3}\}}(\mathbf{Y^{d-3}}) \\ &\quad \times \; \mathbb{1}_{\{f(\mathbf{x}+\mathbf{Y^1}+\ldots+\mathbf{Y^{d-2}})=z_{d-1}\}}(\mathbf{Y^{d-2}}) \\ &= \;\; \mathbb{1}_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{1}{2^{d-1+n}} \left[\, 2^{(d-1)n} + (-1)^{z_1+\ldots+z_{d-1}} \, \widehat{b}(\mathbf{x}) \, \right] \end{aligned} \tag{5.40}$$

où $b = (\widehat{f_\chi})^{d-1}$ and $f_\chi = (-1)^f$. ∎

La proposition ci-dessus souligne le fait qu'il suffit d'évaluer $\Delta_{\mathbf{z}}$ pour des parités données de $\sum_{i>0} z_i$ (et pour une valeur de z_0 donnée). Les quantités qui nous intéressent sont donc dorénavant :

$$P\Big[\; Z_1 + \ldots + Z_{d-1} \;\; = \;\; 0 \pmod 2 \;\Big|\; \mathbf{X} = \mathbf{x}, \; f(\mathbf{X}), \; \mathbf{X} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n} \Big] \; , \tag{5.41}$$

$$P\Big[\; Z_1 + \ldots + Z_{d-1} \;\; = \;\; 1 \pmod 2 \;\Big|\; \mathbf{X} = \mathbf{x}, \; f(\mathbf{X}), \; \mathbf{X} + \mathbf{Y^1} + \ldots + \mathbf{Y^{d-1}} = \mathbf{0_n} \Big] \; , \tag{5.42}$$

que nous noterons respectivement $\Delta_{\text{even}}(\mathbf{x})$ et $\Delta_{\text{odd}}(\mathbf{x})$.
Par exemple, dans le cas des équations de poids $d = 3$ traité au paragraphe 5.4.1, la valeur d'intérêt était bien la valeur de la somme $z_{t+\theta_{m,1}} + z_{t+\theta_{m,2}}$ qui était nulle lorsque le vecteur d'entrée était le vecteur nul.

Corollaire 3 $\forall \mathbf{z}$, $\forall \mathbf{x}$, Δ_{even} *et* Δ_{odd} *s'expriment de la façon suivante :*

$$\begin{aligned} \Delta_{even}(\mathbf{x}) &= 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{1}{2}\left[1 + \frac{\widehat{b}(\mathbf{x})}{2^{n(d-1)}}\right], \\ \Delta_{odd}(\mathbf{x}) &= 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{1}{2}\left[1 - \frac{\widehat{b}(\mathbf{x})}{2^{n(d-1)}}\right], \end{aligned} \tag{5.43}$$

avec $b = (\widehat{f_\chi})^{d-1}$.

Preuve :
Remarquons tout d'abord que les quantités Δ_{even} et Δ_{odd} vérifient :

$$\forall \mathbf{x},\ \Delta_{\text{even}}(\mathbf{x}) + \Delta_{\text{odd}}(\mathbf{x}) = 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x})\ .$$

En utilisant l'expression donnée par l'équation (5.38), nous pouvons écrire :

$$\begin{aligned} \Delta_{\text{even}}(\mathbf{x}) &= \sum_{\mathbf{z}|z_1+\ldots+z_{d-1}=0 \mod 2} \Delta_{\mathbf{z}}(\mathbf{x}) \\ &= \frac{1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x})}{2^{(d-2)n}} \sum_{i=0}^{\lfloor \frac{d-1}{2} \rfloor} \binom{d-1}{2i} \times \frac{1}{2^{d-1+n}} \left[\ 2^{(d-1)n} + \widehat{b}(\mathbf{x})\ \right]\ . \end{aligned}$$

Comme

$$\sum_{i}^{\lfloor \frac{d-1}{2} \rfloor} \binom{d-1}{2i} = \sum_{i}^{\lceil \frac{d-1}{2} \rceil} \binom{d-1}{2i+1} = 2^{d-2},$$

nous obtenons

$$\begin{aligned} \Delta_{\text{even}}(\mathbf{x}) &= 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{1}{2}\left[1 + \frac{\widehat{b}(\mathbf{x})}{2^{n(d-1)}}\right] \\ \Delta_{\text{odd}}(\mathbf{x}) &= 1\!\!1_{\{f(\mathbf{x})=z_0\}}(\mathbf{x}) \times \frac{1}{2}\left[1 - \frac{\widehat{b}(\mathbf{x})}{2^{n(d-1)}}\right]\ . \end{aligned}$$

∎

Appliquons ces formules au cas traité dans le paragraphe 5.4.1, où, lorsque les équations étaient de poids $d = 3$, nous constations que si le vecteur d'entrée était nul, la somme des deux bits était toujours paire.
En effet, lorsque $d = 3$, $b = (\widehat{f_\chi})^2$ et

$$\begin{aligned} \widehat{b}(\mathbf{0_n}) &= \sum_{\mathbf{u}} (\widehat{f_\chi})^2(\mathbf{u}) \\ &= 2^{2n} \end{aligned}$$

en vertu de l'égalité de Parseval (cf. chapitre 1, équation (1.18) page 14).
On a alors

$$\begin{aligned}\Delta_{\text{even}}(\mathbf{0_n}) &= \mathbb{1}_{\{f(\mathbf{x})=z_0\}}(\mathbf{0_n}) \\ \Delta_{\text{odd}}(\mathbf{0_n}) &= 0\end{aligned}$$

Ainsi, il nous suffira de calculer les valeurs de $\Delta_{\text{even}}(\mathbf{x}), \quad \forall \mathbf{x} \in \{0,1\}^n$ et ranger les valeurs d'intérêt dans une table $\mathcal{T}$: les valeurs de Δ_{even} qui nous intéressent sont celles qui ont des formes linéaires constantes sur leur image réciproque.
Illustrons notre propos par un exemple concret.

Exemple 13 *Considérons la fonction booléenne f_1 à $n = 5$ entrées, 1-résiliente décrite en annexe.*
Supposons que nous travaillions avec des équations de parité de poids $d = 5$. La table 5.6 donne pour chaque valeur de Δ_{even} son image réciproque.

$z_0 = 0$		$z_0 = 1$	
valeur de Δ_{even}	Δ_{even}^{-1}	valeur de Δ_{even}	$\Delta_{\text{even}}^{-1}$
0.5	(00100), (11100) (00010), (11010) (11110), (10101) (01011), (11111)	0.437500	(11000), (10100) (01010), (00110) (01001), (00101) (11011), (10111)
0.546875	(01100), (10010)	0.5	(10000), (01000)
0.562500	(10001), (11101) (00011), (01111)		(10110), (01110) (11001), (01101)
0.578125	(00000), (00001)		(10011), (00111)

Tab. 5.6 – Construction de la table $\mathcal{T}$ de la fonction f_1 à $n = 5$ entrées, équations de poids $d = 5$.

Intéressons-nous par exemple à la partie droite de cette table, i.e., aux valeurs de Δ_{even} obtenues lorsque le bit reçu est un 1. Les vecteurs d'entrée de la fonction peuvent correspondre à deux valeurs de Δ_{even} : $\Delta_{even} = 0.4375$ ou $\Delta_{even} = 0.5$.
Observons les images réciproques de ces deux valeurs :

– *les vecteurs correspondant à $\Delta_{even} = 0.4375$ vérifient*

$$\begin{cases} x_2 + x_3 & = 1 \\ x_1 + x_4 + x_5 & = 1 \ ; \end{cases}$$

– *ceux qui correspondent à $\Delta_{even} = 0.5$ vérifient*

$$x_1 + x_2 + x_3 + x_4 + x_5 = 1 \ .$$

Ainsi, si l'on dispose de suffisamment d'équations vectorielles, i.e si μ_v est assez grand, on peut, pour chaque vecteur d'entrée $\mathbf{X}(t)$, évaluer les proportions de Δ_{even} et Δ_{odd} sur l'ensemble des équations vectorielles. Suivant la valeur de ces proportions, proches de 1/2 ou proches de 0.4375 ou de $1 - 0.4375 = 0.5625$, on pourra identifier l'ensemble auquel $\mathbf{X}(t)$ doit appartenir, et en déduire une ou deux formes linéaires constantes sur la séquence d'entrée.

Une fois que l'on aura réussi à identifier K formes linéaires indépendantes, on retrouvera l'état initial du générateur par une simple inversion linéaire.

Ce procédé n'est pas sans risque d'erreur : nous évaluons la probabilité d'erreur dans la section 5.4.5, mais nous pouvons d'ores et déjà prévoir que plus le nombre d'équations sera grand, plus la valeur de Δ_{even} calculée sur les vecteurs sera précise et plus le risque d'erreur sera faible.

5.4.4 Description complète de l'algorithme

Introduisons les valeurs dérivées de l'observation.

$$\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) = \frac{1}{\mu_v}\ \text{card}\Big\{m \in [1,\mu_v] \text{ tel que } \ z_{t+\theta_{m,1}} + \ldots + z_{t+\theta_{m,d-1}} = 0 \mod 2\Big\}$$

est la valeur de Δ_{even} calculée expérimentalement, *i.e.* sur les données dont nous disposons. L'exemple qui précède introduit l'algorithme du « Probability-matching », dérivé dans une version plus générale dans le tableau 5.7.

À la lecture de cet algorithme, le lecteur sera probablement amené à se poser les deux questions suivantes :

Q1. Comment allons-nous procéder pour appliquer la seconde étape de l'algorithme, *i.e.* vérifier que $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big)$ correspond à une valeur dans la table $\mathcal{T}$?

Il n'y a aucune réponse générale à cette question, puisque chaque fonction correspond à une table $\mathcal{T}$ bien précise, et pour chaque table, il existera une stratégie optimale permettant de minimiser la probabilité d'erreur. Dans le paragraphe 5.4.6, nous étudions plusieurs stratégies et les probabilités d'erreur correspondantes.

Q2. Dans quelle mesure notre algorithme est-il à rapprocher de la cryptanalyse linéaire [64] ?

La récupération de bits de clé (ou de relations linéaires sur ces derniers) est faite en identifiant certains vecteurs qui satisfont une relation de la forme :

$$(\mathcal{L})\ :\qquad \ell(a_i)_{i\in I} + \textstyle\sum_{j\in J_I} z_j = 0 \mod 2 \qquad (p_{\mathcal{L}})$$

$$\text{avec} \qquad \begin{array}{c} I \subset \{0,\ldots,K-1\} \\ J_I \subset \{0,\ldots,N-1\} \end{array}$$

avec une certaine probabilité, $p_{\mathcal{L}}$; ils sont identifiés grâce aux valeurs prises par $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big)$. Ces valeurs sont calculées sur le plus grand nombre d'équations possibles : plus on disposera d'équations, plus la probabilité d'erreur sera faible.

5.4.5 Complexité de l'algorithme et calcul de la probabilité d'erreur associée au « Probability-matching »

Complexité

Nous nous intéressons ici à la complexité de notre algorithme, sans inclure la complexité de la phase de calculs des équations de parité. La complexité de l'algorithme se décline alors en trois

Algorithme Probability-matching

1. ***Pré-calculs :***
   la table $\mathcal{T}$ contenant les valeurs de Δ_{even} ayant des formes linéaires constantes sur leur image réciproque $\forall t,\ \mathcal{E}_v(t)$.
2. ***Entrées :***
   $g(x)$, $(z_t)_0^{N-1}$, $\mathcal{T}$ et $\forall t,\ \mathcal{E}^v(t)$
3. ***Déroulement :***
   soit ν_{ind} le nombre de formes linéaires constantes de la séquence d'entrée obtenues.
   $\nu_{ind} = 0$.
   (a) Si $t < N$ et $\nu_{ind} < K$, évaluer

$$\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) = \frac{1}{\mu_v}\ \text{card}\Big\{m \in [1, \mu_v] \text{ tel que } z_{t+\theta_{m,1}} + \ldots + z_{t+\theta_{m,d-1}} = 0 \bmod 2\Big\}\ .$$

   (b) Vérifier si $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big)$ peut correspondre à une valeur de Δ_{even} dans la table $\mathcal{T}$.
   Stocker l'éventuelle forme linéaire correspondante.
   Si $\nu_{ind} < K$, incrémenter t et retourner à l'étape 3a. Sinon, aller à l'étape 4.
4. ***Terminaison :***
   si $\nu_{ind} \geq K$, inverser le système linéaire $(K \times K)$ afin de retrouver l'état initial du générateur PN.

Tab. 5.7 – Algorithme « Probability-matching ».

phases principales :

- la phase de **création de la table $\mathcal{T}$** qui requiert $\mathcal{O}(n2^n)$ opérations pour évaluer les valeurs de Δ_{even} ;
- la phase de **recherche des formes linéaires constantes** sur l'image réciproque de Δ_{even} a généralement une complexité négligeable, le nombre de vecteurs appartenant à l'image réciproque de chaque valeur étant restreint ;
- la complexité de la phase d'**évaluation des parités** $\widetilde{\Delta}_{\text{even}}$ est en $\mathcal{O}\left(\mu_v dN\right)$, où N est la longueur de séquence (z) nécessaire pour obtenir K formes linéaires indépendantes.

La complexité globale du noyau de l'algorithme est alors en :

$$\mathcal{O}\left(\mu_v dN + n2^n\right) \approx \mathcal{O}\left(\mu_v dN\right)\ . \tag{5.44}$$

Sans oublier la phase finale, qui est l'inversion d'un système linéaire ($K \times K$).

