Thèse

présentée à
l´École Polytechnique

pour obtenir le titre de
Docteur de l´École Polytechnique en Informatique

par
Matthieu Finiasz

sous la direction de
Nicolas Sendrier

Nouvelles constructions utilisant des codes correcteurs d'erreurs en cryptographie à clef publique

---

Soutenue le 1er octobre 2004

Rapporteurs :	Alexander Barg
	Thierry Berger
Directeur :	Nicolas Sendrier
Jury :	François Morain
	Marc Girault
	Gilles Villard

Table des matières

Figures et tableaux

Remerciements

Je tiens à remercier toutes les personnes qui ont eu l'occasion de m'aider au cours de ma thèse et des années qui ont précédé. D'une part ceux qui ont participé à l'amélioration du contenu scientifique de ce document, mais aussi tous les autres qui, même sans pouvoir apporter de compétences techniques, m'ont soutenu et rendu ces quelques années très agréables.

Je remercierai donc tout d'abord Nicolas Sendrier, le directeur de thèse que tout thèsard rêverait d'avoir : toujours disponible quand il le faut et jamais oppressant si ce n'est pas nécessaire.

Daniel Augot, un coauteur de choix, débordant d'idées (bonnes ou moins bonnes) et prêt à les partager avec qui voudra.

Pascal Charpin, Anne Canteaut, Jean-Pierre Tillich, mais aussi Christelle, Cedric, Emmanuel, Gabien, Marion, Harold, Thomas, Mathieu, Ludovic, Cédric, Stéphane et tous les autres membres du projet CODES, pour le temps qu'ils ont passé à me faire découvrir les mondes du codage et de la cryptographie, mais aussi pour l'ambiance générale qu'ils contribuent à entretenir et qui rend plaisant le travail au projet.

Je tiens aussi à remercier Alexander Barg et Thierry Berger pour avoir accepté d'être rapporteurs pour ma thèse et François Morain, Marc Girault et Gilles Villard de faire partie de mon jury.

Enfin, je dis un grand merci à toutes celles et ceux qui m'ont aidé en dehors du travail : Thuriane qui m'a fait découvrir la Bretagne (mais pas seulement) ; Erwan qui m'a fait découvrir les profondeurs abyssales du monde des nuits parisiennes ; Vincent, Denis, François, Marianne, Béné et Christelle pour les restos, le roller, les belotes, le ski, Barcelone... et tout ce qui m'a fait regarder ailleurs que vers un écran ; et aussi mon frère Arnaud et ma sœur Catherine, maman, et mes grand-parents de Carvin et Montbonnot pour m'avoir aidé à arriver où je suis.

La dernière phrase de ces remerciements sera à la mémoire de mon père qui, j'en suis certain, aurait été très heureux de pouvoir assister à ma soutenance.

Introduction

Les résultats présentés dans ce manuscrit sont le fruit de mes quatre années de thèse, de septembre 2000 à 2004, au sein du projet CODES à l'INRIA Rocquencourt. Ces travaux ont fait l'objet de plusieurs publications [6, 22, 24, 36] et d'un rapport de recherche [23]. Les travaux les plus récents, présentés dans le dernier chapitre de ce document, suivent encore actuellement le processus d'acceptation et ne sont pas officiellement publiés.

Comme l'indique son titre, cette thèse traite de l'utilisation des codes correcteurs d'erreurs en cryptographie, et plus particulièrement de trois nouvelles constructions qui ont abouti d'une part au premier schéma de signature basé sur le cryptosystème de McEliece, d'autre part à un nouveau cryptosystème à clef publique faisant reposer sa sécurité sur le problème difficile de reconstruction polynomiale et enfin à une nouvelle famille de fonctions de hachage cryptographiques, à la fois rapides et offrant de bons arguments de sécurité.

Ce document est structuré en trois parties. Dans la première, je commence par introduire quelques notions de base de la théorie des codes correcteurs d'erreurs, essentielles à la compréhension du reste du manuscrit. J'explique ensuite les raisons qui font de la théorie des codes un bon outil pour la conception de primitives cryptographiques et présente quelques uns des exemples les plus célèbres de ses applications.

La deuxième partie est consacrée à mes travaux concernant les codes de Goppa. Elle commence donc par quelques rappels des propriétés importantes des codes de Goppa, et en particulier des codes de Goppa binaires. J'expose ensuite les résultats que j'ai obtenus en utilisant des méthodes expérimentales pour déterminer le nombre de mots de poids minimum dans un code de Goppa binaire [36]. J'étends ensuite cette technique à l'évaluation du nombre d'autres mots de poids voisins du poids minimum. Ces résultats n'ont pas de rapports directs avec la cryptographie, mais être capable de trouver un mot de poids minimum dans un code étant un problème difficile, il est certainement possible d'utiliser des techniques similaires à celles que j'ai employées pour des applications cryptographiques. Enfin, je présente le schéma de signature utilisant les codes mis au point en collaboration avec N. Sendrier et N. Courtois [22]. Cette construction est à ce jour la seule dont la sécurité repose exactement sur les mêmes problèmes que le cryptosystème de McEliece. Elle offre en plus l'avantage de fournir des signatures de petite taille que l'on peut même raccourcir afin d'obtenir les plus courtes signatures connues.

La troisième et dernière partie de cette thèse est consacrée au problème de

syndrome decoding, le plus célèbre problème NP-complet issu de la théorie des codes correcteurs d'erreurs. Après un chapitre introductif décrivant ce problème et certaines de ses formes dérivées et présentant les meilleurs algorithmes connus pour sa résolution, je décris deux nouvelles constructions faisant reposer leur sécurité en partie sur ce problème. La première est un nouveau cryptosystème à clef publique mis au point avec D. Augot [6] et exploitant la difficulté de certaines instances du problème de *polynomial reconstruction*. Malheureusement cette construction présentait une faiblesse et j'explique ensuite comment ce système peut être cassé. Enfin, dans le dernier chapitre de cette thèse, j'expose mes travaux les plus récents, concernant une nouvelle famille de fonctions de hachage cryptographiques rapides et sûres. Ce travail, effectué en commun avec D. Augot et N. Sendrier [7], est le premier exemple de fonction de hachage qui allie à la rapidité une réduction de sécurité, à une instance d'un problème difficile, ici, *syndrome decoding*. Cette construction n'a toutefois pas vocation à remplacer les constructions existantes qui offrent en général une meilleure sécurité et une meilleure vitesse, et auxquelles il manque juste une preuve de cette sécurité.

Première partie

Préliminaires

Chapitre 1

Codes correcteurs d'erreurs

Les codes correcteurs d'erreurs sont un outil visant à améliorer la fiabilité des transmissions sur un canal bruité. La méthode qu'ils utilisent consiste à envoyer sur le canal plus de données que la quantité d'information à transmettre. Une redondance est ainsi introduite. Si cette redondance est structurée de manière exploitable, il est alors possible de corriger d'éventuelles erreurs introduites par le canal. On peut alors, malgré le bruit, retrouver l'intégralité des informations transmises au départ.

Une grande famille de codes correcteurs d'erreurs est constituée des codes par blocs. Pour ces codes l'information est d'abord coupée en blocs de taille constante et chaque bloc est transmis indépendamment des autres, avec une redondance qui lui est propre. La plus grande sous-famille de ces codes rassemble ce que l'on appelle les codes linéaires.

Dans un code linéaire, les messages que l'on veut coder sont lus sous la forme d'un k-uplet d'éléments d'un corps fini $\mathbb{K}$. Cet élément de $\mathbb{K}^k$ est ensuite transformé en élément de $\mathbb{K}^n$ par une application linéaire. La longueur n est choisie plus grande que la dimension k et c'est ainsi que la redondance est ajoutée. Ce que l'on appellera *code* est en fait le sous-espace vectoriel $\mathcal{C}$ de dimension k de $\mathbb{K}^n$ correspondant à l'espace image de l'application linéaire. La *matrice génératrice* $\mathcal{G}$ du code est la matrice associée à cette application linéaire. On appellera *taux de transmission* de ce code le rapport $\frac{k}{n}$ entre la longueur du message transmis et la quantité d'information utile qu'il contient.

Dans ce manuscrit, le terme code désignera, sauf mention explicite, un code linéaire sur le corps $\mathbb{F}_2$ ou une de ses extensions de la forme $\mathbb{F}_{2^m}$.

1-1 Que signifie décoder ?

Décoder dans un code $\mathcal{C}$ désigne l'action d'associer un mot du code (un élément du sous-espace vectoriel) à un mot de l'espace vectoriel. On cherche le plus souvent à décoder en associant à un mot le mot de code duquel il est le plus proche. Cependant il faut d'abord décider du sens que l'on veut donner à l'expression « le plus proche ».

Dans le cas du décodage d'un message transmis le long d'un canal bruité, on s'intéresse essentiellement au *décodage à vraisemblance maximale*. Cela consiste à toujours associer le mot de code qui a la plus grande probabilité d'avoir donné ce mot en étant transmis sur le canal. Bien évidemment, décoder n'aura pas le même sens selon la nature du canal.

Le modèle de canal le plus souvent utilisé est le *canal binaire symétrique*. Un tel canal ne possède pas de mémoire et les données qui y sont transmises sont binaires. Il est caractérisé par sa *probabilité de transition* p qui est la probabilité qu'un bit émis (0 ou 1) soit changé en son opposé. Pour ce canal, le décodage à vraisemblance maximale consiste à trouver le mot de code qui a le plus de coordonnées en commun avec le mot reçu. On introduit pour cela la notion de *poids de Hamming* d'un élément de $\mathbb{K}^n$: c'est le nombre de coordonnées non nulles de cet élément. Ce poids est donc compris entre 0 et n et définit une norme au sens mathématique du terme. La *distance de Hamming* est la distance déduite de cette norme : la distance entre deux éléments est le nombre de coordonnées où ils diffèrent.

On cherche donc à trouver le mot de code le plus proche pour la distance de Hamming. Cependant, effectuer un *décodage complet* (c'est-à-dire savoir décoder n'importe quel mot de l'espace) à vraisemblance maximale, s'avère en général être un problème très difficile.

On s'intéresse donc aussi à d'autres formes de décodage, moins fortes : par exemple le *décodage borné*. Cela consiste à renvoyer un mot de code quelconque se trouvant à une distance inférieure à une borne ρ fixée. Évidemment, selon la valeur de ρ certains mots de l'espace peuvent s'avérer non décodables. On appellera *rayon de recouvrement* la valeur minimale de ρ pour laquelle l'espace est entièrement décodable : si on construit une « boule » de rayon ρ (pour la distance de Hamming) autour de chaque mot de code, c'est la valeur minimale pour laquelle les boules recouvrent tout l'espace. Ainsi, en effectuant un décodage borné jusqu'au rayon de recouvrement on a aussi un décodage complet, mais pas toujours à vraisemblance maximale. De plus, calculer le rayon de recouvrement d'un code est encore une fois un problème difficile. La *sphere packing bound* qui est la plus petite valeur de ρ pour laquelle le volume total des boules (le volume d'une boule multiplié par le nombre de mots dans le code) excède le volume de l'espace, nous fournit un minimum, mais il est rare que les deux valeurs soient égales.

De la même façon, on introduit la *distance minimale* d'un code. C'est la plus petite distance d séparant deux mots du code. Comme le code est linéaire c'est aussi le poids minimum d'un mot de code non nul. On écrira $[n, k, d]$ les paramètres d'un code de longueur n, dimension k et distance minimale d. En effectuant un décodage borné jusqu'à la distance $\left\lfloor \frac{d-1}{2} \right\rfloor$ on a un décodage unique qui se trouve donc être un décodage à vraisemblance maximale.

Pour un *code parfait* le rayon de recouvrement vaut exactement $(d-1)/2$ et le décodage borné devient alors aussi un décodage complet à vraisemblance maximale, malheureusement on ne connaît que peu de tels codes.

On cherche donc à construire des codes possédant une bonne *distance construite* (une borne inférieure sur la distance minimale du code, liée à la structure de ce code) et tels qu'il existe un algorithme efficace (polynomial en général) permettant de décoder de l'une des façons précédentes. La plupart du temps ces algorithmes permettront de décoder jusqu'à la moitié de la distance construite du code.

1-2 Quelques exemples

Dans cette section, je présente quelques exemples de codes par blocs les plus célèbres et les algorithmes utilisés pour les décoder.

1-2.1 Le code à répétitions

Ce code est la forme la plus simple de code par blocs : chaque bit à transmettre est simplement répété d fois dans le message transmis. Ainsi pour $d = 3$ sur $\mathbb{F}_2$ le message 1101 devient 111111000111. Ce code peut aussi être défini sur n'importe quel autre corps et sa matrice génératrice est toujours la matrice $1 \times d$ remplie de 1. Cette construction donne une distance minimale de d mais une dimension de 1 pour une longueur d aussi. On construit donc des codes de la forme $[d, 1, d]$.

Le décodage est ensuite très simple puisqu'il suffit de garder le bit majoritaire dans chaque bloc. La capacité de correction ainsi obtenue est donc de $\left\lfloor \frac{d-1}{2} \right\rfloor$. Le principal problème est qu'avec ce code le taux de transmission est de $\frac{1}{d}$ ce qui est très peu. On peut toutefois noter que ce code est parfait quand on choisit d impair.

1-2.2 Le code de Hamming

Ce code est lui aussi très simple mais a un bien meilleur taux de transmission. Pour le définir il est nécessaire d'introduire la notion de *matrice de parité* : c'est une matrice $\mathcal{H}$ de taille $(n - k) \times n$ qui a pour noyau le code $\mathcal{C}$ tout entier. Ainsi si $c \in \mathcal{C}$, on aura toujours $\mathcal{H}c^T = 0$, et uniquement dans ce cas là[1]. Remarquons qu'un même code $\mathcal{C}$ peut avoir plusieurs matrices de parité différentes, de même qu'il peut avoir plusieurs matrices génératrices différentes. On appellera *syndrome* d'un mot c l'élément $\mathcal{S}$ obtenu en calculant $\mathcal{S} = \mathcal{H}c$. Ainsi les mots de $\mathcal{C}$ seront tous les éléments de $\mathbb{F}_q^n$ ayant un syndrome nul.

Le code de Hamming est donc un code binaire défini par sa matrice de parité plutôt que par sa matrice génératrice. C'est la matrice de dimension $r \times (2^r - 1)$ qui contient toutes les colonnes non nulles distinctes que l'on peut écrire sur r bits. Ainsi pour $r = 3$ on a :

$$\mathcal{H} = \begin{pmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Le code de Hamming est l'ensemble des mots de longueur $2^r - 1$ dans le noyau de $\mathcal{H}$. C'est donc un espace de dimension $2^r - 1 - r$. De plus, la distance minimale de ce code est 3 car il n'existe aucun mot de code de poids 1 ou 2 puisque cela signifierait qu'il y a une colonne nulle ou deux colonnes égales dans $\mathcal{H}$. On a donc un code $[2^r - 1, 2^r - 1 - r, 3]$ dans lequel on doit donc pouvoir décoder une erreur. Effectivement, on peut facilement décoder une erreur seule en utilisant la matrice de parité : si on reçoit un mot bruité $c' = c + e$ où c est un mot du code et e une erreur de poids 1 on calcule $\mathcal{S} = \mathcal{H}c' = \mathcal{H}c + \mathcal{H}e = \mathcal{H}e$ puisque c est dans le code et donc dans le noyau de $\mathcal{H}$. L'erreur e étant de poids

[1] Notons que pour plus de simplicité, dans la suite de cette thèse, je noterai simplement $\mathcal{H}c = 0$, sans transposition, quand cela n'introduit pas d'ambiguïté.

1, $\mathcal{S}$ est donc égal à la colonne de $\mathcal{H}$ où se situe l'erreur et comme toutes les colonnes sont distinctes on peut retrouver e et donc c.

Cette technique de décodage passant par le calcul d'un syndrome est la méthode utilisée pour décoder quasiment tous les codes par blocs et est un moyen facile d'obtenir une information ne dépendant que de e et pas du message transmis contenu dans c.

Pour $r = 1$ le code de Hamming n'est pas intéressant puisqu'il ne contient que le mot nul et pour $r = 2$ c'est le code à répétition de longueur 3. En revanche, pour n'importe quelle autre valeur de r c'est un code parfait.

1-2.3 Les codes de Reed-Solomon

Les codes de Reed-Solomon ont été développés par Reed et Solomon dans les années 50 [81] mais avaient déjà été construits par Bush [15] un peu avant, dans un autre contexte. Ces codes sont certainement les codes par blocs les plus utilisés pour la correction d'erreurs en étant présents dans les CD, les DVD et la plupart des support de données numériques. Ils sont très utilisés car ils sont extrêmaux du point de vue de la capacité de correction.

a) Construction

La façon la plus simple de voir ces codes est en tant que *code d'évaluation* : chaque élément du support du code est associé à un élément du corps $\mathbb{F}_q$ sur lequel est défini le code et chaque mot de code est l'évaluation d'une fonction $f \in \mathcal{F}$ sur le support. Pour les Reed-Solomon on prend $\mathbb{F}_q = \mathbb{F}_{2^m}$ et $\mathcal{F}$ l'ensemble des polynômes de degré strictement inférieur à k sur $\mathbb{F}_{2^m}$. Ainsi on a bien un code linéaire de longueur $n \leq 2^m$ et dimension k. Une matrice génératrice du code peut alors s'écrire :

$$\mathcal{G} = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & & \alpha_n \\ \vdots & & & \vdots \\ \alpha_1^{k-1} & \alpha_2^{k-1} & \cdots & \alpha_n^{k-1} \end{pmatrix}$$

Par sa nature ce code a une distance minimale d'au moins $n-k+1$ car deux polynômes de degré $< k$ distincts ne peuvent pas être égaux en plus de $k-1$ positions distinctes. Cette distance est même exactement égale à $n-k+1$ puisque l'évaluation d'un polynôme de la forme $\prod_{i=1}^{k-1}(X - \alpha_i)$ est de poids $n-k+1$. On a donc des codes sur $\mathbb{F}_{2^m}$ de la forme $[n, k, n-k+1]$ qui peuvent donc avoir à la fois un bon taux de transmission et une bonne capacité de correction.

Notons que cette distance minimale est la meilleure que l'on puisse atteindre avec ces paramètres : une distance minimale supérieure entrerait en contradiction avec la dimension du code.

Ces codes ont aussi l'avantage de pouvoir corriger des rafales d'erreurs du fait de leur structure sur $\mathbb{F}_{2^m}$. En effet, une fois écrits en binaire, si une erreur atteint plusieurs bits à la suite il y a de bonnes chances pour que cela n'affecte qu'un seul bloc de m bits et donc ne crée au final qu'une seule erreur.

b) Décodage

Plusieurs algorithmes polynomiaux permettent de décoder efficacement $\frac{n-k}{2}$ erreurs dans un code de Reed-Solomon. Celui de Berlekamp-Massey [10, 64] est

le plus connu et le plus utilisé, mais l'algorithme le plus simple à expliquer est, selon moi, celui de Berlekamp-Welch [13].

On suppose que l'on a reçu un mot $c' = c + e$ où e est une erreur de poids w et c un mot de code, évaluation d'un polynôme $\mathcal{P}_c$ de degré $< k$. On appelle *polynôme localisateur* de e le polynôme unitaire $\mathcal{L}_e$ qui s'annule en tous les éléments du support où e est non nul. Le degré de $\mathcal{L}_e$ est donc w. Puisque e n'est pas connu ce polynôme est aussi inconnu. Si on multiplie le mot reçu par l'évaluation de ce polynôme on trouve :

$$\forall i \in [1; n] \quad c'_i \times \mathcal{L}_e(\alpha_i) = (\mathcal{P}_c(\alpha_i) + e_i) \times \mathcal{L}_e(\alpha_i) = \mathcal{P}_c(\alpha_i) \times \mathcal{L}_e(\alpha_i). \quad (1\text{-}1)$$

Le polynôme $\mathcal{P}_c$ de degré $< k$ est lui aussi inconnu (c'est ce que l'on cherche à trouver), et on linéarise le système en introduisant le polynôme $\mathcal{N} = \mathcal{P}_c \times \mathcal{L}_e$ qui est donc de degré au plus $k - 1 + w$. On obtient alors le système linéaire suivant :

$$\forall i \in [1; n] \quad c'_i \times \mathcal{L}_e(\alpha_i) = \mathcal{N}(\alpha_i). \quad (1\text{-}2)$$

Il est composé de n équations en $k+2w$ inconnues (w inconnues pour $\mathcal{L}_e$ qui est unitaire de degré w et $k + w$ pour $\mathcal{N}$). De plus, toute solution du système 1-1 donne aussi une solution pour ce système. Il suffit donc qu'il y ait un nombre fini de solutions à 1-2 pour pouvoir retrouver une solution. Ce sera le cas si $n \geq k + 2w$ et donc si $w \leq \frac{n-k}{2}$. On peut donc facilement corriger jusqu'à la moitié de la distance construite des Reed-Solomon. C'est le maximum que l'on puisse faire si on veut garder un décodage unique.

Au-delà de cette borne de $\frac{n-k}{2}$ erreurs on peut encore décoder en temps polynomial en utilisant l'algorithme de Guruswami-Sudan [47, 92]. Cet algorithme permet d'effectuer du *décodage par liste*, c'est-à-dire qu'il renvoie la liste de tous les mots plus proches que la distance fixée. Avec, on peut corriger jusqu'à $n - \sqrt{nk}$ erreurs, et même s'il peut renvoyer une liste de solutions, dans la pratique la liste ne contiendra en général qu'une seule solution.

Quand on essaye de corriger plus d'erreurs, il n'existe pas d'algorithme polynomial pour décoder : on arrive dans les instances difficiles du problème de *polynomial reconstruction* que nous verrons plus en détail en section 7-1 page 88.

1-2.4 Les codes de Goppa

Ces codes seront étudiés en détail dans le chapitre 3 page 33. Ils sont célèbres car ce sont les codes utilisés dans les cryptosystèmes de McEliece et Niederreiter que nous verrons sections 2-2 page 25 et qu'ils sont aussi de très bons codes binaires.

1-2.5 Les codes cycliques, BCH, de Reed-Muller, de Golay...

Il existe beaucoup d'autres classes de codes par blocs, toutes présentant leurs intérêts, soit du point de vue de la capacité de correction, soit de la simplicité du codage ou du décodage... Je ne fais que mentionner leur existence car leur connaissance n'est pas nécessaire à la compréhension de cette thèse. Ils n'est en revanche pas exclu que des applications cryptographiques de la théorie algébrique des codes puissent passer par l'utilisation de ces codes : c'est d'ailleurs déjà le cas pour certains problèmes (recherche de fonctions booléennes, *traitor tracing*...) que je n'exposerai pas dans ce document.