Remarques :

1. La complexité de l'algorithme du « Probability-matching » apparaît identique à celle qui fut donnée pour le SOJA-2 (cf. équation 5.32 page 165). Il nous faut toutefois souligner la différence qui réside dans les opérations effectuées : le SOJA-2 évalue des quantités réelles (des probabilités), tandis que le « Probability-matching » calcule des sommes discrètes. Si le nombre total d'opérations effectuées est du même ordre de grandeur, la nature de ces opérations est très différentes. Le « Probability-matching » sera donc plus rapide et moins complexe.
2. La structure de la fonction a un rôle déterminant puisqu'elle va définir le nombre de formes linéaires constantes pour chaque valeur de Δ_{even} : plus on pourra trouver de formes linéaires constantes, moins on aura besoin de bits de séquence (z). La table de la fonction influence donc la complexité de l'algorithme par l'intermédiaire de la longueur de séquence nécessaire au succès de ce dernier.

Calcul de la probabilité d'erreur

Dans ce paragraphe, nous nous intéressons à la probabilité de prendre une mauvaise décision, c'est-à-dire de **décider à tort qu'un vecteur d'entrée appartient à l'image réciproque d'une valeur donnée de Δ_{even}**. La probabilité d'erreur « réelle » qui serait celle de décider qu'une combinaison linéaire des bits de la séquence d'entrée est constante alors qu'elle ne l'est pas, est toujours inférieure à la probabilité que nous allons évaluer ci-après (cette probabilité d'erreur s'obtient en pondérant la probabilité que nous allons calculer par la probabilité que la combinaison linéaire soit vérifiée sur l'ensemble de vecteurs considéré).
L'hypothèse que nous faisons dans le calcul qui va suivre est la suivante : pour un bit reçu donné, par exemple un '1' (qui, de façon visuelle, définit le côté de la table $\mathcal{T}$ auquel on s'intéresse), il n'existe que deux valeurs possible de Δ_{even}, que nous noterons pour simplifier P_A et P_B. Ces deux valeurs ont pour image réciproque les ensembles $\mathcal{A}$ et $\mathcal{B}$ respectivement. Nous supposons que l'ensemble d'intérêt est $\mathcal{B}$, c'est-à-dire que $\mathcal{B}$ est le seul ensemble sur lequel on puisse trouver des formes linéaires constantes.

Cette hypothèse n'est absolument pas restrictive : comme nous allons le voir, le calcul de la probabilité d'erreur repose sur l'évaluation de cardinaux, ou de façon équivalente, de probabilités conditionnelles. Nous pourrons donc adapter les calculs développés ci-après à chaque stratégie.

On a

$$\begin{aligned} \mathcal{A} &= \Big\{ \mathbf{X} \in \{0,1\}^n \mid f(\mathbf{X}) = 1 \text{ et } \Delta_{\text{even}}(\mathbf{X}) = P_A \Big\} \\ \mathcal{B} &= \Big\{ \mathbf{X} \in \{0,1\}^n \mid f(\mathbf{X}) = 1 \text{ et } \Delta_{\text{even}}(\mathbf{X}) = P_B \Big\}, \end{aligned}$$

tels que $\text{card}(\mathcal{A}) + \text{card}(\mathcal{B}) = 2^{n-1}$ si la fonction est équilibrée.

De par la structure du générateur PN, les vecteurs en entrée de la fonction peuvent être considérés équidistribués, *i.e.* :

$$\forall\ \mathbf{x} \in \{0,1\}^n,\ P(\mathbf{X} = \mathbf{x}) = \frac{1}{2^n}.$$

La probabilité d'occurrence qu'un vecteur quelconque en entrée de la fonction appartienne à $\mathcal{A}$ ou à $\mathcal{B}$ est alors respectivement :

$$\begin{aligned} P(\mathbf{X} \in \mathcal{A}) &= \frac{\text{card}(\mathcal{A})}{2^n}, \\ P(\mathbf{X} \in \mathcal{B}) &= \frac{\text{card}(\mathcal{B})}{2^n}. \end{aligned}$$

Rappelons que $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big)$ est la valeur de Δ_{even} calculée expérimentalement et que l'on cherche à identifier à l'une des valeurs possibles de Δ_{even}. D'après les hypothèses faites sur le système plus haut, l'une des stratégies qui permet d'avoir un taux d'erreur minimal est de se concentrer sur les valeurs de $\widetilde{\Delta}_{\text{even}}$ qui sont inférieures à P_B. La probabilité d'erreur peut alors être calculée de la façon suivante :

$$P_{err} = \frac{\text{card}\Big\{\mathbf{X} \in \mathcal{A} / \widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B\Big\}}{\text{card}\Big\{\mathbf{X} \in \mathcal{A} / \widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B\Big\} + \text{card}\Big\{\mathbf{X} \in \mathcal{B} / \widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B\Big\}}. \tag{5.45}$$

La quantité décrite ci-dessus est équivalente à une écriture faisant intervenir des probabilités conditionnelles. Nous choisissons de conserver cette écriture car elle peut être directement reliée à la phase 3a de l'algorithme.

Le calcul de la probabilité d'erreur consiste à distinguer deux (ou plus) distributions binomiales. Nous détaillons ce calcul et les approximations faites ci-dessous.

– **Estimation du cardinal : card$\Big\{\mathbf{X} \in \mathcal{A} / \widetilde{\Delta}_{\textbf{even}}(\mathbf{X}) \leq P_B\Big\}$**

$$\begin{aligned} \text{card}\Big\{\mathbf{X} \in \mathcal{A} / \widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B\Big\} &= N \times P\Big(\mathbf{X} \in \mathcal{A} \text{ et } \widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B\Big) \\ &= N \times P(\mathbf{X} \in \mathcal{A}) \times P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B \mid \mathbf{X} \in \mathcal{A}\Big) \end{aligned}$$

en utilisant la définition de la probabilité conditionnelle.
$\widetilde{\Delta}_{\text{even}}$ évalué sur les vecteurs éléments de $\mathcal{A}$ suit une loi binomiale $B(\mu_v, P_A)$.
Ainsi,

$$\begin{aligned} P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B \mid \mathbf{X} \in \mathcal{A}\Big) &= \sum_{\lambda=0}^{\lfloor \mu_v P_B \rfloor} \binom{\mu_v}{\lambda} P_A^{\lambda} (1-P_A)^{\mu_v - \lambda} \\ &= \binom{\mu_v}{\lfloor \mu_v P_B \rfloor} P_A^{\lfloor \mu_v P_B \rfloor} (1-P_A)^{\mu_v - \lfloor \mu_v P_B \rfloor} \\ &\times \left[1 + \sum_{i=1}^{\lfloor \mu_v P_B \rfloor} \left(\frac{1-P_A}{P_A}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_B \rfloor - j + 1}{\mu_v - \lfloor \mu_v P_B \rfloor + j}\right]. \end{aligned}$$

En utilisant la formule de Stirling au second ordre, on peut écrire :

$$\begin{aligned} P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B \mid \mathbf{X} \in \mathcal{A}\Big) &\approx \frac{1}{\sqrt{2\pi \mu_v P_B (1-P_B)}} 2^{-\mu_v D_K(P_B || P_A)} \\ &\times \left[1 + \sum_{i=1}^{\lfloor \mu_v P_B \rfloor} \left(\frac{1-P_A}{P_A}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_B \rfloor - j + 1}{\mu_v - \lfloor \mu_v P_B \rfloor + j}\right], \end{aligned} \tag{5.46}$$

où D_K est la distance de Kullback entre les deux distributions de probabilité :

$$D_K(P_B||P_A) = P_B \log_2 \frac{P_B}{P_A} + (1-P_B)\log_2 \frac{1-P_B}{1-P_A} \ . \tag{5.47}$$

Nous avons tracé sur la figure 5.10 les variations de la distance de Kullback en fonction de P_B, en fixant $P_A = \frac{1}{2}$.

– **Estimation du cardinal : card$\Big\{\mathbf{X} \in \mathcal{B}/\widetilde{\Delta}_{\mathbf{even}} \leq P_B\Big\}$**

En procédant comme précédemment, on obtient :

$$\begin{aligned} P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_B \mid \mathbf{X} \in \mathcal{B}\Big) &= \frac{1}{\sqrt{2\pi\mu_v P_B(1-P_B)}} \\ &\times \left[1 + \sum_{i=1}^{\lfloor \mu_v P_B \rfloor} \left(\frac{1-P_B}{P_B}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_B \rfloor - j + 1}{\mu_v - \lfloor \mu_v P_B \rfloor + j}\right]. \end{aligned} \tag{5.48}$$

En utilisant les expressions (5.45), (5.46) et (5.48), on obtient l'expression finale de la probabilité d'erreur :

$$P_{err} = \frac{1}{1 + 2^{\mu_v D_K(P_B||P_A)} \frac{\text{card}(\mathcal{B})}{\text{card}(\mathcal{A})} \times \frac{1+\sum_{i=1}^{\lfloor \mu_v P_B \rfloor}\left(\frac{1-P_B}{P_B}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_B \rfloor - j+1}{\mu_v - \lfloor \mu_v P_B \rfloor + j}}{1+\sum_{i=1}^{\lfloor \mu_v P_B \rfloor}\left(\frac{1-P_A}{P_A}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_B \rfloor - j+1}{\mu_v - \lfloor \mu_v P_B \rfloor + j}}} \ . \tag{5.49}$$

Remarques :

• La formule (5.49) est identique à celle que nous aurions obtenue en utilisant l'approximation de Stirling au premier ordre. Cela est dû au fait que nous calculons les rapports de cardinaux d'ensembles dans l'équation (5.45). Cependant, le calcul exact des cardinaux, utile si l'on souhaite évaluer le nombre de formes linéaires constantes disponibles, nécessite l'utilisation de la formule au second ordre.
• La formule de la probabilité d'erreur que nous venons de calculer est ici spécifique à la stratégie de décodage que nous avons choisi de considérer. Néanmoins, cette stratégie apparaît comme « une brique de base » de nombre de stratégies de décodage envisageables. Le calcul de la probabilité d'erreur développé ci-dessus est aisément généralisable : il suffira, en fonction des ensembles considérés, de calculer les cardinaux des ensembles pertinents.
Nous détaillons d'autres calculs de probabilités d'erreur fondés sur ce principe dans le paragraphe suivant.
Afin de simplifier leurs écritures, nous introduisons d'ores et déjà deux quantités supplémentaires.

Soit $\mathcal{U}$ un ensemble de vecteurs associé par Δ_{even} à la probabilité P_U :

$$\forall \mathbf{X} \ \in \mathcal{U},\ E\left[\widetilde{\Delta}_{\text{even}}(\mathbf{X})\right] = P_U.$$

Soit $\mathcal{S}$ un ensemble de vecteurs quelconques.
Alors, on pose :

$$P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_U \mid \mathbf{X} \in \mathcal{S}\Big) = P_{inf}(\mathcal{U},\mathcal{S}), \tag{5.50}$$

$$P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \geq P_U \mid \mathbf{X} \in \mathcal{S}\Big) = P_{sup}(\mathcal{U},\mathcal{S}). \tag{5.51}$$

D'après la formule (5.46), nous pouvons écrire :

$$\begin{aligned} P_{inf}(\mathcal{U},\mathcal{S}) &= P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \leq P_U \mid \mathbf{X} \in \mathcal{S}\Big) \\ &\approx \frac{1}{\sqrt{2\pi\mu_v P_U(1-P_U)}} 2^{-\mu_v D_K(P_U||P_S)} \\ &\times \left[1 + \sum_{i=1}^{\lfloor \mu_v P_U \rfloor} \left(\frac{1-P_S}{P_S}\right)^i \prod_{j=1}^{i} \frac{\lfloor \mu_v P_U \rfloor - j + 1}{\mu_v - \lfloor \mu_v P_U \rfloor + j}\right], \end{aligned} \tag{5.52}$$

et

$$\begin{aligned} P_{sup}(\mathcal{U},\mathcal{S}) &= P\Big(\widetilde{\Delta}_{\text{even}}(\mathbf{X}) \geq P_U \mid \mathbf{X} \in \mathcal{S}\Big) \\ &\approx \frac{1}{\sqrt{2\pi\mu_v P_U(1-P_U)}} 2^{-\mu_v D_K(P_U||P_S)} \\ &\times \left[1 + \sum_{i=1}^{\mu_v - \lceil \mu_v P_U \rceil} \left(\frac{1-P_S}{P_S}\right)^i \prod_{j=1}^{i} \frac{\mu_v - \lceil \mu_v P_U \rceil - j + 1}{\lceil \mu_v P_U \rceil + j}\right], \end{aligned} \tag{5.53}$$

$$\text{avec } P_{inf}(\mathcal{U},\mathcal{S}) + P_{sup}(\mathcal{U},\mathcal{S}) = 1. \tag{5.54}$$

Nous avons vu que ces quantités étaient exponentiellement proportionnelles à la distance de Kullback :

$$P_{inf}(\mathcal{U},\mathcal{S}) \propto 2^{-\mu_v D_K(P_U||P_S)}, \tag{5.55}$$

$$P_{sup}(\mathcal{U},\mathcal{S}) \propto 2^{-\mu_v D_K(P_U||P_S)}. \tag{5.56}$$

En particulier, nous pouvons réécrire la probabilité d'erreur de la formule (5.49) sous la forme :

$$P_{err} = \frac{1}{1 + \frac{\text{card}\mathcal{B}}{\text{card}\mathcal{A}} \times \frac{P_{inf}(\mathcal{B},\mathcal{B})}{P_{inf}(\mathcal{A},\mathcal{B})}} \ . \tag{5.57}$$

5.4.6 Résultats de simulation - Comparaison des résultats expérimentaux à la théorie

Nous présentons en premier lieu les résultats que nous avons obtenus en utilisant des équations de parité de poids $d = 5$. Ces équations ont été obtenues en utilisant l'algorithme Canteaut-Trabbia développé dans [10] (cf. chapitre 2, paragraphe 2.2.1) sur le polynôme suivant :

$$\begin{aligned} g(x) &= 1 + x + x^3 + x^5 + x^9 + x^{11} + x^{12} + x^{17} + x^{19} + x^{21} + x^{25} + x^{27} + x^{29} \\ &\quad + x^{32} + x^{33} + x^{38} + x^{40} \ . \end{aligned}$$

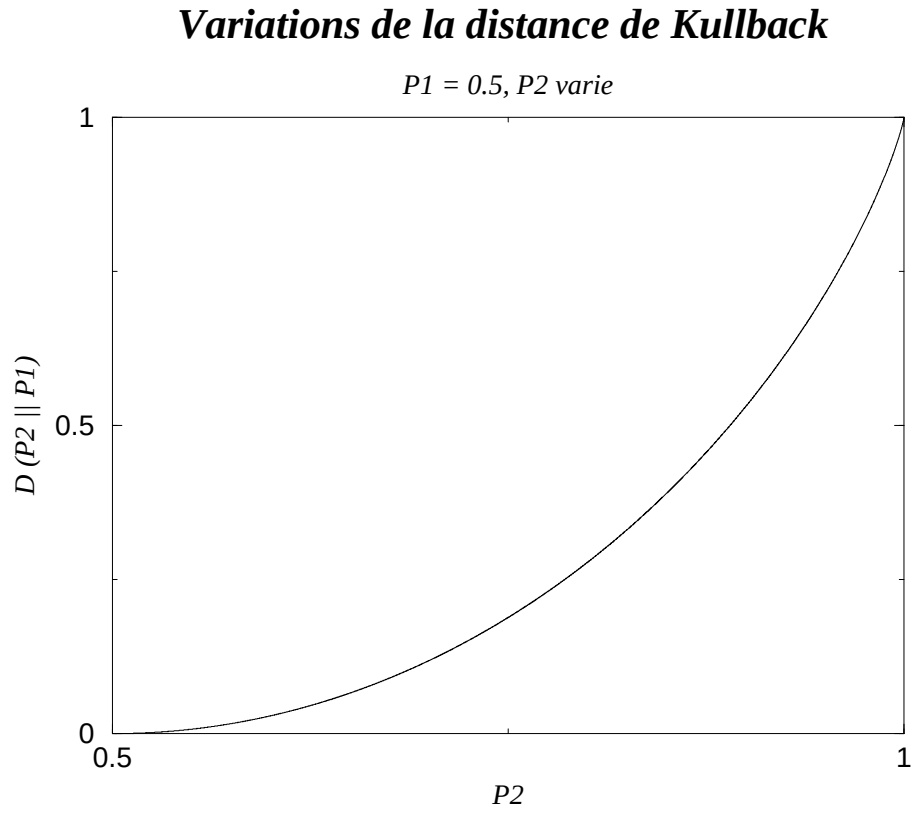


Fig. 5.10 – Variations de la distance de Kullback D_K entre une loi binomiale d'espérance $P_1 = \frac{1}{2}$ et une loi binomiale dont l'espérance P_2 varie.

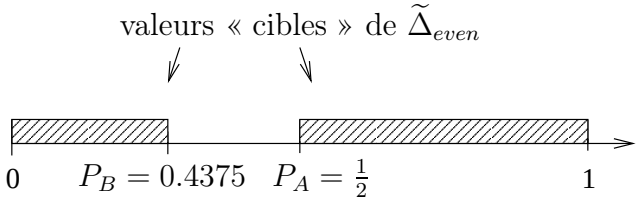


Fig. 5.11 – Illustration de la première stratégie de cryptanalyse de la fonction f_1 en utilisant l'algorithme du « Probability-matching ».

Résultats obtenus sur une fonction à $n = 5$ entrées, 1-résiliente, f_1

Cette fonction a été étudiée dans le paragraphe 5.4.3, dans lequel nous avons également donné la table $\mathcal{T}$.
Nous proposons ci-après deux stratégies :

- ***première stratégie :***
  cette stratégie est la plus simple. Comme nous l'avons vu dans l'exemple donné au paragraphe 5.4.3, lorsque le bit reçu en sortie de la fonction est égal à '1', huit vecteurs d'entrée génèrent une « even-parity » égale à 0.4375, les huit vecteurs restant ont une « even-parity » égale à 0.5. Ces deux ensembles de vecteurs vérifient la propriété d'avoir des formes linéaires constantes, et nous proposons donc la stratégie suivante : en posant $P_B = 0.4375$, $P_A = 0.5$ lorsque le bit reçu est un '1', nous nous intéressons aux vecteurs $\mathbf{X}(t)$ qui vérifient :
  $$\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) < P_B \text{ ou } \widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) > P_A \ .$$
  Nous illustrons cette stratégie sur la figure 5.11 ;
- ***deuxième stratégie :***
  la deuxième stratégie que nous proposons, et qui donne de bien meilleurs résultats, consiste à s'intéresser dorénavant à la partie gauche de la table $\mathcal{T}$ de la fonction, *i.e.* à l'image réciproque de '0' par Δ_{even}.
  Cependant, nous allons regrouper les vecteurs appartenant à l'image réciproque de $\Delta_{\text{even}} = 0.578125$, $\Delta_{\text{even}} = 0.5625$ et $\Delta_{\text{even}} = 0.546875$ en un seul ensemble, et chercher des formes linéaires constantes sur la réunion de ces trois ensembles. Pourquoi opérer une telle réunion ? Les valeurs de Δ_{even} que nous venons de citer sont très proches les unes des autres. D'après l'équation (5.49), la probabilité d'erreur est inversement proportionnelle à la distance de Kullback entre les probabilités des ensembles considérés, $D_K(P_B||P_A)$: plus cette distance est grande, plus la probabilité d'erreur est faible. Or l'allure de $D_K(P_B||P_A)$ est donnée sur la figure 5.10 : il s'agit d'une fonction croissante de l'écart entre les probabilités.
  Pour la même raison que celle que nous venons d'évoquer, il apparaît nettement que cette seconde stratégie est meilleure que la première : la distance entre $\frac{1}{2}$ et 0.578125 est supérieure à celle qui existe entre $\frac{1}{2}$ et 0.4375.
  Notons toutefois que le nombre de formes linéaires constantes pour une longueur de séquence donnée N est moins important si l'on utilise cette seconde stratégie : contrairement au cas précédent, lorsque le bit reçu est un '0', $\Delta^{-1}_{\text{even}}(0.5)$ ne présente pas de forme linéaire constante. Il faudra donc observer davantage de bits en sortie de la fonction pour obtenir K combinaisons linéaires des premiers bits indépendantes.

  La seconde stratégie sera la suivante : si le bit reçu est un '0', nous évaluons la valeur de $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big)$ et ne conservons que les vecteurs $\mathbf{X}(t)$ qui satisfont :
  $$\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) \geq 0.578125.$$
  Alors ce vecteur est attribué à
  $$\Delta^{-1}_{\text{even}}(0.578125) \ \cup \ \Delta^{-1}_{\text{even}}(0.5625) \ \cup \ \Delta^{-1}_{\text{even}}(0.546875) \ .$$

Détaillons ci-après les probabilités d'erreur associées à ces deux stratégies :

• Probabilité d'erreur associée à la première stratégie

Cette stratégie consiste à ne garder que les vecteurs $\mathbf{X}(t)$ qui vérifient $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) < P_B$ ou $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) > P_A$.
Une erreur sera donc commise dès lors qu'un vecteur vérifiera « à tort » l'une des deux conditions ci-dessus. Le nombre d'erreurs commises est donné par :

$$\text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \leq P_B\Big\} + \text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\} \ .$$

Le nombre total de vecteurs vérifiant $\widetilde{\Delta}_{\text{even}} < P_B$ ou $\widetilde{\Delta}_{\text{even}} > P_A$ est donné par :

$$\begin{aligned} D &= \text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \leq P_B\Big\} \ + \ \text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\} \\ &\quad + \text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\} \ + \ \text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \leq P_B\Big\} \\ &= \text{card}\Big\{\mathbf{X} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\} + \text{card}\Big\{\mathbf{X} \mid \widetilde{\Delta}(\mathbf{X}) \leq P_B\Big\} \ . \end{aligned}$$

En appliquant le même type de calculs que ceux développés dans la partie 5.4.5, nous obtenons :

$$\text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\} = N \times \frac{\text{card}(\mathcal{B})}{2^n} \times P_{sup}(\mathcal{A}, \mathcal{B})$$

et

$$\text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\} = N \times \frac{\text{card}(\mathcal{B})}{2^n} \times P_{sup}(\mathcal{A}, \mathcal{A}) \ .$$

Or $\text{card}(\mathcal{A}) = \text{card}(\mathcal{B})$, et donc

$$P_{err}^{\text{stratégie 1}} = \frac{\text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \leq P_B\Big\} + \text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_A\Big\}}{D} \tag{5.58}$$

$$= \frac{1}{1 + \frac{P_{sup}(\mathcal{A},\mathcal{A}) + P_{inf}(\mathcal{B},\mathcal{B})}{P_{inf}(\mathcal{A},\mathcal{B}) + P_{sup}(\mathcal{A},\mathcal{B})}} \ . \tag{5.59}$$

Or le terme $\frac{P_{sup}(\mathcal{A},\mathcal{A}) + P_{inf}(\mathcal{B},\mathcal{B})}{P_{inf}(\mathcal{A},\mathcal{B}) + P_{sup}(\mathcal{A},\mathcal{B})}$ est proportionnel à $2^{\mu_v D_K(P_B,P_A)}$: $P_A = \frac{1}{2}$, $P_B = 0.4375$, ce qui correspond à une valeur de la distance de Kullback : $D_K(P_A, P_B) = 0.0113$.

Ce calcul explique les résultats observés sur la figure 5.12.

• Probabilité d'erreur associée à la seconde stratégie :

Posons

$$\begin{aligned} P_A &= \frac{1}{2} \\ P_B &= 0.571825 \\ P_C &= 0.5625 \\ P_D &= 0.546875 \end{aligned}$$

et soient $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}$ les images réciproques respectives de ces probabilités par Δ.

La seconde stratégie consiste à dénombrer les vecteurs $\mathbf{X}(t)$ qui appartiennent à $\mathcal{B} \cup \mathcal{C} \cup \mathcal{D}$ et vérifient $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) \geq P_B$. Le choix de cette stratégie est directement dicté par le fait que P_B est la valeur de la probabilité la plus « éloignée » de $\frac{1}{2}$. Nous aurions pu choisir d'autres valeurs de probabilité cibles, la moyenne de P_B, P_C et P_D pondérées par le cardinal de chacun des ensembles correspondant par exemple, mais la probabilité d'erreur résultante aurait été supérieure.

Nous procédons de la même façon que pour le calcul précédant. La probabilité d'erreur associée à la seconde stratégie est donnée par

$$P_{err}^{\text{stratégie 2}} = \frac{\text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\}}{\text{card}\Big\{\mathbf{X} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\}}$$

Le dénominateur est en fait égal à :

$$\begin{aligned}\text{card}\Big\{\mathbf{X} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\} = \text{card}\Big\{\mathbf{X} \in \mathcal{A} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\} + \text{card}\Big\{\mathbf{X} \in \mathcal{B} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\} \\ + \text{card}\Big\{\mathbf{X} \in \mathcal{C} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\} + \text{card}\Big\{\mathbf{X} \in \mathcal{D} \mid \widetilde{\Delta}(\mathbf{X}) \geq P_B\Big\}\end{aligned}$$

et donc

$$P_{err}^{\text{stratégie 2}} = \frac{1}{1 + \frac{1}{\text{card}(\mathcal{A})P_{sup}(\mathcal{A},\mathcal{B})}\Big[\text{card}(\mathcal{B})P_{sup}(\mathcal{B},\mathcal{B}) + \text{card}(\mathcal{C})P_{sup}(\mathcal{C},\mathcal{B}) + \text{card}(\mathcal{D})P_{supp}(\mathcal{D},\mathcal{B})\Big]} \tag{5.60}$$

Or,

$$|P_A - P_B| \gg |P_B - P_C| \approx |P_B - P_D|$$

donc le terme dominant au dénominateur est de la forme :

$$\frac{1}{4}2^{\mu_v D_K(P_A,P_B)}$$

où $D_K(P_A, P_B) = 0.019$.
D'où la courbe observée sur la figure 5.12 et la différence de performances observée entre les deux stratégies.
Cette figure illustre la probabilité d'erreur obtenue en appliquant les deux stratégies considérées ; pour chacune de ces stratégies, nous avons également tracé les probabilités d'erreur analytiques. Les probabilités d'erreur analytiques sont extrêmement proches des probabilités d'erreur expérimentales : **l'approximation de l'indépendance entre les vecteurs d'entrée de la fonction est donc *a posteriori* justifiée**.
Examinons la complexité correspondant à cette attaque : en utilisant la seconde stratégie, la longueur de séquence reçue nécessaire à l'obtention de K formes linéaires indépendantes a été de l'ordre de $N \approx 9000$ bits, la probabilité d'erreur correspondante étant de l'ordre de $P_{err} \approx 10^{-2}$. Ceci correspond à un nombre moyen d'équations par vecteur $\mu_v \approx 250$. De par les résultats obtenus au paragraphe 5.4.5 (voir page 187), la complexité globale est alors en $\mathcal{O}(2^{23})$. À titre indicatif, rappelons que la complexité d'une attaque exhaustive serait ici de l'ordre de $\mathcal{O}(2^{39})$.