1-2.6 Les codes aléatoires

Cette dernière catégorie n'est pas vraiment une famille de codes puisqu'il s'agit en fait de tous les codes construits sans structures particulières. Pour obtenir un code aléatoire il suffit de tirer une matrice génératrice aléatoire et de chercher son image. Bien sûr, une fois choisi, le code n'est plus aléatoire, mais de façon générale, un code construit de cette façon aura de bonnes propriétés en moyenne : il a en général une bonne distance minimale.

Malheureusement pour un tel code il n'existe pas d'algorithme de décodage polynomial. Comme nous le verrons plus en détail pour les codes binaires dans le chapitre 6 page 77, décoder correspond au problème NP-complet de *syndrome decoding* qui est à la base de la plupart des applications cryptographiques des codes.

1-3 Remarques

Afin d'être plus complet, il pourrait être judicieux de parler un peu des autres variétés de codes utilisées de nos jours. En effet les codes par blocs sont faciles à analyser et sont, historiquement, les codes les plus utilisés. Cependant, ils tendent à être souvent remplacés par des codes comme les turbo codes, LDPC ou autre codes convolutifs ayant de meilleures capacités de correction ou des algorithmes de décodage plus efficaces. Ils sont en général conçus pour effectuer du *décodage souple*, c'est-à-dire quand on associe à chaque bit reçu une probabilité d'erreur propre.

Ces codes sont cependant beaucoup plus difficiles à analyser, et il est même en général assez difficile de connaître leur exacte capacité de correction. Ils ont pourtant quand même déjà des applications en cryptographie, pour des attaques sur des chiffrements à flots [52, 53] par exemple. Il faudra par contre certainement encore attendre un peu avant de voir apparaître les premiers cryptosystèmes les utilisant.

Chapitre 2

Applications à la cryptographie

En 1976, avec l'invention du premier cryptosystème à clef publique par Diffie et Hellman [28], est née une nouvelle branche de la cryptographie moderne. L'idée nouvelle était de faire reposer la sécurité d'un système non pas sur la connaissance d'une clef (partagée secrètement par les utilisateurs), mais sur la difficulté d'inverser une *fonction à sens unique avec trappe*. Une *fonction à sens unique* est simplement une fonction calculatoirement difficile à inverser, une *trappe* est un algorithme secret rendant facile cette inversion. Ainsi la trappe n'est connue que d'une personne, seule à pouvoir déchiffrer les messages créés en utilisant la fonction à sens unique qui est elle publique.

Comme on le constate, par rapport à la cryptographie symétrique pour laquelle il suffit d'avoir une fonction à sens unique secrète (ou définie à partir d'une clef secrète), il faut en plus qu'il soit difficile de retrouver la trappe à partir de la fonction (ou de la clef qu'elle utilise). Il fallait donc introduire de nouveaux problèmes difficiles, mais suffisamment structurés pour pouvoir y cacher une trappe. Diffie et Hellman ont pensé au problème de logarithme discret, puis pour RSA [84] le problème de la factorisation de grands entiers, plus récemment les logarithmes sur des courbes elliptiques...

Dès 1978, McEliece [66] a imaginé le premier et le plus célèbre des cryptosystèmes à clef publique utilisant des codes correcteurs d'erreurs. Comme nous allons le voir dans ce chapitre, la théorie des codes contient elle aussi de multiples problèmes bien structurés et difficiles à résoudre, plus ou moins biens adaptés pour une utilisation en cryptographie. Suivra un petit inventaire de quelques unes des applications les plus connues faisant appel à ces problèmes.

Notons quand même que les codes ont aussi beaucoup d'applications dans d'autres domaines de la cryptographie y compris quand ce ne sont pas les problèmes difficiles qu'ils offrent qui sont utilisés. Je n'en parlerai cependant pas dans ce chapitre car cela ne correspond pas réellement à l'utilisation faite des codes dans le reste de cette thèse.

2-1 Quelques problèmes difficiles liés aux codes

Voici une liste de quelques uns des problèmes difficiles liés à la théorie des codes. Bien sûr cet inventaire n'est pas exhaustif et ce sont souvent des problèmes dérivés de ces problèmes très généraux qui sont utilisés en pratique dans les cryptosystèmes.

Il est intéressant de remarquer que ces problèmes, en plus d'être calculatoirement difficiles sur une machine de Turing classique, restent calculatoirement difficiles quand on essaye de les résoudre à l'aide d'une machine quantique. Contrairement aux problèmes de factorisation et de logarithmes discrets, aucun algorithme actuellement connu ne permet d'attaquer ces problèmes liés aux codes.

2-1.1 Construction de code

On peut énoncer le problème de la façon suivante :

Construction de code :
Données : le corps $\mathbb{F}_q$ et des paramètres n, k et d.
Problème : construire un code $[n, k, d]$ sur $\mathbb{F}_q$.

De façon générale ce problème est difficile et n'a pas de solution polynomiale connue. De même, si on le restreint à un corps particulier il reste difficile. Toutefois, pour certains jeux de paramètres on a des solutions constructibles en temps polynomial, de même pour d'autres jeux, on sait qu'il n'existe pas de tels codes. La borne de Gilbert-Varshamov permet toutefois de déterminer des paramètres pour lesquels on est sûr qu'un tel code linéaire existe, mais elle n'est pas constructive. Elle s'énonce ainsi :

Théorème : (Gilbert-Varshamov)
Si $\sum_{i=0}^{d-2} \binom{n-1}{i} (q-1)^i < q^{n-k}$ alors il existe un code $[n, k, d]$ sur $\mathbb{F}_q$.

Il est aussi à noter qu'en général, vérifier que le code a bien la distance minimale que l'on annonce est aussi un problème difficile (voir 2-1.3.a page ci-contre).

2-1.2 Décodage

La première chose à faire avant de parler du problème de décodage d'un code est de choisir de quel type de décodage on parle. En effet, comme nous l'avons vu dans le chapitre précédent, les problèmes sont assez différents selon que l'on parle de décodage borné, décodage complet, décodage borné jusqu'au rayon de recouvrement, décodage borné par liste...

Le bon point est que généralement, quelle que soit la formulation que l'on choisit, le problème de décodage sera toujours soit NP-complet soit NP-dur. Nous verrons cela plus en détail en étudiant le problème de *syndrome decoding* (SD) pour le cas général d'un code aléatoire dans le chapitre 6 page 77.

Pour les codes construits de manière à être faciles à décoder, le décodage peut se faire en temps polynomial, mais uniquement jusqu'à la *distance construite* : la borne inférieure sur la distance minimale du code que l'on peut déduire de sa

structure algébrique. Décoder au-delà de cette distance devient alors en général aussi un problème difficile. Nous verrons par exemple cela pour les codes de Reed-Solomon section 7-1 page 88.

2-1.3 Calcul de paramètres

En général, on considère que les paramètres d'un code linéaire sont sa longueur, sa dimension et sa distance minimale. Cela suffit en général à avoir une bonne idée des caractéristiques de ce code. Il existe cependant beaucoup d'autres grandeurs fournissant des informations, parfois utiles, sur le code.

De façon générale, toutes ces grandeurs peuvent être déterminées à l'aide du polynôme énumérateur des poids du code. Malheureusement, calculer ce polynôme nécessite en général (sauf pour certains codes particuliers) d'énumérer tous les mots du code et de regarder leur poids. Il n'est donc pas possible de le calculer explicitement. De plus, à part en certains points particuliers, connaître son évaluation en un point est aussi un problème difficile.

a) Distance minimale

La distance minimale est généralement l'une des données à laquelle on s'intéresse le plus pour un code. En effet, en fonction de sa longueur et de sa dimension, la valeur de la distance minimale du code va permettre de savoir si c'est un « bon » code. On cherche par exemple à construire des codes qui sont sur la borne de Gilbert-Varshamov (voir section 2-1.1 page précédente).

Cependant, calculer cette grandeur pour un code aléatoire est un problème NP-dur [96]. Les familles de codes que nous avons vues section 1-2 page 17 sont construites pour pouvoir décoder un certain nombre d'erreurs de façon unique avec un algorithme efficace. Ceci fixe une borne inférieure pour la distance minimale du code, mais cette valeur n'est pas nécessairement atteinte. Pour certains codes (Hamming, Reed-Solomon...) on sait que c'est le cas et construire des mots ayant ce poids minimum est même facile, en revanche, pour d'autres codes, on ne peut rien prouver. Comme nous le verrons chapitre 4 page 41 pour les codes de Goppa, il est assez difficile de construire des mots de poids minimum, et il n'existe encore aucune preuve que de tels mots existent toujours : il est possible que certains codes de Goppa aient une distance minimale supérieure à leur distance construite.

b) Nombre de mots de poids w

La distance minimale d'un code est la plus petite valeur de w non nulle telle qu'il y ait au moins un mot de poids w dans le code. De façon générale, calculer le nombre de mots de poids w d'un code est difficile.

On sait qu'il y a toujours un mot de poids 0 (car on regarde des codes linéaires), et on sait que jusqu'à la distance minimale (et donc au moins jusqu'à la distance construite aussi) il n'y en a pas. Au-delà on ne sait pas du tout combien il y en a, sauf pour certains codes particuliers.

L'ensemble des nombres de mots de poids w d'un code est ce que l'on appelle sa *distribution des poids*. Pour un code aléatoire, cette distribution des poids est en moyenne une distribution binomiale, c'est-à-dire que le nombre de mots de poids w est en moyenne :

$$\mathcal{N}_w = \frac{\binom{n}{w}}{2^{n-k}}.$$

On peut grâce à cela déterminer la valeur moyenne de la distance minimale d'un code aléatoire qui est, approximativement, la plus petite valeur de w pour laquelle $\mathcal{N}_w \geq 0.5$, et donc en déduire qu'un code aléatoire est en moyenne proche de la borne de Gilbert-Varshamov (voir section 2-1.1 page 22 avec $q = 2$).

c) Énumérateur des poids

Le *polynôme énumérateur des poids* d'un code $\mathcal{C}$ est le polynôme homogène défini par la formule suivante :

$$W_{\mathcal{C}}(x, y) = \sum_{v \in \mathcal{C}} x^{n-\text{poids}(v)} y^{\text{poids}(v)} = \sum_{i=0}^{n} \mathcal{N}_i \; x^{n-i} y^i$$

Avec $\mathcal{N}_i$ le nombre de mots de poids i dans $\mathcal{C}$.

Calculer tous les coefficients de ce polynôme nécessite de regarder les poids de tous les mots du code $\mathcal{C}$ et est donc nécessairement quelque chose de très coûteux. On sait en revanche, à moindre coût, évaluer ce polynôme en certains points particuliers. On le voit facilement sur les exemple suivants :

- $W_{\mathcal{C}}(1, 1) = 2^k$, le nombre total de mots du code
- $W_{\mathcal{C}}(1, -1) = 0$ si le code contient des mots de poids impair, 2^k autrement.

En revanche, le nombre de tels points est fini et ces points sont tous bien connus [97]. Évaluer $W_{\mathcal{C}}$ en un autre point est un problème NP-dur, comme le calcul du polynôme en entier.

Ce polynôme est relativement important en théorie des codes car il contient presque toutes les informations concernant le code et est en même temps invariant par permutation, changement de base... Ainsi deux codes équivalents auront le même polynôme énumérateur des poids : il contient tous les invariants classiques utilisés dans la recherche d'équivalence de codes (voir section 2-1.4). Cependant, deux codes ayant le même énumérateur des poids peuvent quand même ne pas être équivalents.

d) Rayon de recouvrement

Le rayon de recouvrement est une des rares grandeurs d'un code qui ne soit pas contenue dans son polynôme énumérateur des poids. Il est de façon générale très difficile à déterminer de façon sûre car il nécessite de calculer le poids minimal de tous les syndromes possibles.