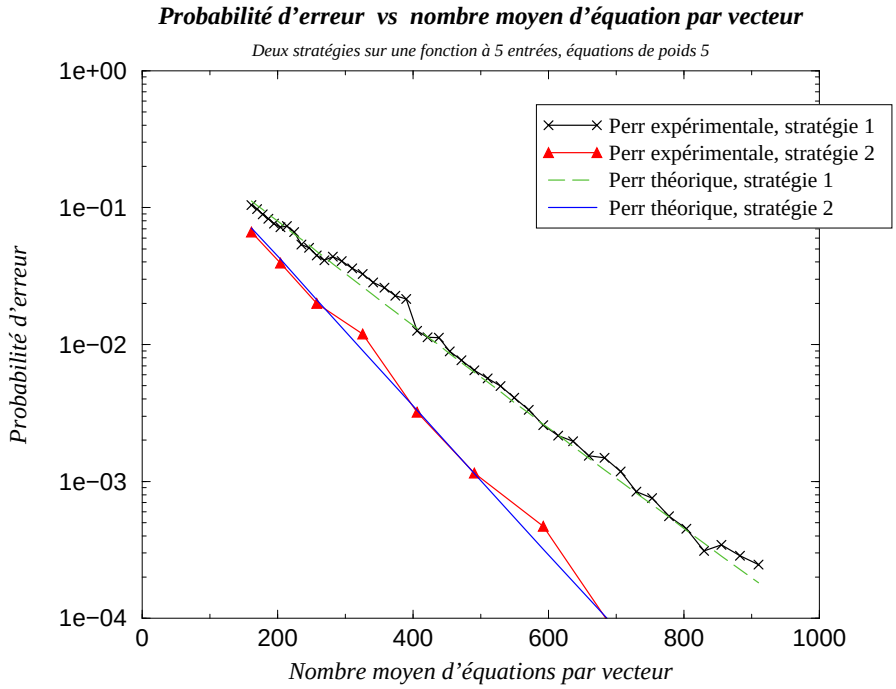


Fig. 5.12 – Probabilités d'erreur de l'algorithme « Probability-matching » en fonction du nombre d'équations vectorielles par vecteur, obtenues sur la fonction f_1 à $n = 5$ entrées.

Résultats obtenus sur la fonction à $n = 7$ entrées, 3-résiliente

Nous considérons la fonction f_{13} dont l'expression est donnée dans l'annexe A. La table $\mathcal{T}$ de cette fonction est donnée dans la table 5.8. Ne figurent dans cette table que les images réciproques des valeurs de Δ_{even} différentes de $1/2$: attention néanmoins à ne pas oublier que dans les deux cas, $z_1 = 0$ et $z_1 = 1$, cinquante-quatre vecteurs d'entrée correspondent à une valeur de Δ_{even} égale à $1/2$.
Comme les valeurs de Δ_{even} pour un bit reçu donné sont assez proches les unes des autres, nous choisissons d'appliquer la stratégie suivante :

– si le bit reçu est égal à 0, nous cherchons les vecteurs d'entrée $\mathbf{X}(t)$ de la fonction qui vérifient $\widetilde{\Delta}_{\text{even}}\Big(\mathbf{X}(t)\Big) < 0.46875$, mais nous ajoutons à l'image réciproque de Δ_{even} les

$z_0 = 0$		$z_0 = 1$	
valeurs de Δ_{even}	Δ_{even}^{-1}	valeurs de Δ_{even}	Δ_{even}^{-1}
0.46875	(0001000), (0000100)	0.484375	(0000001), (0001101)
0.484375	(0010000), (0011100) (0000010), (0001110) (0011011), (0010111)	0.515625	(0011000), (0010100) (0001010), (0000110) (0010011), (0011111)
0.515625	(0001001), (0000101)	0.53125	(0000000), (0001100)

Tab. 5.8 – Construction de la table $\mathcal{T}$ de la fonction f_{13} à $n = 7$ entrées. Poids des équations : $d = 5$.

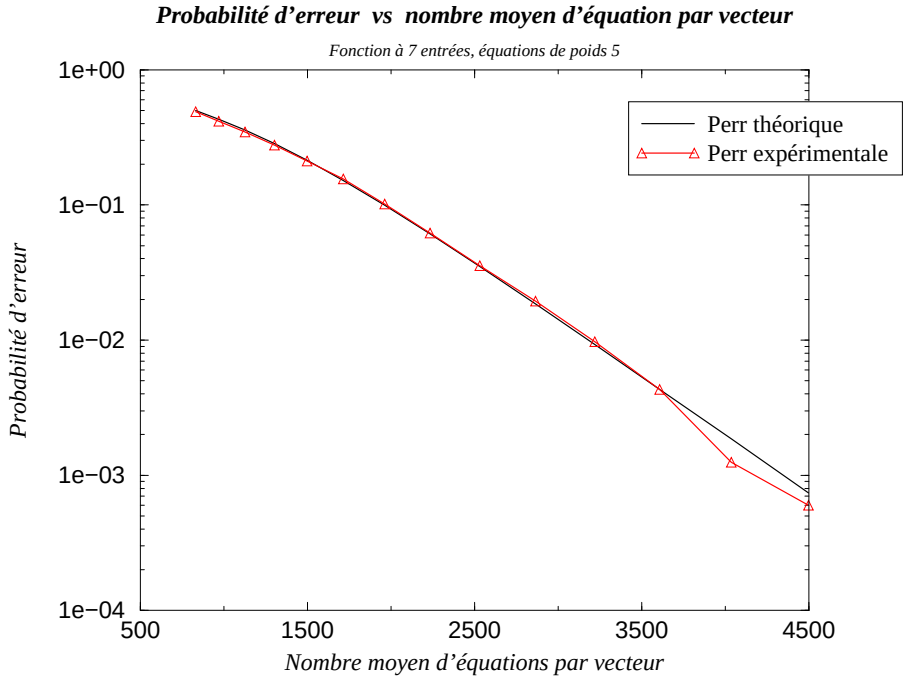


Fig. 5.13 – Probabilités d'erreur de l'algorithme « Probability-matching » en fonction du nombre d'équations vectorielles par vecteur, obtenues sur la fonction f_{13} à $n = 7$ entrées.

vecteurs appartenant à l'image réciproque de $\Delta_{\text{even}} = 0.48375$;

– de la même façon, si le bit reçu est égal à 1, nous cherchons les vecteurs d'entrée vérifiant $\widetilde{\Delta}_{even}\Big(\mathbf{X}(t)\Big) > 0.53125$ en ajoutant à l'image réciproque de 0.53125 les vecteurs contenus dans l'image réciproque de 0.515625.

Nous avons représenté sur la figure 5.13 les résultats expérimentaux et théoriques en fonction du nombre moyen d'équations vectorielles. Comme dans l'exemple précédent, il apparaît nettement que les courbes sont quasiment confondues.
La complexité dans ce cas de figure est de l'ordre de $\mathcal{O}(2^{27})$ opérations élémentaires : pour obtenir une probabilité d'erreur de l'ordre de $P_{err} \approx 10^{-2}$, il nous a fallu $N = 16500$ bits de séquence de clé, ce qui correspond à $\mu_v \approx 3200$ équations en moyenne.

5.4.7 Comparaison du « Probability-matching » et des algorithmes du SOJA

À l'aune des résultats de simulation présentés dans ce chapitre, essayons de comparer les différents algorithmes : SOJA-1, SOJA-2 et « Probability-matching ».

Le tableau 5.9 résume les complexités en terme de calculs, le tableau 5.10 résume les complexités en mémoire des différents algorithmes. La complexité en mémoire de l'algorithme du « Probability-matching » a été arbitrairement fixée à n, puisque cette complexité dépend fortement de la fonction considérée : elle correspond en effet à la mémorisation des formes linéaires constantes associées à une valeur donnée de Δ_{even}. Ayant constaté expérimentalement que le nombre de formes linéaires constantes était de l'ordre de n, nous avons fixé la complexité en mémoire du « Probability-matching » à cette valeur.

Nous constatons tout d'abord que le SOJA-1 est l'algorithme le plus coûteux en terme de calculs et de mémoire. Rappelons également que, si les complexités du SOJA-2 et du « Probability-

Algorithme	**Complexité en calculs**
SOJA-1	$Nd\mu_v 2^n$
SOJA-2	$Nd\mu_v$
« Probability-matching »	$Nd\mu_v$

Tab. 5.9 – Résumé des complexités en calculs des algorithmes SOJA et du « Probability-matching ».

Algorithme	**Complexité en mémoire**
SOJA-1	$N2^n$
SOJA-2	n
« Probability-matching »	n

Tab. 5.10 – Résumé des complexités en mémoire des algorithmes SOJA et du « Probability-matching ».

matching », semblent identiques, ce n'est pas réellement le cas, puisque les opérations effectuées par le « Probability-matching » sont beaucoup moins complexes : il s'agit de simples calculs de parité.

Les résultats présentés sur la fonction f_{13} montrent que le SOJA-1 a les meilleures performances puisqu'il permet d'obtenir une probabilité d'erreur inférieure à 10^{-4} pour une longueur de séquence de $N = 17000$ bits. Le SOJA-2 permet, quant à lui, pour une longueur de séquence identique, d'obtenir une probabilité d'erreur de 2.10^{-2}, et le « Probability-matching » une probabilité d'erreur de 10^{-2} pour une longueur de séquence quasi-identique ($N = 16500$). Cependant, les complexités diffèrent : si l'on s'autorisait une longueur de séquence de $N' = N \times 2^n$ pour le « Probability-matching », la complexité en calculs serait identique à celle du SOJA-1 utilisant une longueur N. Évaluons alors la probabilité d'erreur correspondant à N' sur le « Probability-matching ». Les équations ici utilisées sont générées à partir de l'algorithme Canteaut-Trabbia. Le nombre moyen d'équations de parité par bit obtenu avec cette méthode est (cf. chapitre 2, équation 2.6 page 36) :

$$\frac{Tm(d)}{2N}$$

où $m(d) = \frac{N^{d-1}}{2^K(d-1)!}$.
Multiplier N par 2^n reviendrait à multiplier le nombre d'équations de poids $d = 5$ par $(2^n)^{d-2}$. Or pour $N = 16500$, μ_v vaut 3200. D'après l'allure de la courbe 5.13, si μ_v devient égale à $\mu'_v = 2^{7\times3}$, la probabilité d'erreur obtenue sera nettement inférieure à 10^{-4}. On peut en effet montrer que la courbe de probabilité d'erreur en échelle logarithmique peut être approchée par une loi linéaire d'équation :

$$\log(P_{err}) \approx -2,2.10^{-4}\mu_v - 3,9 \ . \tag{5.61}$$

Ainsi, on peut vérifier que :

$$\mu_v = 3200 \times 2^{21} \Longrightarrow P_{err} \ll 10^{-4} \ .$$

Ainsi, l'algorithme du « Probability-matching » est un assez bon compromis entre performances et complexité de calculs. Les performances de cet algorithme sont cependant déterminées par les propriétés de la fonction booléenne : dans les exemples que nous avons choisis, les fonctions booléennes satisfaisaient la propriété indispensable au fonctionnement du « Probability-matching », à savoir que certaines valeurs de Δ_{even} correspondaient à des combinaisons linéaires constantes sur leur image réciproque. Dans les cas où de telles propriétés ne pourront être exhibées, il nous faudra chercher des combinaisons linéaires non pas constantes, mais satisfaites sur un grand nombre de vecteurs, ou chercher des fonctions de degré supérieur à un. Cela aurait des répercussions en termes de performances et/ou de complexité. Le SOJA peut également présenter une alternative.

5.4.8 Conclusion

Dans ce paragraphe, nous avons décrit un nouvel algorithme de cryptanalyse du registre filtré, que nous avons baptisé « Probability-matching ». Le principe de cet algorithme est proche dans l'esprit de celui de la cryptanalyse linéaire, puisqu'il cherche, à l'aide du plus d'équations possibles, à identifier des bits de clé par l'intermédiaire de certains vecteurs d'entrée, à partir de la probabilité avec laquelle une équation reliant bits de clé et bits observés est vérifiée. Par ailleurs, cet algorithme fait les mêmes hypothèses de départ que l'algorithme du SOJA, puisqu'il utilise des équations vectorielles conjointement avec la sortie de la fonction. Remarquons également que le fait de chercher des formes linéaires constantes sur un ensemble de vecteurs identifiable n'est pas sans rappeler le principe de l'algorithme déterministe du treillis, décrit dans le chapitre 4, paragraphe 4.1.

Nous avons pu, dans le cas du « Probability-matching », évaluer la probabilité d'erreur associée à cet algorithme, probabilité qui s'est révélée très proche de celle obtenue en pratique. Par ailleurs, cet algorithme fait apparaître un nouveau **critère de conception des fonctions booléennes**, dans la mesure où il serait souhaitable que les fonctions utilisées résistent à l'attaque décrite ci-dessus : nous avons en effet observé que la quasi-totalité des fonctions testées présentaient de telles faiblesses, même la fonction à $n = 10$ entrées proposée pour le standard LILI 128(cf. [22]).

Enfin, cet algorithme peut être étendu à des cas où aucune forme linéaire n'est constante sur les ensembles de vecteurs que nous parvenons à identifier. En effet, nous pouvons également :

- chercher des **fonctions de degré supérieur à 1**, constantes sur cet ensemble, par exemple des fonctions quadratiques. Nous pouvons ensuite linéariser le système, et la clé sera à nouveau identifiée à l'aide de l'inversion d'un système linéaire. Le dimension de ce dernier sera toutefois supérieure à K ;
- rechercher des fonctions qui ne sont pas constantes sur tous les vecteurs de l'ensemble identifié, mais sur **un grand nombre d'entre eux**.