2-1.4 Équivalence de codes

Étant donnés deux codes $\mathcal{C}_0$ et $\mathcal{C}_1$, ce problème va consister à déterminer s'il existe une permutation[1] permettant de passer de l'un à l'autre.

Un peu comme pour l'isomorphisme de graphes, tester cette équivalence est un problème théoriquement difficile et pour lequel on sait construire des instances difficiles. Cependant, Petrank et Roth [77] ont prouvé qu'il n'est pas NP-complet. De plus, contrairement aux problèmes de décodage, une instance aléatoire de ce problème est en général facile en pratique [85].

Pour prouver que deux codes ne sont pas équivalents il suffit de trouver un *invariant* (une grandeur invariante par permutation et passage à un code équivalent, comme la distance minimale et la distribution de poids) qui diffère

[1] permutation des coordonnées de l'espace dans lequel est défini le code.

entre les deux codes, pour prouver qu'ils sont équivalents il est en revanche nécessaire d'expliciter une transformation permettant de passer d'une matrice génératrice de $\mathcal{C}_0$ à une de $\mathcal{C}_1$. Toutefois, cette opération n'est en général pas très chère.

2-2 Les systèmes connus

Il existe donc une grande variété de problèmes difficiles en théorie des codes. Même si certains semblent mieux adaptés que d'autres, chacun de ces problèmes pourraient servir de base à un cryptosystème : il suffit de pouvoir y cacher une trappe. Cependant, la sécurité de presque tous les cryptosystèmes liés aux codes connus à ce jour, ainsi que celle des trois nouveaux systèmes présentés dans cette thèse (chapitres 5, 7 et 8), repose en majeure partie sur le même problème : *syndrome decoding* (SD). Ce problème sera étudié en détail dans le chapitre 6 page 77 et correspond au problème de décodage d'un code quelconque.

La raison de ce choix vient du fait que le problème SD est à la fois pratique d'utilisation (il est défini dans un contexte très général rendant facile l'ajout d'une trappe) et d'une sécurité remarquable (même les meilleurs algorithmes sont très coûteux, et il est rarement possible de les améliorer, même dans des cas très particuliers, avec des paramètres précis). Cependant, rien n'empêche d'imaginer faire reposer la sécurité d'un système sur la difficulté de trouver un mot de poids donné, ou n'importe quel autre problème.

Par ailleurs, même si le problème SD est pratique du point de vue de la sécurité, il a quand même quelques inconvénients : la manipulation d'une matrice souvent de taille relativement importante va en général faire que tous les systèmes auront besoin, à un moment ou à un autre, de manipuler des objets de grande taille. Ceci fait que dans les systèmes à clef publique utilisant des codes, la taille de la clef sera toujours assez importante et, de façon générale, de tels systèmes ne seront jamais très adaptés à une implantation sur des architectures limitées en mémoire (les cartes à puce par exemple).

2-2.1 Le cryptosystème de McEliece

C'est le plus ancien cryptosystème à clef publique utilisant des codes correcteurs d'erreurs. Il a été imaginé, comme son nom l'indique, par McEliece [66] en 1978, juste après l'invention des premiers cryptosystèmes à clef publique, à peu près en même temps que RSA [84], le système le plus utilisé aujourd'hui.

Comme tous les cryptosystèmes à clef publique, ce système est constitué de 3 algorithmes : la génération de clefs, le chiffrement (utilisant la clef publique) et le déchiffrement (utilisant la clef secrète).

Génération de clef

On commence par générer un code de Goppa corrigeant t erreurs (voir comment chapitre 3 page 33) et sa matrice de parité $\mathcal{G}$ de taille $k \times n$. On va maintenant mélanger cette matrice pour la rendre indistinguable d'une matrice aléatoire : pour cela on a besoin d'une matrice de permutation aléatoire $\mathcal{P}$ de taille $n \times n$ et d'une matrice inversible aléatoire $\mathcal{Q}$ de taille $k \times k$.

La clef publique sera la matrice $\mathcal{G}' = \mathcal{Q} \times \mathcal{G} \times \mathcal{P}$ qui est indistinguable d'une matrice aléatoire.

La clef secrète est composée des trois matrices $\mathcal{Q}$, $\mathcal{P}$ et $\mathcal{G}$ qui permettent de retrouver la structure du code de Goppa et donnent donc accès à l'algorithme de décodage.

Chiffrement

Soit m un message de k bits que l'on veut chiffrer. On ne dispose pour cela que de la clef publique $\mathcal{G}'$.

On commence par calculer le mot de code c de longueur n associé à m :

$$c = m \times \mathcal{G}'$$

Ensuite on génère une erreur aléatoire e de longueur n et poids t. Le chiffré sera simplement le mot de code bruité : $c' = c + e$.

Déchiffrement

Pour déchiffrer en connaissant $\mathcal{P}$, $\mathcal{Q}$ et $\mathcal{G}$ il suffit de calculer :

$$c' \times \mathcal{P}^{-1} = m\mathcal{G}'\mathcal{P}^{-1} + e\mathcal{P}^{-1} = m\mathcal{Q} \times \mathcal{G} + e\mathcal{P}^{-1}.$$

$m\mathcal{Q} \times \mathcal{G}$ est un mot du code de Goppa et $e\mathcal{P}^{-1}$ est une erreur de poids t (car $\mathcal{P}$ est une permutation et conserve donc le poids des mots), donc on peut décoder cette erreur et retrouver le message initial $m\mathcal{Q}$. Il ne reste plus qu'à multiplier par $\mathcal{Q}^{-1}$ pour retrouver le message m et avoir fini de déchiffrer.

Comme nous le verrons plus en détail pour l'algorithme de signature (section 5-3.1 page 60), la sécurité de ce système repose à la fois sur le problème de distinguabilité d'un code de Goppa permuté d'un code aléatoire (section 3-3.3 page 38) et sur le problème de *Goppa parameterized syndrome decoding* décrit section 6-1.2.c page 79. Le premier de ces problèmes correspond à la difficulté de retrouver la clef secrète à partir de la clef privée, et le deuxième au problème du déchiffrement quand on ne connaît pas la clef secrète et que $\mathcal{G}'$ est donc vue comme une matrice aléatoire.

Notons cependant que si l'on chiffre deux fois le même message avec ce système, en y ajoutant deux erreurs e différentes, on pourra facilement retrouver le message m uniquement à partir des $n-2t$ bits identiques dans les deux chiffrés. Il est donc nécessaire d'utiliser ce système dans un protocole qui n'autorise pas à rechiffrer un même message différemment.

Avec cette construction, la clef publique fait nk bits, le taux de transmission est $\frac{k}{n}$ et la taille de bloc k bits. Le chiffrement est essentiellement un produit d'une matrice par un vecteur d'un coût de kn opérations. Le déchiffrement coûte environ $t^2(\log_2 n)^3$ pour la part de décodage et n^2 pour le reste.

Initialement, McEliece avait proposé des paramètres $n = 1024$, $k = 524$ et $t = 50$ qui étaient suffisants pour résister à une attaque par ensemble d'information standard (voir 6-3 page 82). Cependant, avec les progrès faits en matière d'attaque [17], afin d'y résister avec une sécurité de 2^{80} opérations binaires, les paramètres qui semblent les mieux adaptés de nos jours sont : $n = 2048$, $k = 1685$ et $t = 33$. Cela donne un taux de transmission de 0.82, des blocs de 1685 bits et une clef de 3 Mbits, que l'on peut tout de même réduire à 600 kbits en mettant la matrice sous forme systématique (ce qui n'est pas gênant si les messages à chiffrer sont tous équiprobables).

Remarquons enfin qu'une version de ce cryptosystème où la clef publique reste secrète a aussi été imaginée [79, 80], mais elle ne présente cependant que peu d'intérêt dans la pratique...

2-2.2 La variante de Niederreiter

Cette variante du cryptosystème de McEliece a été mise au point par Niederreiter [70] en 1986. Elle est exactement équivalente du point de vue de la sécurité (mis à part qu'avec cette version, chiffrer deux fois le même message ne présente pas de risque) et est un peu plus efficace en temps de calcul. Elle fonctionne comme le chiffrement de McEliece, mais en utilisant la matrice de parité du code et en utilisant l'erreur pour contenir le message.

Génération de clef

Comme pour McEliece, on commence par générer un code de Goppa et sa matrice de parité $\mathcal{H}$ de taille $(n-k) \times n$. On génère une permutation aléatoire $\mathcal{P}$ de taille $n \times n$ et une matrice inversible $\mathcal{Q}$ de taille $(n-k) \times (n-k)$. La clef publique est :

$$\mathcal{H}' = \mathcal{Q} \times \mathcal{H} \times \mathcal{P}.$$

Chiffrement

Pour chiffrer on commence par coder le message[2] m en un mot e_m de poids t et de longueur n . On peut donc mettre au plus $\log_2 \binom{n}{t}$ bits d'information dans m.

Le chiffré transmis est le syndrome $\mathcal{S}$ de ce mot :

$$\mathcal{S} = \mathcal{H}' \times e_m^T.$$

Déchiffrement

Pour déchiffrer on procède comme avec le système de McEliece. On commence par calculer :

$$\mathcal{Q}^{-1} \times \mathcal{S} = \mathcal{H} \times \mathcal{P} e_m^T.$$

Encore une fois $\mathcal{P} e_m^T$ est de poids t lui aussi et on sait donc retrouver l'erreur correspondante au syndrome $\mathcal{Q}^{-1} \times \mathcal{S}$. On retrouve donc $\mathcal{P} e_m^T$ et donc aussi e_m. On en déduit alors le message clair m.

Ici, la clef publique fera $n(n-k)$ bits (ou $k(n-k)$ sous forme systématique), les blocs sont de longueur $\log_2 \binom{n}{t}$ et le taux de transmission est $\frac{\log_2 \binom{n}{t}}{n-k}$. Le chiffrement est lui beaucoup moins coûteux avec juste le codage en mot de poids constant (qui sera malheureusement souvent l'étape la plus coûteuse) et la somme de t colonnes de $\mathcal{H}$ pour un coût de $t(n-k)$. Le déchiffrement a lui un coût très similaire, mais un certain gain sur les produits de matrice puisque l'on travaille sur des tailles un peu plus petites.

Pour les paramètres $[2048, 1685, 67]$, cela donne un taux de transmission de 0.66 (donc un peu moins que McEliece), des blocs de 363 bits pour 240 bits d'information (donc beaucoup plus courts que ceux de McEliece) et une matrice de 750 kbits (ou encore 600 kbits sous forme systématique).

Ce système a donc l'avantage d'avoir des blocs beaucoup plus courts et un chiffrement bien plus rapide, sans pour autant perdre beaucoup sur les autres points vis-à-vis du cryptosystème de McEliece.

[2] voir section 5-4.1 page 63 pour plus de détails sur comment faire cela

2-2.3 Un schéma d'identification Zero-Knowledge

L'idée qui se cache derrière le terme de *schéma d'identification* est due à Fiat et Shamir [35] : elle consiste à utiliser un protocole à clef publique permettant à n'importe quel individu $\mathcal{A}$ ayant généré une paire clef publique/clef privée de s'identifier, avec une probabilité aussi bonne que désirée, auprès d'un autre individu $\mathcal{B}$. Le concept de *Zero-Knowledge* est essentiel à un tel protocole car il garantit qu'aucune information concernant la clef secrète de $\mathcal{A}$ ne risque de fuir au court du protocole, quelles que soient les requêtes de $\mathcal{B}$. Cela garantit par la même occasion que $\mathcal{B}$ ne pourra faussement prouver son identité en essayant de reproduire les réponses de $\mathcal{A}$.