5.5 Conclusion

Dans ce chapitre, nous avons présenté plusieurs algorithmes, qui sont tous fondés sur l'utilisation conjointe d'équations vectorielles et de la connaissance de la fonction ; nous avons ainsi pu, à partir de la séquence reçue, calculer des probabilités sur les bits d'entrée. Dans un premier temps, nous avons décrit l'algorithme du SOJA, qui cherche à déterminer des probabilités sur les vecteurs d'entrée, en énumérant les vecteurs qui satisfont les équations vectorielles et correspondent à la séquence observée. Ces probabilités sont ensuite utilisées par des décodeurs : nous avons suggéré d'utiliser un décodeur Belief Propagation, ou un décodage à seuil. Nous avons

observé en pratique que ce dernier, moins complexe, donnait aussi de meilleures performances. Notons que, comme l'ont suggéré A. Canteaut et J-P. Tillich (cf. [14]) et comme nous l'avons déjà mentionné dans le chapitre 4, d'autres types de décodage plus performants que le décodage à seuil pourraient être appliqués sur les probabilités générées par le SOJA.

Une suite naturelle de l'algorithme du SOJA fut d'itérer la génération des probabilités, en fonction de celles déterminées à l'étape précédente : l'algorithme du SOJA-itératif apparaît en fait comme une version vectorielle de l'algorithme Belief Propagation. Malheureusement, l'apparition de cycles dès la deuxième itération freine la convergence de cet algorithme, qui n'est donc utile que sur ses deux premières itérations.

Enfin, nous avons présenté un nouvel algorithme, dit du « Probability-matching». Cet algorithme, peu complexe puisqu'il consiste à examiner la parité de sommes, est à mi-chemin entre le SOJA et la cryptanalyse linéaire. Nous avons simulé plusieurs systèmes : nombre de fonctions ont présenté des faiblesses relativement à cette attaque, faiblesses conditionnées cependant par le poids des équations disponibles.

Conclusions

La vocation de ce document était de présenter quelques algorithmes de cryptanalyse du registre filtré. Dans un premier temps, nous avons étudié le système cryptographique qu'est le registre filtré par une fonction booléenne, et la modélisation dont il fait couramment l'objet dans la littérature : celle d'un codeur dont la sortie est bruitée par un canal binaire symétrique. Nous avons ensuite décrit quelques-unes des nombreuses attaques publiées dans la littérature, que nous avons classifiées en deux catégories : celle des attaques déterministes, celle des attaques probabilistes. Le troisième chapitre a eu pour objet d'évaluer les performances de différents algorithmes de type « Belief Propagation » en faisant varier d'une part, la phase d'initialisation de l'algorithme, d'autre part les paramètres du système : poids des équations de parité, nombre d'équations de parité, probabilité de transition du canal et canal de transition lui-même (binaire symétrique ou booléen). Nous avons, dans un quatrième chapitre, présenté les algorithmes de type « treillis » dont nous sommes à l'origine : ces algorithmes sont, selon les conditions d'application, soit déterministes, soit probabilistes. Enfin le dernier chapitre a été consacré aux algorithmes de type SOJA, simple ou itératif, et à un algorithme à mi-chemin entre la cryptanalyse linéaire et le SOJA que nous avons baptisé « Probability-Matching ».

Dans les trois derniers chapitres, nous avons illustré les algorithmes évoqués par des résultats de simulation, et tenté d'expliquer leur comportement. Malheureusement, dans bien des cas, il nous a paru extrêmement ardu d'étudier leurs performances théoriques, et le lecteur a dû se contenter de leur description et des résultats de simulation. Ceci constitue une piste de recherche potentielle pour tout lecteur souhaitant connaître les performances théoriques des algorithmes que nous avons présentés. Par ailleurs, nos algorithmes utilisent la connaissance de la fonction booléenne impliquée dans le système de chiffrement : il serait intéressant dans chaque cas de construire des fonctions booléennes qui résisteraient à nos attaques, sans qu'elles présentent pour autant trop de faiblesses vis-à-vis des attaques classiques. Enfin, la plupart des algorithmes que nous avons présentés utilisent des équations de parité de poids le plus faible possible, dont la génération reste un problème-clé en cryptanalyse. S'en abstenir, ou du moins contourner ce problème (par le biais d'un test exhaustif par exemple) constituerait une grande avancée. Nous travaillons actuellement à l'élaboration d'un nouvel algorithme qui fragmente le code simplexe tronqué en sous-codes, décodés en utilisant un treillis ; l'importance du poids des équations est alors moindre dans ce cas de figure.

Annexe A : description des fonctions booléennes utilisées

Cette annexe répertorie les fonctions booléennes que nous avons utilisées dans ce document.

Fonction f_1

Cette fonction à $n = 5$ entrées, introduite dans [1], a la forme algébrique normale suivante :

$$f_1(x_1, \ldots, x_5) = x_1 + x_2 + (x_1 + x_3)(x_2 + x_4 + x_5) + (x_1 + x_4)(x_2 + x_3)x_5,$$

ce qui équivaut à la table :

$$01110100001011100011101001011100.$$

Il s'agit d'une fonction 1-résiliente.
$\max_{\mathbf{u}} |\widehat{f_1}(\mathbf{u})| = 8$ ce qui correspond à une probabilité de transition : $p_{\max} = 0.25$.

Fonction f_2

Il s'agit d'une fonction à $n = 5$ entrées, introduite dans [11], et dont la forme algébrique normale est :

$$f(x_1, \ldots, x_5) = x_1x_2x_3 + x_1x_2x_4 + x_1x_2x_5 + x_1x_4 + x_2x_5 + x_3 + x_4 + x_5.$$

La table de cette fonction est

$$00001110101101011101001101101000.$$

C'est une fonction 1-résiliente, plateau (cf. définition dans le chapitre 1).
$\max_{\mathbf{u}} |\widehat{f_2}(\mathbf{u})| = 4$ ce qui équivaut à une probabilité de transition : $p_{\max} = 0.375$.

Fonction f_3

Il s'agit d'une fonction équilibrée à $n = 5$ entrées, non-résiliente. Sa table des valeurs est la suivante :

$$10001001111001010011100010111001.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 3$, $\mathbf{u} = 22$ et $\mathbf{u} = 27$ et vaut $\max_{\mathbf{u}} |\widehat{f_3}(\mathbf{u})| = 6$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.
Les valeurs des transformées de Fourier des formes linéaires de poids 1 sont :

$$\begin{aligned} \widehat{f_3}(1) &= 2 \\ \widehat{f_3}(2) &= 0 \\ \widehat{f_3}(4) &= 2 \\ \widehat{f_3}(8) &= -4 \\ \widehat{f_3}(16) &= 0. \end{aligned}$$

Fonction f_4

Il s'agit d'une fonction équilibrée à $n = 5$ entrées, non-résiliente, dont la table des valeurs est la suivante :

$$11000110111101100001010100001101.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 19$ et $\mathbf{u} = 20$ et vaut $\max_{\mathbf{u}} |\widehat{f_4}(\mathbf{u})| = 6$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.
Les valeurs des transformées de Fourier des formes linéaires de poids 1 sont :

$$\begin{aligned} \widehat{f_4}(1) &= -4 \\ \widehat{f_4}(2) &= -2 \\ \widehat{f_4}(4) &= -2 \\ \widehat{f_4}(8) &= -2 \\ \widehat{f_4}(16) &= 4. \end{aligned}$$

Fonction f_5

La fonction f_5 est une fonction équilibrée à $n = 5$ entrées, n'ayant aucune autre propriété particulière. Sa table des valeurs est la suivante

$$01010100111001001101001010011110.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 7$ et $\max_{\mathbf{u}} |\widehat{f_5}(\mathbf{u})| = 6$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.

$$\begin{aligned} \widehat{f_5}(1) &= -2 \\ \widehat{f_5}(2) &= 4 \\ \widehat{f_5}(4) &= 4 \\ \widehat{f_5}(8) &= -2. \end{aligned}$$

Fonction f_6

La fonction f_6 est une fonction équilibrée à $n = 5$ entrées, introduite dans [11].
Sa forme algébrique normale est la suivante :

$$f_6(x_1, \ldots, x_5) = x_1x_2x_3 + x_1x_4 + x_2x_5 + x_3.$$

Il s'agit d'une fonction plateau non résiliente, telle que $\max_{\mathbf{u}} |\widehat{f_6}(\mathbf{u})| = 4$ ce qui équivaut à une probabilité de transition : $p_{\max} = 0.375$.

Fonction f_7

Il s'agit d'une fonction à $n = 5$ entrées, équilibrée, n'ayant aucune propriété particulière. Sa table des valeurs est la suivante :

$$00110001010111100110110000011110.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 6, 13, 15$, et 26 et vaut $\max_{\mathbf{u}} |\widehat{f_7}(\mathbf{u})| = 6$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.

Fonction f_8

Il s'agit d'une fonction à $n = 5$ entrées, équilibrée, n'ayant aucune propriété particulière. Sa table des valeurs est la suivante :

$$00110100100011100010011111101010.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 9$, 10 et 28 et vaut $\max_{\mathbf{u}} |\widehat{f_8}(\mathbf{u})| = 6$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.

Fonction f_9

Il s'agit d'une fonction à $n = 6$ entrées, n'ayant aucune propriété particulière de résilience particulière. Sa table est la suivante :

$$01010100111001001101001010011111$$
$$11001100011111111011000000000010.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 48$ $\max_{\mathbf{u}} |\widehat{f_9}(\mathbf{u})| = 10$ ce qui correspond à une probabilité de transition : $p_{\max} = 0.34375$.
Les valeurs des transformées de Fourier des formes linéaires de poids 1 sont :

$$\begin{aligned}
\widehat{f_9}(1) &= -2 \\
\widehat{f_9}(2) &= 4 \\
\widehat{f_9}(4) &= 4 \\
\widehat{f_9}(8) &= -4 \\
\widehat{f_9}(16) &= 4 \\
\widehat{f_9}(32) &= 2.
\end{aligned}$$

Fonction f_{10}

Il s'agit d'une fonction à $n = 6$ entrées, introduite dans [11]. La forme algébrique normale de cette fonction est

$$f_{10} = x_1x_2x_3 + x_2x_3x_6 + x_1x_2 + x_3x_4 + x_5x_6 + x_4 + x_5.$$

La table de cette fonction est

$$01100111011010000110011110010111$$
$$01100100011010111001101101101011.$$

C'est une fonction équilibrée.
$\max_{\mathbf{u}} |\widehat{f_{10}}(\mathbf{u})| = 8$ et est atteint 32 fois et correspond à une probabilité de transition : $p_{\max} = 0.375$.

Fonction f_{11}

Il s'agit d'une fonction à $n = 6$ entrées, équilibrée, n'ayant aucune propriété particulière.
La table de cette fonction est

$$10001001111001010011100010111001$$
$$01011000001001101101010111110010.$$

C'est une fonction équilibrée. Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 54$ et vaut $\max_{\mathbf{u}} |\widehat{f_{11}}(\mathbf{u})| = 10$ ce qui correspond à une probabilité de transition : $p_{\max} = 0.3475$.

Fonction f_{12}

Il s'agit d'une fonction à $n = 6$ entrées, équilibrée, n'ayant aucune propriété particulière.
La table de cette fonction est

$$00100010111111110000101100101011$$
$$11010100111100100000001001111001.$$

C'est une fonction équilibrée. Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 8$ et vaut $\max_{\mathbf{u}} |\widehat{f_{12}}(\mathbf{u})| = 12$, ce qui correspond à une probabilité de transition : $p_{\max} = 0.3125$.

Fonction f_{13}

Il s'agit d'une fonction à $n = 7$ entrées, 3-résiliente, plateau. Sa forme algébrique normale est la suivante :

$$\begin{aligned} f_{13}(x) = \quad 1 \quad & +x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_1x_7 + x_2(x_3 + x_7) + \\ & x_1x_2(x_3 + x_6 + x_7)\ , \end{aligned}$$

ce qui correspond à la table :

$$10010100011010110110101110010100$$
$$01111010100001011000010101111010$$
$$11100011000111000001110011100011$$
$$00001101111100101111001000001101.$$

Cette fonction est plateau, de paramètre $r = 4$: $\forall \mathbf{u}\ |\widehat{f_{13}}(\mathbf{u})| = 0$ ou 2^4, ce qui équivaut à une probabilité de transition : $p_{\max} = 0.375$.

Fonction f_{14}

Il s'agit d'une fonction équilibrée à $n = 7$ entrées, non-résiliente, dont la table des valeurs est la suivante :

$$01010100111001001101001010011111$$
$$01001100011111111011000010000011$$
$$10111100111111000110100001101011$$
$$10010010110000010010001110100010.$$

Le pic de transformée de Fourier de cette fonction est atteint en $\mathbf{u} = 83$ et $\mathbf{u} = 87$, $\max_{\mathbf{u}} |\widehat{f}_{14}(\mathbf{u})| = 16$ ce qui correspond à une probabilité de transition : $p_{\max} = 0.375$.
Les valeurs des transformées de Fourier des formes linéaires de poids 1 sont :

$$\begin{aligned}
\widehat{f}_{14}(1) &= 4\\
\widehat{f}_{14}(2) &= 2\\
\widehat{f}_{14}(4) &= 8\\
\widehat{f}_{14}(8) &= -10\\
\widehat{f}_{14}(16) &= 4\\
\widehat{f}_{14}(32) &= 8\\
\widehat{f}_{14}(64) &= 2.
\end{aligned}$$

Fonction f_{15}

Il s'agit d'une fonction à $n = 8$ entrées, 2-résiliente, plateau, de forme algébrique :

$$\begin{aligned}
f_{15}(x) = x_1 &+ x_4 + x_5 + x_6 + x_7 + x_1(x_2 + x_7) + x_2x_6 + x_3(x_6 + x_8)\\
&+ x_1x_2(x_4 + x_6 + x_7 + x_8) + x_1x_3(x_2 + x_6) + x_1x_2x_3(x_4 + x_5 + x_8).
\end{aligned}$$

La table équivalente est :

0100010110101010101110110101010 0
1001001001111101011011001000001 1
1111111000010001000000001110111 1
0010100111000110110101110011100 0
0101101010110101101001000100101 1
1000110101100010011100111001110 0
1110000100001110000111111111000 0
0011011011011001110010000010011 1.

Cette fonction est plateau, de paramètre $r = 6$ $\forall \mathbf{u}$ $|\widehat{f}_{15}(\mathbf{u})| = 0$ ou 2^4 ce qui correspond à une probabilité de transition : $p_{\max} = 0.4375$.

Fonction f_{16}

Cette fonction est une fonction à $n = 10$ entrées, qui a été proposée pour le standard LILI-128 [22]. La table de cette fonction est la suivante :

0011110011000011110000110011110
0011110011000011110000110011110
0011110011000011110000110011110
1100001100111100001110011000011
0101101010100101101001010101101
0101101010100101101001010101101
0101101010100101101001010101101
1010010101011010010110101010101
0110011010011001100110010110011
0110011010011001100110010110011
0110100101101001100101101001011
0110100110010110011010011001011
0000111111110000111100000000111
1111000000001111000011111111000
0011001111001100110011000011001
1100110000110011001100111100110
0011110000111100110000111100001
1100001111000011001111000011110
0011110011000011001111001100001
1100001100111100110000110011110
0101010110101010101010100101010
1010101001010101010101011010101
0101101001011010101001011010010
1010010110100101010110100101101
0101101010100101010110101001010
1010010101011010101001010101101
0110011001100110100110011001100
1001100110011001011001100110011
0110011010011001011001101001100
1001100101100110100110010110011
0110100101101001011010010110100
1001011010010110100101101001011.

Il s'agit d'une fonction 3-résiliente.
$\max_{\mathbf{u}} |\widehat{f_{16}}(\mathbf{u})| = 32$ ce qui équivaut à une probabilité de transition : $p_{\max} = 0.46875$.

Annexe B : matrice de transition du 90-150 testé au chapitre 4

Le 90-150 est un automate cellulaire tel que l'état de chaque cellule résulte d'un XOR entre l'état précédent de ses cellules adjacentes et éventuellement de son propre état précédent.

La matrice de transition d'un tel automate comporte donc des '1' de part et d'autre de la diagonale (excepté sur la première et la dernière colonne qui concernent les cellules situées aux extrémités et qui dépendent de l'état de leur unique cellule voisine, et éventuellement de leur état précédent).
Un tel automate est donc entièrement caractérisé par la diagonale de sa matrice de transition.
Dans le cas testé au chapitre 4, la diagonale du 90-150 testé est la suivante :

$$0,1,1,0,1,0,1,0,1,1,0,1,0,0,0,0,1,0,1,1,0,$$
$$1,1,0,0,0,1,1,1,1,0,1,0,1,0,0,0,0,1,1,1,0,$$
$$1,1,0,1,1,1,0,0,1,1,1,0,0,0,1,0,0,1,0,1,0,1,1.$$

La matrice de transition associée à cet automate était donc

$$\mathbf{G_{90-150}} = \begin{pmatrix} \mathbf{0} & 1 & 0 & \dots & \dots & 0 \\ 1 & \mathbf{1} & 1 & \ddots & & \vdots \\ 0 & 1 & \mathbf{1} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & & 1 & 0 \\ \vdots & & \ddots & 1 & \mathbf{1} & 1 \\ 0 & \dots & \dots & 0 & 1 & \mathbf{1} \end{pmatrix},$$

où la diagonale est surlignée à des fins de visibilité.

Annexe C : quelques notions sur les codes convolutifs

Un code convolutif (cf. [46] pour plus de renseignements) de rendement $\frac{1}{m+1}$ et de mémoire B est décrit de la façon suivante : soit u_i la séquence d'entrée, et $\underline{v}_n$ le vecteur codé de taille $1 \times (m+1)$. Alors

$$\mathbf{v_n} = u_n \mathbf{G_0} + u_{n-1} \mathbf{G_1} + \ldots + u_{n-B} \mathbf{G_B}$$

où $\mathbf{G_i}$ sont des vecteurs colonnes de taille $m+1$.
La matrice génératrice d'un tel code s'écrit alors :

$$\mathbf{G} = \begin{pmatrix} \ddots & \ddots & & \ddots & & & \\ & \mathbf{G_0} & \mathbf{G_1} & \ldots & \mathbf{G_B} & & \\ & & \mathbf{G_0} & \mathbf{G_1} & \ldots & \mathbf{G_B} & \\ & & & \ddots & \ddots & & \ddots \end{pmatrix}$$

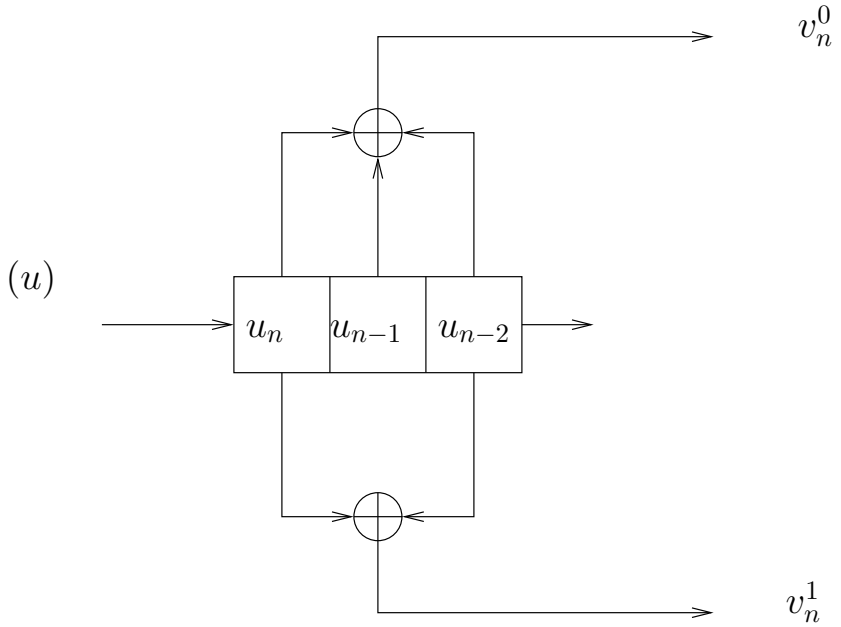


FIG. 5.14 – Code convolutif (7,5), mémoire $B = 3$.

La figure 5.14 représente le code convolutif (7,5) : dans la représentation ci-dessus on a

$$\mathbf{G_0} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\mathbf{G_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$\mathbf{G_2} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Ce code est dit "(7,5)", car ces chiffres écrits en binaire correspondent aux connections : si l'on note $\mathbf{G_i}^j$ la j-ième composante du vecteur $\mathbf{G_i}$, on constate que

$$\sum_{j=1}^{3} \mathbf{G_j}^1 2^j = 7 \tag{5.62}$$

$$\sum_{j=1}^{3} \mathbf{G_j}^2 2^j = 5 \tag{5.63}$$

Posons $\mathbf{G_j} = (g_{1,j}, \ldots, g_{m+1,j})^\dagger$. Les codes convolutifs admettent une représentation polynomiale :

- le code convolutif peut, de façon générale, avoir en entrée k bits ; alors, $\forall i \in [1,k]$ soit $e_i(x) = \sum_{j=0}^{+\inf} e_{i,j} x^j$; où $e_{i,j}$ est le j-ième bit en entrée du i-ème codeur.
- de même en sortie, $\forall t \in [1, m+1]$, on a $s_t(x) = \sum_{t=0}^{+\inf} s_{t,j} x^j$
- le polynôme générateur reliant la i-ème entrée à la t-ième sortie est noté $g_{i,t}$.

On a alors

$$s_t(x) = \sum_{j=1}^{k} e_j(x) g_{j,t}(x)$$

où la multiplication par x correspond à un retard temporel, comme nous l'avons vu précédemment (cf. chapitre 1).
Dans l'exemple ci-dessus, on a $k = 1$ et $m+1 = 3$, $g_1(x) = 1 + x + x^2$ et $g_2(x) = 1 + x^2$.
Enfin, un code convolutif est dit **récursif** lorsque les polynômes générateurs sont remplacés par des fractions polynomiales : certains bits de sortie sont alors réinjectés en entrée du codeur.

Annexe D : algorithme de Viterbi

L'algorithme de décodage de Viterbi [28] est associé à un parcours dans un treillis et est donc généralement dédié aux codes sujets à un effet mémoire, comme les codes convolutifs. Cet algorithme cherche à minimiser la distance entre le mot bruité reçu et un mot du code. La distance que nous venons d'évoquer peut être euclidienne (dans le cas d'un canal AWGN) ou de Hamming (dans le cas d'un canal de type hard, ou si le démodulateur effectue un décodage à seuil sur le signal reçu en sortie du canal). La minimisation de la métrique se fera « pas-à-pas », c'est-à-dire qu'à chaque instant dans le treillis, nous minimisons la métrique « locale ».

Soit ν le nombre d'états dans le treillis considéré. Nous notons σ_i le i-ème état dans le treillis, $pred_j(\sigma_i)$ le j-ième prédécesseur de σ_i.
Notons $\lambda(\sigma_i, t)$ la métrique cumulée à l'instant t évaluant ainsi le coût cumulé d'être à l'état σ_i, relativement à la séquence reçue.
Soit $T(pred_j(\sigma_i), \sigma_i)$ la branche reliant le j-ième prédécesseur de σ_i à σ_i.
Nous supposons que le treillis démarre à l'état $\sigma_0 = \mathbf{0_n}$: ceci est souvent le cas lorsque l'on considère des codes convolutifs. L'algorithme est alors décrit dans le tableau 5.11.

L'observation est constituée des symboles bruités par le canal.
La distance d^2 est la distance Euclidienne lorsque le canal est gaussien, la distance de Hamming lorsqu'il s'agit d'un canal à sorties dures.
En moyenne lorsque le code convolutif est de mémoire $B+1$, la largeur de la fenêtre décodée est de l'ordre de $W = 5 \times (B+1)$.

Algorithme de Viterbi

1. ***Initialisation :***

$$\begin{aligned} \lambda(0, t=0) &= 0 \\ \lambda(\sigma_i, t=0) &= +\inf, \quad \forall i \neq 0 \end{aligned}$$

2. ***Déroulement :***
   À l'instant t, pour tout état σ_i, on choisit de conserver la branche incidente qui minimise la métrique cumulée, on mémorise cette transition, et on réactualise la métrique :

$$\lambda(t, \sigma_i) \leftarrow \min_{pred_j \sigma_i} \left[\lambda(t-1, pred_j(\sigma_i)) + d^2(T(pred_j(\sigma_i), \sigma_i), \text{observation}) \right]$$

3. ***Terminaison :***
   $t = W$: on choisit la dernière branche correspondant à la plus petite métrique cumulée, et à l'aide des survivants mémorisés à chaque instant, on remonte le treillis pour trouver le chemin minimal correspondant.

TAB. 5.11 – Algorithme de Viterbi.

Annexe E : brève présentation des Turbo-Codes

Un turbo-code (cf. [4], [5], ...) est constitué de deux codes, généralement convolutifs, séparés par un entrelaceur. Il existe principalement deux structures de turbo-codes :

- les turbo-codes parallèles, dont un schéma est donné sur la figure 5.15 : les N bits d'information (x) sont codés par le premier codeur et par le second après avoir subi un entrelaçage. Les codes sont des codes RSC, *i.e.* des codes convolutifs récursifs. L'entrée du second codeur est, à une permutation près, identique à celle du premier ; les bits d'information ne sont donc pas transmis sauf cas particulier (lorsque le canal est mauvais). Le rendement d'un tel code est

$$\begin{aligned} R &\approx \frac{R_1 R_2}{R_1 + R_2 - R_1 R_2} \\ &\approx \frac{R_1}{2 - R_1} \end{aligned}$$

  si les deux codes sont identiques ;
- les turbo-codes série, dont la structure est présentée sur la figure 5.16 : deux codes, séparés par un entrelaceur sont ici concaténés. Le premier code est dit code externe, le second est dit code interne. Ce dernier doit toujours être un code RSC (récursif). Le rendement d'un tel code est

$$R \approx R_1 \times R_2 \ .$$

Les performances du turbo-décodage se rapprochent de celles du décodage optimal qui est celui du décodage à maximum de vraisemblance, inutilisable en pratique sur des codes concaténés, car présentant une complexité prohibitive.
Le décodage turbo est un décodage bit à bit, qui maximise la probabilité *a posteriori* de chacun d'entre eux. Notant b_i les bits d'information, c_i les bits codés et (z_i) les bits reçus après passage dans le canal, on peut montrer que

$$\begin{aligned} APP(c_i) &\propto \sum_{c \in C | c_i} \prod_{j=1}^{J} p(z_j|c_j)\pi(c_j) \\ APP(b_i) &\propto \sum_{c \in C | b_i} \prod_{j=1}^{J} p(z_j|c_j)\pi(c_j), \end{aligned}$$

où $\pi(c_j)$ est la probabilité *a priori* du bit c_j.