Le protocole que je présente ici a été mis au point par Girault [39] en reprenant une idée de Stern [89]. Dans ce protocole, tous les utilisateurs partagent une même matrice de parité $\mathcal{H}$ binaire de taille $r \times n$ choisie aléatoirement. La clef secrète de chaque utilisateur est un mot e de poids t et sa clef publique est le syndrome $\mathcal{S}_e = \mathcal{H} \times e$ qui y est associé. Retrouver la clef secrète à partir de la clef publique nécessite donc de résoudre une instance du problème SD (voir chapitre 6 page 77). Le protocole fonctionne ensuite en trois étapes :

1. $\mathcal{A}$ choisit une matrice de permutation aléatoire $\mathcal{P}$ de taille $n \times n$ et une matrice inversible $\mathcal{Q}$ de taille $r \times r$. $\mathcal{A}$ transmet à $\mathcal{B}$ la matrice permutée $\mathcal{H}' = \mathcal{Q} \times \mathcal{H} \times \mathcal{P}$ et sa clef publique modifiée $\mathcal{S}'_e = \mathcal{Q}\mathcal{S}_e$.
2. $\mathcal{B}$ choisit un bit b et le transmet à $\mathcal{A}$.
3. – Si $b = 0$, $\mathcal{A}$ répond en donnant $\mathcal{Q}$ et $\mathcal{P}$ à $\mathcal{B}$ qui vérifie que l'on a bien $\mathcal{Q}\mathcal{H}\mathcal{P} = \mathcal{H}'$ et $\mathcal{Q}\mathcal{S}_e = \mathcal{S}'_e$.
   – Si $b = 1$, $\mathcal{A}$ répond en donnant $e' = \mathcal{P}^{-1}e$ à $\mathcal{B}$ qui vérifie que le poids de e' est bien t et que $\mathcal{H}'e' = \mathcal{S}'_e$.

Ces trois étapes sont ensuite répétées s fois pour garantir que $\mathcal{A}$ est bien lui-même avec une probabilité de $1 - 2^{-s}$.

En effet, un usurpateur d'identité ne peut pas répondre au challenge de $\mathcal{B}$ correctement dans les deux cas ($b = 0$ et $b = 1$) sans connaître la clef secrète de $\mathcal{A}$. Il a donc une probabilité de succès d'au plus $\frac{1}{2}$ pour chaque challenge de $\mathcal{B}$.

Ce protocole est bien *Zero-Knowledge* si on considère que résoudre une instance du problème SD ou retrouver une permutation entre deux matrices sont deux problèmes difficiles. Cependant il n'est pas très pratique puisqu'il nécessite d'échanger au moins une matrice permutée (qui doit être de taille conséquente pour assurer une bonne sécurité) à chaque tour du protocole.

De plus, en utilisant l'algorithme de séparation du support [85], retrouver une permutation entre deux codes peut être facile dans la plupart des cas. Dans ces cas-là le protocole n'est alors plus *Zero-Knowledge*. On peut toutefois contrer cette attaque en choisissant des matrices particulières, correspondant à des codes ayant un *hull* (l'espace vectoriel intersection entre un code et son dual) de grande dimension.

2-2.4 Le schéma d'identification de Stern

Cet autre schéma d'identification, mis au point par Stern en 1993 [90] reprend des idées similaires au schéma précédent. Il a en revanche l'avantage d'être

beaucoup plus pratique, dans le sens où la quantité de données transmises est ici beaucoup plus raisonnable.

Encore une fois, tous les utilisateurs partagent une matrice de parité binaire $\mathcal{H}$ et les clef secrètes et publiques des utilisateurs sont des mots e de poids donné t et leurs syndromes $\mathcal{S}_e$.

Il utilise en revanche un nouveau concept afin de minimiser la quantité de données transmises : c'est la notion de *commitment*, consistant à prouver que l'on connaît un élément en ne donnant que son haché, fourni par une fonction de hachage cryptographique sûre.

Le protocole est alors le suivant :

1. $\mathcal{A}$ choisit un mot y de longueur n aléatoire et une permutation σ sur n éléments. Il transmet à $\mathcal{B}$ des *commitments* c_1, c_2 et c_3 pour les trois valeurs suivantes : $(\sigma, \mathcal{H}y)$, $\sigma(y)$ et $\sigma(y \oplus e)$.
2. $\mathcal{B}$ répond par un élément de $\{0, 1, 2\}$.
3. – Si $b = 0$, $\mathcal{A}$ dévoile y et σ et $\mathcal{B}$ vérifie les *commitments* c_1 et c_2.
   – Si $b = 1$, $\mathcal{A}$ dévoile $y \oplus e$ et σ et $\mathcal{B}$ vérifie c_1 et c_3. Notons qu'il peut vérifier c_1 car $\mathcal{H}y = \mathcal{H}(y \oplus e) \oplus \mathcal{S}_e$.
   – Si $b = 2$, $\mathcal{A}$ dévoile $\sigma(y)$ et $\sigma(e)$ et $\mathcal{B}$ vérifie c_2 et c_3 et que le poids de $\sigma(e)$ est bien t.

Pour chaque tour, quelqu'un qui ne connaît pas e ne peut pas répondre juste avec une probabilité de plus de $\frac{2}{3}$.

Ce schéma est encore une fois *Zero-Knowledge*, et l'algorithme de séparation du support ne peut rien contre lui puisqu'à aucun moment des permutations de matrices n'entrent en jeu. De plus, l'utilisation de *commitments* rend la quantité de communication nécessaire relativement faible.

Deuxième partie

Les codes de Goppa

Chapitre 3

Propriétés des codes de Goppa

NTRODUITS en 1970, les codes de Goppa [11, 43] sont des codes linéaires sur un corps fini $\mathbb{F}_p$. Ils ont dans un premiers temps beaucoup été étudiés pour leurs propriétés en tant que codes correcteurs d'erreurs et des algorithmes de décodage efficaces ont ainsi été mis au point [64, 76]. Ensuite, avec l'apparition du cryptosystème de McEliece [66] ils ont été étudiés pour leurs propriétés cryptographiques.

Notons que le terme de code de Goppa désigne aussi parfois des codes appartenant à la famille des codes géométriques [44, 93], eux aussi découverts par Goppa. Ces codes ont aussi de très bonnes propriétés et de multiples applications dont je ne traiterai cependant pas ici. Le terme de code de Goppa désignera donc toujours ici les codes de Goppa classiques.

Dans ce chapitre nous verrons tout d'abord comment sont construits les codes de Goppa. Nous verrons ensuite quelques algorithmes permettant de les décoder efficacement. Enfin, nous regarderons les propriétés particulières des codes de Goppa binaires, qui en font de bons codes pour la cryptographie.

3-1 Construction

Les codes de Goppa sont des codes linéaires sur un corps fini $\mathbb{F}_p$. Cependant, leur construction passe par l'utilisation d'une extension $\mathbb{F}_{p^m}$. Un code de Goppa $\Gamma(\mathcal{L}, g)$ est défini par son polynôme de Goppa g de degré t à coefficients dans $\mathbb{F}_{p^m}$ et son support $\mathcal{L} \subset \mathbb{F}_{p^m}$ de n éléments. Si on note $\alpha_0, \ldots, \alpha_{n-1}$ les éléments de son support, sa matrice de parité est alors obtenue à partir de la matrice suivante[1] :

$$\mathcal{H}_{aux} = \begin{bmatrix} \frac{1}{g(\alpha_0)} & \cdots & \frac{1}{g(\alpha_{n-1})} \\ \vdots & & \vdots \\ \frac{\alpha_0^{t-1}}{g(\alpha_0)} & \cdots & \frac{\alpha_{n-1}^{t-1}}{g(\alpha_{n-1})} \end{bmatrix}$$

[1] le polynôme g ne peut donc pas avoir de racines parmi les éléments du support $\mathcal{L}$.

Chaque élément de cette matrice est ensuite décomposé en m éléments de $\mathbb{F}_p$, placés en colonnes, en utilisant une projection de $\mathbb{F}_{p^m}$ dans $\mathbb{F}_p^m$. On passe ainsi d'une matrice de taille $t \times n$ sur $\mathbb{F}_{p^m}$ à une nouvelle matrice de parité $\mathcal{H}$ de taille $mt \times n$ sur $\mathbb{F}_p$.

Les éléments du code $\Gamma(\mathcal{L}, g)$ seront donc tous les éléments c de $\mathbb{F}_p^n$ tels que $\mathcal{H} \times c^T = 0$. C'est donc un code de longueur n et dimension $k \geq n - mt$. De plus, un tel code a une distance minimale au moins égale à $t+1$. En effet, toute sous-matrice carrée $t \times t$ de $\mathcal{H}_{aux}$ est inversible car $\mathcal{H}_{aux}$ s'écrit comme le produit d'une matrice de Vandermonde et d'une matrice diagonale inversible :

$$\mathcal{H}_{aux} = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & & \vdots \\ \alpha_0^{t-1} & \cdots & \alpha_{n-1}^{t-1} \end{bmatrix} \times \begin{bmatrix} \frac{1}{g(\alpha_0)} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{g(\alpha_{n-1})} \end{bmatrix}.$$

Il n'existe donc pas de mot de code de poids inférieur ou égal à t. Les codes de Goppa sont donc de la forme $[n, k \geq n - mt, d \geq t+1]$ sur $\mathbb{F}_p$, avec comme seule contrainte de longueur $n \leq 2^m$.

Par rapport aux codes de Reed-Solomon vus section 1-2.3 page 18 ils ont donc l'avantage de ne pas être limités pour leur longueur en fonction de la taille du corps sur lequel on veut travailler. De plus, si on fixe $m = 1$ on se retrouve avec des codes qui ont exactement les même performances que les Reed-Solomon.

3-1.1 Une autre caractérisation des éléments du code

Il est important de remarquer que l'on peut caractériser l'appartenance au code $\Gamma(\mathcal{L}, g)$ sans avoir à faire appel à une matrice de parité du code. En effet, si comme pour la construction de $\mathcal{H}_{aux}$ on indexe les coordonnées des mots de code par les éléments du support $\mathcal{L}$ et que l'on note c_β la coordonnée du mot c associée à l'élément $\beta \in \mathcal{L}$, on a :

$$c \in \Gamma(\mathcal{L}, g) \quad \Longleftrightarrow \quad \sum_{\beta \in \mathcal{L}} \frac{c_\beta}{z - \beta} = 0 \mod g(z).$$

3-2 Décodage d'un code de Goppa

Plusieurs techniques existent pour décoder les codes de Goppa, mais elles fonctionnent toutes sur le même principe : si on cherche à décoder un mot $c' = c + e$, où c est un élément du code de Goppa et e une erreur de poids inférieur ou égal à $\frac{t}{2}$, on va commencer par calculer un syndrome sur $\mathbb{F}_{p^m}$ de ce mot bruité, à partir de ce syndrome on va écrire ce que l'on appelle une équation clef et on finira le décodage en résolvant l'équation clef pour retrouver e.