Comme dans le cas du décodage de Gallager décrit au chapitre 3, à chaque itération, le décodeur turbo évalue une extrinsèque sur le bit b_i (respectivement c_i) en fonction des autres

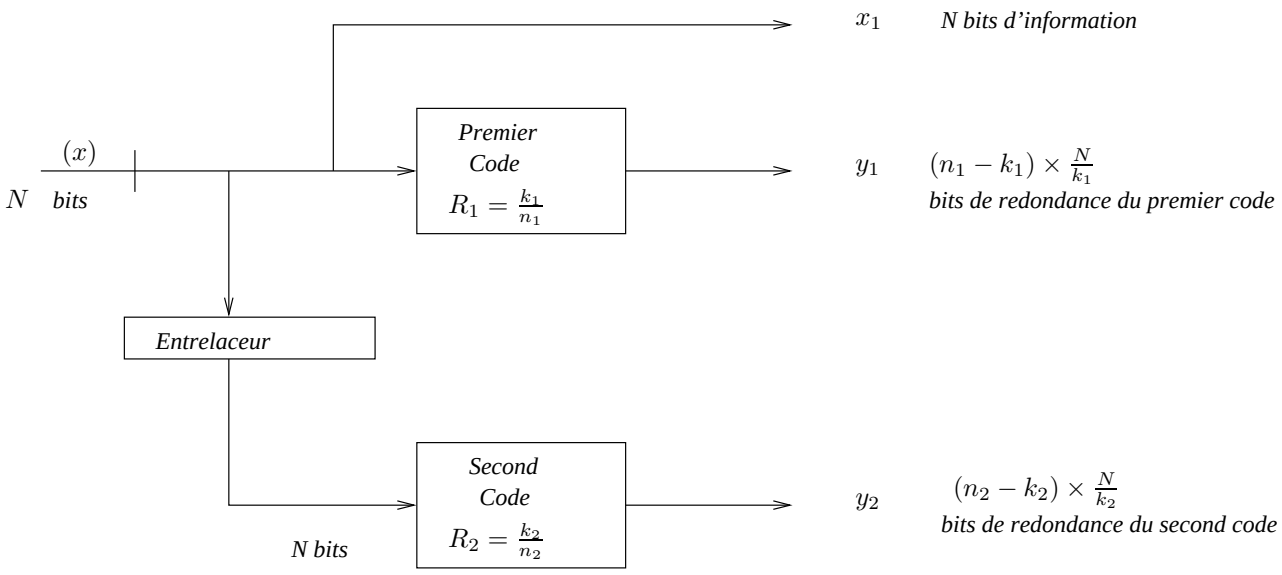


FIG. 5.15 – Structure d'un turbo-code parallèle.

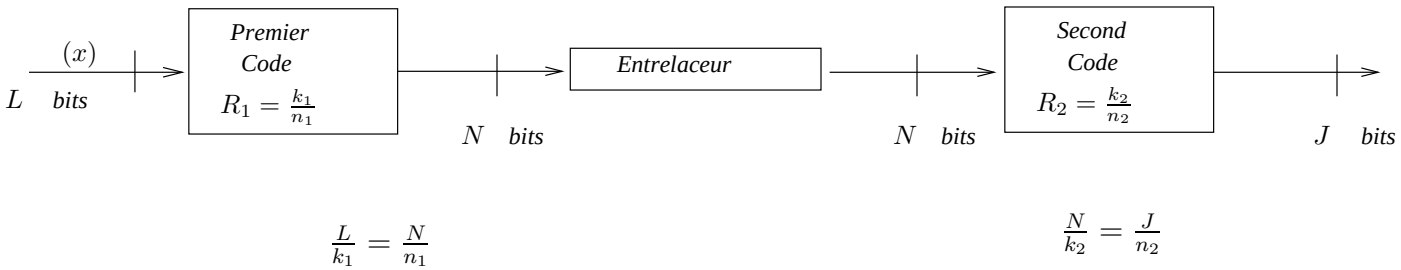


FIG. 5.16 – Structure d'un turbo-code série.

bits différents de b_i respectivement c_i. Cette information est alors entrée comme probabilité *a priori* au décodeur suivant. Une fois le nombre d'itérations souhaitées atteint, une probabilité *a posteriori* globale est calculée, permettant une décision sur chacun des bits.

Annexe F : algorithme Forward-Backward

Dans cette annexe, nous décrivons le principe du fonctionnement de l'algorithme forward-backward. Cette annexe s'inspire fortement de [77], document auquel le lecteur pourra se référer afin d'avoir une description plus complète de l'algorithme (voir également [3]).

Supposons que nous souhaitions décoder un code représentable sous forme de treillis, bruité par un canal sans mémoire. L'algorithme forward-backward permet de générer une probabilité *a posteriori* sur les états du treillis et sur les transitions associées aux états.

L'algorithme de Viterbi décrit dans l'annexe D permet de minimiser la probabilité d'erreur par mot ; l'algorithme forward-backward cherche quant à lui à minimiser une probabilité d'erreur par symbole.

L'ensemble des états est noté S_t et est de cardinal M.

Soit S_t un état à l'instant t. Ces états prennent les valeurs appartenant à l'intervalle $[0; M-1]$.

Le canal est représenté sur la figure 5.17. Le fait qu'il soit sans mémoire se traduit par la relation

$$P(y_1, \ldots, y_N \mid x_1, \ldots, x_N) = \prod_{i=1}^{N} P(x_i|y_i).$$

Entrées de l'algorithme :

- probabilité de transition du canal sans mémoire ;
- treillis du code ;
- séquence (y).

Sorties de l'algorithme :

1. $\forall t,\ Prob\,(S_t = m \mid (y))$.
2. $\forall t,\ Prob\,(S_t = m, S_{t-1} = m' \mid (y))$.

En d'autres termes, l'algorithme va, à partir de la séquence observée en sortie du canal, nous renseigner sur la probabilité d'un état (équation (1)) et sur la probabilité des transitions entre états (équation (2)).

L'algorithme va en fait calculer les quantités que nous définissons ci-après :

$$\begin{aligned} \sigma_t(m) &= Prob\,(S_t = m, (y)) \\ \theta_t(m, m') &= Prob\,\big(S_{t-1} = m, S_t = m', (y)\big) \end{aligned}$$

puisque ces deux quantités, par Bayes, sont proportionnelles aux quantités précitées 1, 2.



FIG. 5.17 – Passage de la séquence (x) à travers un canal sans mémoire.

Nous définissons également les quantités suivantes

$$\begin{aligned} \alpha_t(m) &= Prob\,(S_t = m, (y)) \\ \beta_t(m) &= Prob\,(y_{t+1} \ldots, y_N \mid S_t = m) \\ \gamma_t(m, m') &= Prob\,\big(S_t = m, y_t \mid S_{t-1} = m'\big) \end{aligned}$$

Les deux quantités cherchées, σ et θ, s'expriment en fonction de α, β, γ par les relations

$$\sigma_t(m) = \alpha_t(m)\beta_t(m) \tag{5.64}$$

$$\theta_t(m) = \alpha_{t-1}(m)\gamma_t(m, m')\beta_t(m') \tag{5.65}$$

où α_t, β_t et γ_t peuvent être calculés à l'aide des récurrences :

$$\alpha_t(m) = \sum_{i=0}^{M} \alpha_{t-1}(i)\gamma_t(m, i) \tag{5.66}$$

$$\beta_t(m') = \sum_{j=0}^{M} \beta_{t+1}(j)\gamma_{t+1}(j, m') \tag{5.67}$$

et si la source est équidistribuée, équilibrée, alors

$$\gamma_t(m, m') = \sum_x p(y_t|x) \times Prob\,[x \mid S_{t-1} = m, S_t = m'] \tag{5.68}$$

La multiplication des probabilités de transition $p(y_t|x)$ par le terme $Prob\,[x \mid S_{t-1} = m, S_t = m'\,]$ revient à dire que l'on ne somme que sur les transitions valides, celles qui peuvent correspondre au bit observé.
Pour appliquer l'algorithme, nous supposons débuter à l'état 0, et terminer la description du treillis à l'état 0 également.
L'algorithme est alors donné dans le cadre 5.12.

Algorithme Forward-Backward

– **Entrées :** (y), probabilités de transition du canal, treillis du code.
– **Initialisation :**

$$\begin{aligned} \alpha_0(0) &= 1 \\ \forall\, m \in [1; M-1],\ \alpha_0(m) &= 0 \end{aligned}$$

$$\begin{aligned} \beta_0(M-1) &= 1 \\ \forall\, m \in [0; M-2],\ \beta_N(m) &= 0 \end{aligned}$$

– **Coeur de l'algorithme :**

1. Calcul des γ_t avec l'équation (5.68).
2. Description du treillis de gauche à droite : calcul des α_t avec l'équation (5.66).
3. Description du treillis de droite à gauche : calcul des β_t avec l'équation (5.67).
4. Mise en commun des informations pour le calcul de σ_t et θ_t avec les équations (5.64) et (5.65) .

TAB. 5.12 – Algorithme Forward-Backward.

Bibliographie

[1] R.J. Anderson, "Searching for the optimum correlation attack," *Fast Software Encryption -Leuven'94,* Lectures Notes in Computer Science, vol. 1008, B. Preneel ed., Springer-Verlag, p. 137-143, 1995.

[2] G. Ars, J-C. Faugere, "Gröbner attack of Nonlinear Filter Generator with a small number of Key bits," to be published.

[3] L.R. Bahl, J. Cocke, F. Jelinek and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rates," *IEEE Transactions on Information Theory,* vol. IT-20, 1974, pp. 284-287.

[4] C. Berrou, A. Glavieux and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding : Turbo codes (I)," in *Proc. ICC'93,* Geneva, Switzerland, June 1993, pp. 1064-1070.

[5] J.J. Boutros, "Turbo codes parallèles et séries. Décodages SISO Itératif et Performances ML," cours ENST.

[6] P. Camion, C. Carlet, P. Charpin and N. Sendrier, "On correlation immune functions," *Advances in Cryptology - CRYPTO'91,* Lecture Notes in Computer Science, vol. 576, pp. 86-100, Springer Verlag,1992.

[7] P. Camion and A. Canteaut, "Correlation-immune and resilient functions over a finite alphabet and their applications in cryptography," *Designs, Codes and Cryptography,* (16) 1999, pp. 121-149.

[8] A. Canteaut, C. Carlet, P. Charpin and C. Fontaine, "Propagation characteristics and correlation-immunity of highly nonlinear Boolean functions," *Advances in Cryptology - EUROCRYPT 2000,* Lecture Notes in Computer Science, vol. 1807, pp. 507-522, Springer Verlag, 2000.

[9] A. Canteaut, "Attaques de cryptosystèmes à mots de poids faible et construction de fonctions t-résilientes," Thèse de doctorat de l'Université Paris 6.

[10] A. Canteaut and M. Trabbia, "Improved fast correlation attacks using parity-check equations of weight 4 and 5," *Advances in Cryptology - EUROCRYPT 2000,* Lecture Notes in Computer Science, vol. 1807, pp. 573-588, Springer Verlag, 2000.

[11] A. Canteaut and E. Filiol, "On the Influence of the Filtering Function on the Performance of Fast Correlation Attacks on Filter Generators," *23rd Symposium on Information Theory in the Benelux - May 2002,* Louvain-La-Neuve, Belgium.

[12] A. Canteaut and P. Charpin, "Decomposing bent function,"*IEEE Transactions on Information Theory,* 49(8), August 2003.

[13] A. Canteaut, Communication personnelle, Novembre 2003.

[14] A. Canteaut et J-P. Tillich, Communication personnelle, Novembre 2003.

[15] J. Chen and M. Fossorier, "Near Optimum Universal Belief Propagation Based Decoding of Low-Density Parity Check Codes," *IEEE Transactions on Communications,* to appear.

[16] V.V. Chepyzhov, 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, pp. 181-195, Springer Verlag, 2001.

[17] P. Chose, A. Joux and M. Mitton, "Fast correlation attacks : an algorithmic point of view," *Advances in Cryptology - EUROCRYPT 2002,* Lecture Notes in Computer Science, vol. 2332, pp. 209-221, Springer Verlag, 2002.

[18] A. Clark, J.Dj. Golic and E. Dawson, "A Comparison of Fast Correlation Attacks," *Fast Software Encryption 1996*, Lecture Notes in Computer Science, pp. 145-157.

[19] T.M. Cover and J.A. Thomas : *Elements of information theory.* Wiley series in Telecommunications, 1991.

[20] N. Courtois, "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback," *Crypto 2003*, Lecture Notes in Computer Science 2729, pp : 177-194, Springer.

[21] N. Courtois, W. Meier, "Algebraic Attacks on Stream Ciphers with Linear Feedback," *Advances in Cryptology - Eurocrypt 2003*, Lecture Notes in Computer Science, 2656, pp. 345-359, Springer.

[22] E. Dawson, J. Golić, W. Millan, L. Penna and L. Simpson, "The LILI-128 keystream generator," *Proceedings of the First NESSIE Conference 2000,* Leuwen, 2000.

[23] J-C. Faugère, G. Ars, "An Algebraic Cryptanalysis of Nonlinear Filter Generators using Gröbner bases," rapport INRIA, n° 4739, février 2003.

[24] Maria Fedorova, Yuriy Tarannikov, "On the Constructing of Highly Nonlinear Resilient Boolean Functions by Means of Special Matrices," *INDOCRYPT'2001,* pp. 254-266.

[25] E. Filiol, "Decimation Attack of Stream Ciphers,".*INDOCRYPT'2000,* Lecture Notes in Computer Science, vol. 1977, pp. 31-42, Springer Verlag, 2000

[26] E. Filiol, "Techniques de reconstruction en cryptologie et théorie des codes", PhD Thesis, Ecole Polytechnique, 2001.

[27] C. 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," thèse de doctorat, université Paris VI, 1998.

[28] G.D. Forney, "The Viterbi algorithm," *Proceedings of the IEEE,* vol. 61, no. 3, March 1973, pp.268-278.

[29] G. D. Forney, "Codes on graphs : normal realizations," *IEEE Transactions on Information Theory,* vol. 47(2), pp. 520–548, 2001.

[30] M. Fossorier and S. Lin, "Soft-Decision Decoding of Linear Block Codes Based on Ordered Statistics," *IEEE Transactions on Information Theory,* vol. IT-41, pp.1379-1396, September 1995.

[31] M. Fossorier and S. Lin, "Computationally Efficient Soft-Decision Decoding of Linear Block Codes Based on Ordered Statistics," *IEEE Transactions on Information Theory,* vol. IT-42, pp. 738-750, May 1996.