3-2.1 L'algorithme de Patterson

Cet algorithme, mis au point en 1975 [76] est le plus ancien pour décoder les codes de Goppa. Il ne fait pas appel à la matrice de parité $\mathcal{H}$ que nous avons écrite précédemment et n'a besoin que de la connaissance du support $\mathcal{L}$ et du générateur g du code $\Gamma(\mathcal{L}, g)$.

a) Le syndrome

Dans cet algorithme, le syndrome $\mathcal{R}_c$ d'un mot c est un polynôme sur $\mathbb{F}_{p^m}$ calculé modulo $g(z)$. Il s'écrit :

$$\mathcal{R}_c(z) = \sum_{\beta \in \text{supp}(c)} \frac{c_\beta}{z - \beta} \mod g(z).$$

Si ce syndrome est nul, le mot appartiendra au code : c'est la caractérisation alternative que nous venons de voir.

b) L'équation clef

Afin de pouvoir décoder $\frac{t}{2}$ erreurs dans le code de Goppa, il va falloir être capable, à partir du syndrome $\mathcal{R}_e$ d'une erreur e de poids $\leq \frac{t}{2}$, de retrouver les $\frac{t}{2}$ coefficients e_β dans la somme modulo $g(z)$.

Supposons que l'on ait donc reçu le syndrome $\mathcal{R}_e$ d'un mot de poids $\leq \frac{t}{2}$. On appellera *polynôme localisateur* d'un mot e, le polynôme σ_e unitaire qui a pour racines tous les éléments du support de e. On a donc :

$$\sigma_e(z) = \prod_{\beta \in \text{supp}(e)} (z - \beta).$$

Ce polynôme est donc de degré $\leq \frac{t}{2}$. On introduit alors le polynôme $\omega_e(z)$ défini par :

$$\omega_e(z) = \sigma_e(z)\mathcal{R}_e(z) \mod g(z). \tag{3-1}$$

Ce polynôme sera appelé *polynôme évaluateur* de l'erreur e car si on connaît sa valeur en un point β du support de e on peut déterminer la valeur de l'erreur en ce point. De même l'équation 3-1 sera appelée *équation clef* modulo g. Si on est capable de résoudre cette équation, c'est-à-dire de trouver des polynômes ω_e et σ_e de degrés suffisamment petits la vérifiant, on pourra retrouver e et décoder.

c) Résolution de l'équation clef

La résolution peut se faire de deux façons différentes : soit à l'aide de l'algorithme de Berlekamp-Massey [64] (comme expliqué dans l'article original de Patterson [76]), soit avec un algorithme d'Euclide étendu, qui a l'avantage d'être plus simple à présenter. En effet, on cherche à trouver ω_e et σ_e de degrés $\leq \frac{t}{2}$ qui vérifient :

$$\omega_e(z) = \sigma_e(z)\mathcal{R}_e(z) \mod g(z) = \sigma_e(z)\mathcal{R}_e(z) + k(z)g(z).$$

Si on essaye de calculer le PGCD de $\mathcal{R}_e$ et g avec l'algorithme d'Euclide étendu, on va calculer à chaque étape des polynômes u_i, v_i et r_i vérifiant :

$$\mathcal{R}_e u_i + g v_i = r_i.$$

À chaque étape les polynômes u_i et v_i seront de degré inférieur ou égal à i et le polynôme r_i sera de degré au plus $t - i$. Il existe donc une étape i_0 à laquelle si on arrête l'algorithme on va trouver une solution à l'équation : $\sigma_e = u_{i_0}$ et $\omega_e = r_{i_0}$ à un coefficient scalaire près.

3-2.2 Les codes de Goppa sont des codes alternants

On peut aussi décoder les codes de Goppa en utilisant la matrice de parité telle qu'elle a été définie en premier lieu. En effet, cette matrice correspond à une matrice de parité de code alternant, et le code peut donc être décodé comme un code alternant (voir [63] pour plus de détails). Dans ce contexte, on va tout de même procéder de façon similaire, en calculant d'abord un syndrome, puis en établissant une équation clef et en la résolvant.

a) Le syndrome

Supposons encore une fois que l'on a reçu un mot bruité $c' = c + e$. On part du syndrome sur $\mathbb{F}_p$ défini par $\mathcal{R}_e = \mathcal{H}c' = \mathcal{H}e$. Ce syndrome est de taille mt et on va le réécrire sous la forme d'un polynôme de degré $t-1$ à coefficients dans $\mathbb{F}_{p^m}$: les m premiers termes constituent le terme constant, les m suivants le coefficient de degré 1, et ainsi de suite...

Notons que, vue la forme de $\mathcal{H}_{aux}$, on peut aussi écrire ce syndrome sous la forme suivante :

$$\mathcal{R}_e = \sum_{\beta \in \mathrm{supp}(e)} \frac{e_\beta}{g(\beta)} \frac{1}{1-\beta z} \mod z^t.$$

b) L'équation clef

La forme de l'équation clef vient alors très facilement et on écrira :

$$\tilde{\sigma}_e(z)\mathcal{R}_e(z) = \omega_e(z) \mod z^t.$$

En notant toutefois que le localisateur $\tilde{\sigma}_e$ est cette fois-ci le polynôme qui a pour racines les inverses des éléments du support :

$$\tilde{\sigma}_e(z) = \prod_{\beta \in \mathrm{supp}(e)} (1-\beta z).$$

c) Résolution de l'équation clef

Pour résoudre cette nouvelle équation tout se passe exactement comme avec l'algorithme de Patterson, soit en appliquant l'algorithme de Berlekamp-Massey [64], soit un Euclide étendu.

3-3 Propriétés des codes de Goppa binaires

Les codes de Goppa binaires correspondent au cas particulier où l'on choisit $p = 2$. La matrice de parité $\mathcal{H}$ du code est alors une matrice binaire de taille $mt \times n$ construite à partir de la matrice $\mathcal{H}_{aux}$ de taille $t \times n$ à coefficients dans $\mathbb{F}_{2^m}$ décrite précédemment. Comme nous allons le voir, par rapport au cas général, les codes binaires présentent certains avantages les rendant, par la même occasion, bien adaptés à des usages cryptographiques.

3-3.1 Une nouvelle caractérisation des éléments du code

Comme nous l'avons vu section 3-1.1 page 34, les mots du code sont les mots vérifiant :

$$\sum_{\beta \in \mathcal{L}} \frac{c_\beta}{z-\beta} = 0 \mod g(z).$$

Dans le cas binaire c_β ne peut être égal qu'à 0 ou 1 et l'équation peut donc se réécrire :

$$\sum_{\beta \in \mathrm{supp}(c)} \frac{1}{z-\beta} = 0 \mod g(z),$$

or, si on dérive le polynôme localisateur de c on trouve une écriture similaire :

$$\frac{d\sigma_c(z)}{dz} = \sigma_c(z) \sum_{\beta \in \mathrm{supp}(c)} \frac{1}{z-\beta}.$$

Le polynôme g ayant été choisi sans racines sur $\mathcal{L}$ et σ_c étant scindé, $\frac{d\sigma_c(z)}{dz}$ sera égal à 0 modulo $g(z)$ si et seulement si la somme l'est aussi.

Un mot appartient donc à un code de Goppa binaire $\Gamma(\mathcal{L}, g)$ si et seulement si la dérivée de son polynôme localisateur est divisible par le générateur g du code.

3-3.2 Capacité de correction

Pour l'instant, la seule contrainte sur g était de ne pas avoir de racine parmi les éléments de $\mathcal{L}$. Si on lui ajoute la contrainte supplémentaire d'être sans facteurs multiples on va pouvoir doubler la capacité de correction du code. En effet, quand on utilise la caractérisation précédente, on veut que la dérivée du localisateur d'un mot du code soit divisible par g. Or, sur $\mathbb{F}_{2^m}$, la dérivée d'un polynôme ne contient pas de facteurs de degré impair, et il existe donc toujours un polynôme f vérifiant :

$$\frac{d\sigma_c(z)}{dz} = f^2(z).$$

De ce fait, si g divise la dérivée de σ_c, il divisera aussi f^2. Si g est sans facteurs multiples, cela veut donc dire qu'il doit nécessairement diviser f.

Un mot qui appartient au code $\Gamma(\mathcal{L}, g)$ avec g sans facteurs multiples appartiendra donc aussi au code $\Gamma(\mathcal{L}, g^2)$ qui a lui une distance minimale de $2t+1$ et un algorithme de décodage jusqu'à t erreurs. Un code de Goppa binaire a donc une distance minimale plus grande et une capacité de correction doublée.

a) L'algorithme de décodage

Si on reprend la caractérisation avec la matrice de parité, cela signifie que la matrice de parité $\mathcal{H}_{aux}$ de taille $t \times n$ du code $\Gamma(\mathcal{L}, g)$ engendre toutes les lignes de la matrice de parité du code $\Gamma(\mathcal{L}, g^2)$. On peut donc calculer une matrice $\mathcal{K}$ de taille $2t \times t$ telle que :

$$\mathcal{K} \times \mathcal{H}_{aux}(g) = \mathcal{H}_{aux}(g^2).$$

Pour décoder t erreurs dans le code $\Gamma(\mathcal{L}, g)$ il suffit donc de calculer le syndrome normalement avec $\mathcal{H}_{aux}(g)$, de l'étendre à un syndrome double à l'aide de $\mathcal{K}$ et de résoudre l'équation clef modulo z^{2t}.

b) L'algorithme de Patterson pour le cas binaire

Dans son papier d'origine, Patterson donne un autre algorithme pour faire cela sans avoir à étendre le syndrome, et en restant modulo $g(z)$. Il faut pour

cela d'abord remarquer que dans le cas binaire, le polynôme ω_e que l'on cherche est exactement la dérivée de σ_e. L'équation clef peut donc se réécrire :

$$\sigma_e(z)\mathcal{R}_e(z) = \frac{d\sigma_e(z)}{dz} \mod g(z).$$

Pour la résoudre on décompose σ_e en une partie paire et une partie impaire : $\sigma_e(z) = u(z)^2 + zv(z)^2$. On procède ensuite en trois étapes :

1. on calcule $h(z) = \frac{1}{\mathcal{R}_e(z)} \mod g(z)$,
2. on calcule $\mathcal{S}(z)$ tel que $\mathcal{S}(z)^2 = z + h(z) \mod g(z)$ (c'est l'étape qui correspond à l'extension du syndrome),
3. on résout la nouvelle équation clef : $v(z)^2\mathcal{S}(z)^2 = u(z)^2 \mod g(z)$ qui se réécrit pour g sans facteurs multiples :

$$v(z)\mathcal{S}(z) = u(z) \mod g(z)$$

On se ramène ainsi à résoudre l'équation pour des polynômes de degré deux fois plus bas qui permettent de reconstituer ensuite le polynôme localisateur entier.

3-3.3 Indistinguabilité

Ce que l'on appelle indistinguabilité des codes de Goppa binaires désigne simplement le fait qu'il est calculatoirement difficile de distinguer un code de Goppa dont on ne connaît ni le support, ni le générateur d'un code aléatoire de même longueur et même dimension.

Un distingueur de code de Goppa sera un algorithme capable de dire en temps τ, avec une probabilité supérieure à $\frac{1}{2} + \varepsilon$, si un code qu'on lui donne en entrée est un code de Goppa ou non. Il est conjecturé qu'un tel distingueur ne peut pas exister avec un quotient $\frac{\tau}{\varepsilon}$ polynomial en les paramètres du code.

a) Code de Goppa permuté

La propriété d'indistinguabilité n'a donc normalement pas besoin de faire appel à une matrice de parité particulière pour avoir un sens. Cependant, plutôt que de parler d'un code dont on ne connaît pas le support on a en général tendance à parler d'un code de Goppa permuté, ce qui désigne en fait le code associé à une matrice de parité $\mathcal{H}'$, obtenue en « mélangeant » la matrice $\mathcal{H}$.

Les deux façons de voir les choses sont pourtant à peu près les mêmes : pour représenter un code $\Gamma(\mathcal{L}, g)$ sans donner $\mathcal{L}$ et g, le seul moyen est de donner une matrice génératrice ou une matrice de parité $\mathcal{H}'$ de ce code.

Si $\mathcal{H}_g$ désigne la matrice de parité obtenue directement à partir de $\mathcal{H}_{aux}$ pour un g donné et quand les α_i sont ordonnés, alors la matrice de parité $\mathcal{H}'$ utilisée pour représenter $\Gamma(\mathcal{L}, g)$ est nécessairement de la forme :

$$\mathcal{H}' = \mathcal{Q}\mathcal{H}_g\mathcal{P},$$

où $\mathcal{P}$ est une permutation et $\mathcal{Q}$ une matrice inversible.

En effet, $\mathcal{P}$ sera la permutation qui transforme les α_i bien ordonnés en $\mathcal{L}$, et une fois permutée $\mathcal{H}\mathcal{P}$ ayant le même noyau que $\mathcal{H}'$ (ce sont deux matrices de

parité d'un même code $\Gamma(\mathcal{L}, g)$), il existe une application linéaire $\mathcal{Q}$ inversible pour passer de l'une à l'autre.

Avec ce point de vue, un distingueur de code de Goppa permuté sera un algorithme prenant en entrée une matrice binaire $\mathcal{H}'$ et décidant en temps τ avec une probabilité d'être exacte de $\frac{1}{2} + \varepsilon$ s'il existe deux matrices $\mathcal{Q}$ et $\mathcal{P}$ et un polynôme g permettant d'avoir $\mathcal{H}' = \mathcal{Q}\mathcal{H}_g\mathcal{P}$.

b) Les meilleurs distingueurs

Aucun bon distingueur de code de Goppa binaire n'est connu à ce jour. Les meilleures techniques connues consistent à énumérer tous les codes de Goppa ayant les paramètres adéquats, et pour chacun d'eux, tester s'il est équivalent au code à distinguer. Le coût d'un tel algorithme est développé section 5-3.2.a page 61 dans le cadre d'une attaque structurelle (visant à retrouver la clef secrète) sur le schéma de signature McEliece.

Afin d'avoir un meilleur distingueur il faudrait être capable d'exhiber une propriété des codes de Goppa, invariante par passage à un code équivalent (voir section 2-1.4 page 24), qui ne soit pas vérifiée (avec une probabilité de $\frac{1}{2} + \varepsilon$ au moins) pour un code aléatoire. Malheureusement, les invariants que l'on sait calculer facilement pour un code ne nous apportent aucune information. Même l´énumérateur des poids, qui est certainement le meilleur invariant qui soit en matière de codes, est en général très proche de la distribution binomiale d'un code aléatoire.

Chapitre 4

Mots de poids minimum et distribution des poids des codes de Goppa binaires

PAR construction, les codes de Gopppa, ont une distance minimale qui est au moins égale à $t+1$. Pour les codes de Goppa binaires on peut même montrer que cette distance est au moins égale à $2t+1$ (à condition que g soit sans facteurs multiples). Cependant, il n'existe aucune preuve que des mots ayant ce poids là exactement appartiennent au code. Si une telle preuve semble difficile à donner, il pourrait, en revanche, être plus facile de vérifier cela de manière expérimentale. En même temps, il serait intéressant de pouvoir connaître le nombre moyen de mots de poids $2t+1$ dans ces codes.

Comme vu dans la section 3-3.3 page 38, un code de Goppa binaire dont on ne connaît pas le support est souvent considéré comme indistinguable d'un code aléatoire. On s'attend donc à ce que leur distribution de poids soit proche de la distribution de poids moyenne d'un code aléatoire : une distribution gaussienne. Le nombre de mots de poids w de cette distribution est :

$$\mathcal{N}_w = \frac{\binom{n}{w}}{2^{n-k}}.$$

Des bornes théoriques sur la distribution de poids d'un Goppa binaire sont données dans [62]. Ces bornes sont très précises pour les mots de poids moyen (autour de $\frac{n}{2}$) et montrent bien que la distribution est exactement celle d'un code aléatoire moyen. En revanche, pour les plus petits poids la borne devient moins précise et n'apporte aucune information sur le nombre de mots de poids minimum. Voyons donc si, expérimentalement, on peut vérifier que la distribution est celle que l'on attend pour des mots de poids faible.

Ces résultats ont fait l'objet d'une publication à ISIT 2003 [36].

4-1 Comment trouver des mots de poids minimum

La première méthode qui vient à l'esprit pour trouver des mots de poids minimum est de procéder à une recherche exhaustive. On choisit un mot de

poids $2t+1$ et on regarde s'il appartient au code. Cependant, si on considère que les mots sont bien répartis, la probabilité qu'un tel mot soit dans le code doit être de $\frac{1}{2^{n-k}} = \frac{1}{2^{mt}}$. Le nombre d'essais moyen pour trouver un mot de poids $2t+1$ serait donc de :

$$\mathsf{Ess}_{exh} = 2^{mt}$$

Si on veut pouvoir faire quelques statistiques sur le nombre moyen de mots de poids minimum il faudrait donc que le produit mt reste au maximum de l'ordre de 30 ou 40. Cette méthode ne permet donc en aucun cas de regarder ce qui se passe pour des codes de taille usuelle.

On peut aussi utiliser l'algorithme de Canteaut-Chabaud [17] qui permet de trouver des mots de poids faible dans un code quelconque. Cette méthode sera nettement plus efficace que la recherche exhaustive mais ne permettra quand même pas d'aller très loin : au mieux on pourra atteindre des valeurs de mt de l'ordre de 80 ou 100.

Je propose donc deux autres méthodes, un peu plus efficaces, tirant profit de la structure du code de Goppa.

4-1.1 Méthode basée sur le décodage

Le principal avantage que procure la structure de code de Goppa est l'existence d'un algorithme de décodage très efficace : à partir de n'importe quel mot, on sait dire s'il existe ou non un mot de code à distance t ou moins de lui. Si on essaye de décoder un mot de poids $t+1$, en cas de succès, l'algorithme renverra donc un mot de code ayant un poids compris entre 1 et $2t+1$. Par construction il n'y a pas de mot de code de poids inférieur ou égal à $2t$, et donc le décodage d'un mot de poids $t+1$ n'aboutit que si le *support* de ce mot (l'ensemble des éléments du support du code sur lesquels le mot est non nul) est inclus dans le support d'un mot de code de poids $2t+1$.

Pour remplacer la recherche exhaustive on peut donc tirer des mots de poids $t+1$ aléatoirement et essayer de les décoder. Le coût de la recherche dépend alors de la proportion de mots de poids $t+1$ décodables. Comme nous l'avons vu, pour qu'il soit décodable, un tel mot doit avoir un support inclus dans le support d'un mot de code de poids $2t+1$. De plus, le support d'un mots de poids $t+1$ ne peut pas être entièrement inclus dans les supports de deux mots de code de poids $2t+1$ différents car cela donnerait par addition un mot de code de poids $2t$.

On notera $\mathcal{D}_{t+1}$ le nombre de mots décodables de poids $t+1$. Si $\mathcal{N}_{2t+1}$ désigne le nombre de mots de code de poids $2t+1$, on a alors :

$$\mathcal{D}_{t+1} = \binom{2t+1}{t+1} \mathcal{N}_{2t+1}.$$

On peut en déduire que la probabilité qu'un mot de poids $t+1$ soit décodable est :

$$\mathcal{P}_{t+1} = \frac{\binom{2t+1}{t+1}}{\binom{n}{t+1}} \mathcal{N}_{2t+1}.$$

Si $\mathcal{N}_{2t+1}$ suit bien une loi binomiale comme on s'y attend, le nombre moyen d'essais pour trouver un mot de code de poids $2t+1$ en utilisant cette méthode devient alors :

$$\mathsf{Ess}_1 = \frac{\binom{n}{t+1}\, 2^{mt}}{\binom{2t+1}{t+1}\binom{n}{2t+1}} = \frac{2^{mt}}{\binom{n-t-1}{t}} \approx t!$$

Cette dernière approximation est valable si on considère que t est très petit par rapport à n et que $n = 2^m$ (c'est-à-dire qu'on a pris un code de support le plus grand possible).

Pour des valeurs de t relativement petites, cette méthode semble donc beaucoup plus accessible que la recherche exhaustive. En particulier, le coût ne dépend plus de n et on peut donc l'utiliser pour trouver des mots de poids minimum dans des codes aussi longs que l'on veut, à condition que leur capacité de correction reste petite (autour de 10).

4-1.2 Reconstitution du polynôme localisateur des mots de code

Cette deuxième méthode exploite une autre propriété des codes de Goppa binaires : la dérivée du polynôme localisateur associé à un mot du code (le polynôme qui a pour racines toutes les positions non nulles du mot) est toujours divisible par g et donc aussi par g^2. De plus, il est suffisant que cette propriété soit vérifiée pour que le polynôme localisateur soit celui d'un mot de code.

Quand on regarde ce qui se passe pour un mot de poids $2t+1$ on constate que le polynôme localisateur est de degré $2t+1$ et que sa dérivée est donc de degré $2t$, comme g^2. S'il s'agit d'un mot de code, les deux polynômes étant unitaires et de même degré, si l'un divise l'autre c'est qu'ils sont égaux. On sait donc que tous les mots de code de poids minimal ont un polynôme localisateur dont la dérivée vaut g^2. Puisque l'on est en caractéristique 2, tous les termes de degré pair s'annulent en dérivant, cela signifie que l'on connaît la moitié des coefficients du polynôme localisateur : ils sont identiques pour tous les mots de poids $2t+1$ du code. Si on sait compléter les coefficients manquants on peut alors trouver des mots de poids minimum.

On va donc procéder de la façon suivante : on connaît les $t+1$ coefficients de degré impair du polynôme localisateur que l'on cherche. On choisit aléatoirement les $t+1$ autres termes et on vérifie si le polynôme obtenu est bien un polynôme localisateur : il doit être scindé sur le support du code et sans facteurs multiples. Si c'est un polynôme localisateur valable, ses racines donnent alors un mot de code de poids minimum.

Avec cette méthode le nombre d'essais moyen pour trouver un mot est :

$$\mathsf{Ess}_2 = \frac{(2^m)^{t+1}}{\mathcal{N}_{2t+1}}$$

Une fois encore, si on suppose que le nombre de mots correspond à la distribution binomiale que l'on retrouve pour un code aléatoire, on trouve (encore une fois pour un code de longueur maximale $n = 2^m$) :

$$\mathsf{Ess}_2 = \frac{2^{m(2t+1)}}{\binom{n}{2t+1}} \approx (2t+1)!$$

Ici aussi, l'approximation n'est valable que si l'on considère que t est petit par rapport à n. Dans tous les cas, le nombre d'essais nécessaires avec cette méthode est nettement supérieur à celui de la méthode utilisant le décodage, et même si le test de cette méthode est un peu plus rapide qu'un décodage, les deux sont du même ordre de grandeur et cela ne permet en aucun cas de rattraper la différence.

Pour améliorer cela, il faudrait pouvoir directement choisir des coefficients tels que le polynôme localisateur ait une meilleure chance d'être scindé. Cependant, il semble très difficile de gagner un facteur aussi important que $\frac{2t+1!}{t+1!}$.