[32] M. Fossorier, M. Mihaljevič and H. Imai, "Reduced Complexity Iterative Decoding of Low Density Parity Check Codes Based on Belief Propagation," *IEEE Transactions on Communications,* vol. COM-47, pp. 673-680, May 1999.

[33] M. Fossorier and S. Lin, "Reliability-Based Information Set Decoding of Binary Linear Codes," *IEICE Transactions on Fundamentals,* vol. E82-A, pp. 2034-2042, October 1999.

[34] M. Fossorier, M. Mihaljevič and H. Imai, "Critical Noise for Convergence of Iterative Probabilistic Decoding with Belief Propagation in Cryptographic Applications," *AAECC 1999,* Lecture Notes in Computer Sciences, vol. 1719, pp. 282-293, November 1999.

[35] R.G. Gallager, "Low-density parity-check codes," *IRE Trans. Inform. Theory,* vol. IT-8, pp. 21-28, Jan. 1962.

[36] R.G. Gallager : *Low Density Parity check codes*. MIT Press, Cambridge, MA, 1963.

[37] E. Garrido : Communication interne, Thales Communication, France.

[38] E. Garrido et C. Poli : Communication interne, Thales Communication, France.

[39] J. Golic, M. Mihaljevič, "A generalized correlation attack on a class of stream ciphers based on the Levenshtein distance," *Journal of Cryptology* vol. 3, 1991, pp. 201-212.

[40] J. Golic, "Computation of low-weight parity-check polynomials," *Electronics Letters,* vol. 32, pp. 1981-1982, 1996.

[41] J. Golic, "On the Security of Nonlinear Filter Generators," *Fast Software Encryption,* Lecture Notes in Computer Science, 1996, pp. 173-188

[42] J. Golic, A. Clark and E. Dawson, "Generalized inversion attack on nonlinear filter generators," *IEEE Transactions on computers,* vol.49, NO. 10, October 2000.

[43] S.W. Golomb : *Shift register sequences*. Holden-Day, San Francisco, 1967.

[44] P. Guillot, "Fonctions courbes binaires et transformation de Möbius," thèse de doctorat, université de Caen, 1999.

[45] J. Hagenauer, E. Offer and L. Papke, "Iterative Decoding of binary block and convolutional codes," *IEEE Transactions on Information Theory,* vol. 42, pp. 429-445, Mar. 1996.

[46] R. Johanesson and K.Sh. Zygangirov "Fundamentals of Convolutionnal Coding," IEEE Press, 1999.

[47] T. Johansson and F. Jönsson, "Improved fast correlation attack on stream ciphers via convolutional codes," *Advances in Cryptology - EUROCRYPT'99,* Lecture Notes in Computer Science, vol. 1592, pp. 347-362, Springer Verlag, 1999.

[48] T. Johansson and F. Jönsson, "Fast correlation attacks based on turbo code techniques," *Advances in Cryptology - CRYPTO'99,* in Lecture Notes in Computer Science, vol. 1666, pp. 181-197, Springer Verlag, 1999.

[49] T. Johansson and F. Jönsson, "Fast correlation attacks through reconstruction of linear polynomials," *Advances in Cryptology - CRYPTO'2000,* Lecture Notes in Computer Science, vol. 1880, pp. 300-315, Springer Verlag, 2000.

[50] T. Johansson and F. Jönsson, "Theoretical Analysis of a Correlation Attack Based on Convolutional Codes," *IEEE Transactions on Information Theory,* vol. 48, August 2002 pp. 2173-2181.

[51] T. Johansson and F. Jönsson, "On the complexity of Some Cryptographic Problems Based on the General Decoding Problem," *IEEE Transactions on Information Theory,* vol. 48, October 2002 pp. 2669-2678.

[52] F. Jönsson and T. Johansson, "A fast correlation attack on LILI-128," *Information Processing Letters*, 81(3) ; 127-132, February 2002.

[53] F. Jönsson, "Some results on fast correlation attacks", PHD Thesis, 2002, Lund University.

[54] F.R. Kschischang, B.J. Frey and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on Information Theory,* vol. 47, pp. 458-519, Feb. 2001.

[55] J. Leon, "A probabilistic algorithm for computing minimum weights of large error-correcting codes,", *IEEE Transactions on Information Theory,*, vol. 34, 1988, pp. 1354-1359.

[56] S. Leveiller, J. Boutros, P. Guillot and G. Zémor, "Cryptanalysis of Nonlinear Filter Generators with $\{0,1\}$-Metric Viterbi Decoding," *IMA Int. Conf. 2001* pp.402-414.

[57] S. Leveiller, G. Zémor, P. Guillot and J. Boutros, "A new cryptanalytic attack on PN-generators filtered by a Boolean function," *Selected Areas on Cryptography-2002,* in Lecture Notes in Computer Science pp. 232-249.

[58] S. Leveiller, "Cryptanalysis of nonlinear filter registers through probability-matching", submitted to *IEEE Transactions on Information Theory*, February 2003.

[59] R. Lidl and H. Niederreiter : *Finite Fields*. Encyclopedia of Math. and Its Appl., Vol. 20, Addison-Wesley Publ. Co., Reading, Mass., 1983 ; reprint, Cambridge Univ. Press, Cambridge, 1997.

[60] R. Lucas, M. Fossorier, Y. Kou and S. Lin, "Iterative Decoding of One-Step Majority Logic Decodable Codes Based on Belief Propagation," *IEEE Transactions on Communications,* vol. COM-48, pp. 931-937, June 2000.

[61] F.J. MacWilliams, N.J.A. Sloane : *The Theory of Error-Correcting Codes*. North-Holland Mathematical Library, 1977.

[62] D.J.C MacKay and R.M. Neal, "Near Shannon limit performance of low-density parity-check codes," *Electronic Letters,* vol. 32, pp. 1645-1646, Aug. 1996. Reprinted in *Electronic Letters,* vol. 33, pp. 457-458, Mar. 1997.

[63] D.J.C MacKay, "Good Error-Correcting Codes based on Very Sparse Matrices," *IEEE Transactions on Information Theory,* vol. 45, March 1999.

[64] M. Matsui, "Linear cryptanalysis method for DES cipher," *Advances in Cryptology - EUROCRYPT'93,* Lecture Notes in Computer Science, vol. 765, pp. 386-397, Springer Verlag, 1993.

[65] R.J. McEliece, D.J.C. MacKay and J.-F Chenf, "Turbo decoding as an instance of Pearl's 'belief propagation' algorithm," *IEEE J. Select Areas Commun.,* vol. 16, pp. 140-152, Feb. 1998.

[66] W. Meier and O. Staffelbach, "Fast correlation attack on certain stream ciphers," *Journal of Cryptology,* pp. 159-176, 1989.

[67] W. Meier and O. Staffelbach, "Nonlinearity Criteria for Cryptographic Functions," *Advances in Cryptology - EUROCRYPT'89,* Lecture Notes in Computer Science, vol. 434, pp. 549-562, Springer Verlag, 1990.

[68] M. Mihaljevič, J. Golic, "A comparison of cryptanalysis principles based on iterative error-correction," *Advances in Cryptology -EUROCYPT'91,* Lecture Notes in Computer Science, Springer-Verlag, 1991, vol. 547, pp. 159-176.

[69] M. Mihaljevič, J. Golic, "Convergence of a Bayesian iterative error-correction procedure on a noisy shift register sequence," Lecture Notes in Computer Science, 1993 vol. 658, pp. 24-137. (reprinted in Lecture Notes in Computer Science, vol. 1440, 1999).

[70] M.J. Mihaljevič and J. Golic, "A method for convergence analysis of iterative probabilistic decoding", *IEEE Transactions on Information Theory* vol. 46, Sept. 2000, pp. 2206-2211.

[71] M.J. Mihaljevič, M.P.C. Fossorier and H. Imai, "On decoding techniques for cryptanalysis of certain encryption algorithms," *IEICE Transactions on Fundamentals,* vol. E84-A, pp. 919-930, April 2001.

[72] M.J. Mihaljevič, M.P.C. Fossorier and H. Imai, "A low-complexity and high-performance algorithm for the fast correlation attack," *Fast Software Encryption 2000,* Lecture Notes in Computer Science, vol. 1978, pp. 196-212, Springer Verlag, 2001.

[73] M. Mihaljevič, M. Fossorier and H. Imai, "An Algorithm for Cryptanalysis of Certain Keystream Generators Suitable for High-Speed Software and Hardware Implementations," *IEICE Transactions on Fundamentals,* vol. E84-A, pp. 311-318, January 2001.

[74] M.J. Mihaljevič, M.P.C. Fossorier and H. Imai, "Fast correlation attack algorithm with the list decoding and an application," *Fast Software Encryption 2001,* Lecture Notes in Computer Science, vol. 2355, pp. 196-210, Springer Verlag, 2002.

[75] E. Pasalic, "On Boolean Functions in Symmetric-Key Ciphers", Ph. D. Thesis, Lund University, Feb. 2003.

[76] W. Penzhorn, "Correlation Attacks on Stream Ciphers : Computing low Weight Parity Checks based on Error-Correcting Codes," *FSE 96,* Lecture Notes in Computer Science 1039, Springer Verlag, 1996.

[77] O. Pothier, "Compound codes based on graphs and their iterative decoding," Ph. D. Thesis, Paris, 2000.

[78] T. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Transactions on Information Theory,* vol. 47, pp. 599-618, Feb.2001.

[79] T. Richardson, A. Shokrollahi and R. Urbanke, "Design of provably good low-density parity-check codes," *IEEE Transactions on Information Theory,* vol. 47, pp. 619-637, Feb.2001.

[80] R.A. Rueppel : *Analysis and Design of Stream Ciphers*. Berlin, Springer-Verlag, 1986.

[81] P. Sarkar and S. Maitra, "Construction of nonlinear Boolean functions with important cryptographic properties,"*Advances in Cryptology - EUROCRYPT 2000,* Lecture Notes in Computer Science, vol. 1807, pp. 485-506, Springer Verlag, 2000.

[82] W.G. Schneeweiss : *Boolean Functions with Engineering Applications and Computer Programs*. Springer 1989.

[83] B. Schneier : *Applied Cryptography*. Wiley 1996, Second Edition.

[84] J. Seberry, X. M. Zhang and Y. Zheng, "On constructions and nonlinearity of correlation immune functions," *Advances in Cryptology – EuroCrypt'93,* Lecture Notes in Computer Science, vol. 765, Springer-Verlag, Berlin, 1994, pp. 181-199,.

[85] J. Seberry, X. M. Zhang and Y. Zheng, "Nonlinearly balanced Boolean functions and their propagation characteristics," *Advances in Cryptology – Crypto'93,* Lecture Notes in Computer Science, vol. 773, Springer-Verlag, Berlin, 1994, pp. 49-60.

[86] J. Seberry, X. M. Zhang and Y. Zheng, "Relationships among nonlinearity criteria," *Advances in Cryptology – EuroCrypt'95,* Lecture Notes in Computer Science, vol. 950, Springer-Verlag, 1995, pp. 376-388.

[87] T. Siegenthaler : "Correlation-Immunity of Nonlinear Combining Functions for Cryptographic Applications," *IEEE Transactions on Information Theory,* vol. IT-30, pp. 776-780, 1984.

[88] T. Siegenthaler, "Decrypting a class of stream ciphers using ciphertext only," *IEEE Transactions on Computers,* vol. C-34,pp. 81-85, 1985.

[89] C. Shannon, "A mathematical theory of communication," *Bell system Journal,* vol. 27 pp. 379-423 part I, et pp. 623-656 part II, 1948.

[90] J. Stern, "A method for finding codewords of small weight," *Coding Theory and Applications*, Springer-Verlag, 1989, pp. 106-113.

[91] M. Sudan, "Decoding of Reed Solomon codes beyond error-correction bound," in *Journal of Complexity,* 13(1) : pp. 180-193, 1997.

[92] R.M. Tanner, "A recursive approach to low complexity codes," *IEEE Transactions on Information Theory,* vol. IT-27, pp. 533-547, Sept. 1981.

[93] Yuriy Tarannikov, Peter Korolev and Anton Botev, "Autocorrelation Coefficients and Correlation Immunity of Boolean Functions," *ASIACRYPT 2001,* pp. 460-479.

[94] Yuriy Tarannikov, "New Constructions of Resilient Boolean Functions with Maximal Nonlinearity," *FSE 2001* Lecture Notes in Computer Science, pp. 66-77.

[95] A. Vardy, "Trellis structure of codes," in *Handbook of Coding Theory,* V. Pless and W.C. Huffman, Eds. Amsterdam, The Netherlands : Elsevier 1998.

[96] N. Wiberg, "Codes and decoding on general graphs" Ph.D. dissertation, Univ. Linköping, Linköping, Sweden, 1996.

[97] N. Wiberg, H.-A. Loeliger and R. Kötter, "Codes and iterative decoding on general graphs," *Euro. Trans. Telecomm.,* vol. 6, pp. 513-525, Sept./Oct 1995.

[98] G.-Z Xiao and J-L. Massey, "A spectral characterization of correlation-immune combining functions," *IEEE Transactions on Information Theory,* vol. IT-34 no 3, pp 569-571, 1988.

[99] G. Zémor : *Cours de cryptographie*. Cassini 01/2001.

[100] Y. Zheng and X.-M. Zhang : "Plateaued Functions," *2nd International Conference on Information and Communications Security*, ICISC'99, Lecture Notes in Computer Science, vol. 1758, pp. 284-300, Springer-Verlag, 1999.

[101] Y. Zheng and X. M. Zhang, "Improving upper bound on nonlinearity of high order correlation immune functions," *Selected Area on Cryptography-2000,* in Lecture Notes in Computer Science, vol. 2012, pp. 262-274, Springer Verlag, 2001.