4-2 Utilisation à des fins statistiques

Maintenant que nous avons une méthode efficace pour trouver des mots de poids minimum, il va falloir trouver une façon de l'exploiter pour retrouver la distribution des mots de petit poids de nos codes de Goppa binaires.

4-2.1 Méthode utilisée

Comme vu juste avant, la probabilité qu'un mot de poids $t+1$ soit décodable s'écrit :

$$\mathcal{P}_{t+1} = \frac{\binom{2t+1}{t+1}}{\binom{n}{t+1}} \mathcal{N}_{2t+1}.$$

Si on mesure expérimentalement cette probabilité, on pourra ensuite en déduire le nombre de mots de poids $2t+1$ dans le code que l'on regarde. L'expérience que j'ai menée est donc la suivante :

1. Choisir un jeu de paramètres n, m et t et un compteur $i = 0$,
2. générer un code de Goppa aléatoire avec les paramètres précédents,
3. tirer des mots de poids $t+1$ jusqu'à en trouver un décodable,
4. répéter l'étape 3. cent fois et noter à chaque fois le nombre d'essais nécessaires,
5. calculer la moyenne σ_i des essais précédents et retourner à l'étape 2. en incrémentant i.

Pour chaque jeu de paramètres on conserve l'ensemble des moyennes σ_i récupérées à l'étape 5. du processus. Ainsi, on peut calculer la moyenne des $\frac{1}{\sigma_i}$ et connaître la valeur moyenne de $\mathcal{N}_{2t+1}$ pour un jeu de paramètres donné. De même, on peut calculer l'écart type des σ_i pour savoir si, en général, les codes de Goppa ont une valeur de $\mathcal{N}_{2t+1}$ proche de la moyenne obtenue.

4-2.2 Résultats attendus

En supposant que tous les codes de Goppa ont exactement une distribution de poids gaussienne on aurait toujours :

$$\mathcal{P}_{t+1} = \frac{\binom{2t+1}{t+1}}{\binom{n}{t+1}} \mathcal{N}_{2t+1} = \frac{1}{t!}.$$

Le nombre moyen d'essais pour trouver un mot de poids $t+1$ décodable serait de $t!$ avec un écart type de $t!$ aussi. Pour les σ_i on devrait donc trouver aussi une moyenne de $t!$, mais en revanche un écart type de $\frac{t!}{\sqrt{100}}$, car ce sont des moyennes sur 100 essais. Si on trouve un écart type plus grand, cela signifiera que, pour un même jeu de paramètres, certains codes de Goppa ont plus de mots de poids minimum que la moyenne et d'autre moins. Si on trouve moins, c'est que l'on a eu beaucoup de chance, mais donc aussi que les codes ont de fortes chances d'avoir à peu près toujours le même nombre de mots de poids minimum.

4-2.3 Résultats obtenus – interprétation des résultats

Le tableau 4-1 page suivante contient les résultats obtenus pour différents paramètres. À chaque fois on a choisi $n = 2^m$ puisque c'est presque toujours le cas auquel on s'intéresse avec les codes de Goppa. Le nombre d'essais effectués pour les codes corrigeant 10 erreurs est en général assez petit car les calculs sont assez longs, et trouver 100 mots dans un seul code peut déjà prendre beaucoup de temps.

La première chose que l'on remarque en regardant les statistiques obtenues est que, globalement, Σ et σ sont toujours relativement proches des valeurs attendues $t!$ et $\frac{t!}{10}$. On note quand même quelques tendances particulières :

- plus m est grand, plus on se rapproche, aussi bien pour la moyenne que pour l'écart type, des valeurs attendues.
- l'écart type σ est toujours très proche de $\frac{\Sigma}{10}$, et a même plutôt tendance à être en dessous, ce qui pousse à croire que des codes ayant les même paramètres ont tous à peu près le même nombre de mots de poids minimum.
- pour les codes corrigeant peu d'erreurs, les valeurs sont un peu supérieures à celles attendues, ce qui pourrait vouloir dire qu'un code de Goppa corrigeant peu d'erreurs a moins de mots de poids minimum. Cependant, ceci est principalement dû au fait que l'approximation du Σ attendu à $t!$ est moins juste car t n'est pas négligeable par rapport à 2^m pour ces valeurs. Avec la formule exacte on constate que l'on est toujours proche de la valeur théorique.
- on note quelques aberrations, comme par exemple pour les codes $m = 6$ et $t = 7$. La valeur énorme de l'écart type vient certainement du petit nombre de mots dans un tel code : c'est un code $[64, 22, 15]$ qui ne contient donc en moyenne qu'un trentaine de mots de poids minimum. Il suffit que certains codes en aient quelques uns de moins ou de plus pour très vite faire monter l'écart type.
- pour $t = 10$ les écarts type semblent tous anormalement bas (plus bas que ce que devraient donner des lois binomiales toute identiques et qui est donc le minimum que l'on devrait pouvoir atteindre). Je n'ai pas réellement d'explication à cela, mais cela peut venir d'un biais dans la façon de générer

m	t	Nb	Σ	σ
6	4	20	31	3.5
6	5	20	207	23
6	6	20	1 700	290
6	7	20	22 150	16 000
7	4	40	26	2.7
7	5	40	151	13
7	6	40	1 050	120
7	7	40	8 970	900
7	8	40	85 300	9 000
7	9	40	940 000	100 000
7	10	20	5 410 000	220 000
8	4	20	23	2.2
8	5	20	132	12
8	6	20	840	78
8	7	20	6 620	510
8	8	20	54 600	4 400
8	9	20	589 000	43 000
8	10	10	5 230 000	340 000
9	4	20	24	1.8
9	5	20	124	11
9	6	20	782	89
9	7	20	5 830	470
9	8	20	47 300	5 400
9	9	20	460 000	61 000
9	10	10	4 580 000	280 000
10	4	40	22	2.6
10	5	40	120	13
10	6	40	755	65
10	7	40	5 170	490
10	8	40	44 000	4 400
10	9	40	402 000	49 000
10	10	20	3 960 000	310 000
11	6	20	721	69
11	7	20	5 110	480
11	8	20	40 600	3 800
11	9	20	383 000	32 000
11	10	10	4 180 000	330 000
12	6	20	724	51
12	7	20	5 150	490
12	8	20	41 400	3 200
12	9	20	380 000	31 000
12	10	10	3 720 000	240 000
13	4	20	22	2.4
13	5	20	118	14
13	6	20	702	69
13	7	20	5 100	550
13	8	20	40 700	4 000
13	9	20	378 000	33 000
13	10	10	3 720 000	210 000
14	4	20	23	2.6
14	5	20	117	8.6
14	6	20	715	59
14	7	20	5 080	470
14	8	20	39 600	3 200
14	9	20	351 000	42 000
14	10	10	3 480 000	310 000
15	4	60	23	2.2
15	5	60	116	11
15	6	60	698	75
15	7	60	5 040	510
15	8	60	39 700	3 400
15	9	60	364 000	45 000
15	10	10	3 760 000	220 000
16	4	60	22	2.1
16	5	60	120	12
16	6	60	704	79
16	7	60	5 080	510
16	8	60	40 300	4 000
16	9	60	372 000	39 000
16	10	20	3 720 000	280 000

TAB. 4-1 – *Statistiques obtenues pour la recherche de mots de poids minimum – Nb : nombre de codes regardés ; Σ : moyenne des σ_i ; σ : écart type des σ_i.*

les syndromes aléatoires : avec 3 million d'essais en moyenne par mot, cela fait plus de 3 milliards de syndromes aléatoires à générer pour chaque code, ce qui est certainement trop pour la fonction `random` du C.

Ces résultats sont donc plutôt satisfaisants et confortent donc l'idée initiale selon laquelle les codes de Goppa ont une distribution de poids proche de celle d'un code aléatoire. Même si des codes corrigeant plus d'erreurs (comme ceux utilisés réellement pour le cryptosystème de McEliece), n'ont pas pu être testés, on peut supposer qu'ils auront quand même le nombre de mots de poids minimum attendus.

4-3 Application aux autres mots de petit poids

En décodant des mots de poids $t+1$ on peut trouver des mots de code de poids $2t+1$ et donc, déterminer le nombre moyen de tels mots. Maintenant, supposons que l'on essaye de décoder un mot de poids $t+2$; il peut alors se décoder en un mot de code de poids $2t+1$ ou en un mot de code de poids $2t+2$. En utilisant la même méthode que précédemment on peut déterminer la proportion des mots de poids $t+2$ qui sont décodables et si on connaît déjà le nombre de mots de code de poids $2t+1$ il doit alors être possible d'estimer le nombre de mots de code de poids $2t+2$.

4-3.1 Résultats attendus

Nous avons vu dans la section précédente que la probabilité qu'un mot de poids $t+1$ soit décodable était de $\frac{1}{t!}$. Si cela est vrai pour des mots de poids $t+1$ ce doit aussi être le cas pour des mots de poids plus grands. Si on essaye donc de décoder un mot de poids $t+2$ on devrait donc avoir les trois possibilités suivantes :

- il n'est pas décodable : cela arrive avec une probabilité de $1-\frac{1}{t!}$,
- il se décode en un mot de poids $2t+2$: probabilité de $\mathcal{N}_{2t+2} \times \frac{\binom{2t+2}{t+2}}{\binom{n}{t+2}}$,
- en un mot de poids $2t+1$: probabilité de $\mathcal{N}_{2t+1} \times \frac{\binom{2t+1}{t+2}}{\binom{n}{t+2}} = \frac{1}{n(t-1)!}$.

On peut aussi imaginer qu'un même mot se décode à la fois en un mot de poids $2t+1$ et un mot de poids $2t+2$, mais cela impliquerait l'existence d'un mot de code de poids $2t-1$ correspondant à la somme des deux autres mots de code. Bien sûr, cela n'est pas possible.

De ce fait, les trois possibilités sont bien distinctes et la somme des trois probabilités correspondantes doit être égale à un. On trouve donc :

$$\mathcal{N}_{2t+2} \times \frac{\binom{2t+2}{t+2}}{\binom{n}{t+2}} + \frac{1}{n(t-1)!} = \frac{1}{t!},$$

et donc,

$$\mathcal{N}_{2t+2} = \frac{\binom{n}{t+2}}{\binom{2t+2}{t+2}} \times \frac{1}{t!}\left(1-\frac{t}{n}\right) = \frac{n^{t+2}}{(2t+2)!} \times \left(1-\frac{t}{n}\right).$$

Au premier ordre on trouve donc :

$$\mathcal{N}_{2t+2} \approx \frac{\binom{n}{2t+2}}{n^t} = \frac{\binom{n}{2t+2}}{2^{n-k}}.$$

C'est exactement ce que l'on attend : une distribution binomiale. De même, si on regarde pour des poids plus grand on trouve en essayant de décoder des mots de poids $t+\alpha$:

- soit il n'est pas décodable : probabilité de $1 - \frac{1}{t!}$,
- soit il se décode en un mot de poids w compris entre α et $2t+\alpha$ (puisque l'on peut décoder jusqu'à t erreurs).

Dans ce deuxième cas, la probabilité est un peu plus difficile à évaluer. On décode un mot de poids $t+\alpha$ en un mot de code de poids w. Appelons β le nombre de positions communes à ces deux mots. Cela signifie que l'erreur (de poids inférieur ou égal à t) que l'on décode se décompose en une erreur de poids $t+\alpha-\beta$ incluse dans le support du mot initial et une erreur de poids $w-\beta$ incluse dans le support du mot de code. On a donc :

$$t + \alpha - \beta + w - \beta \leq t,$$

et on en déduit :

$$\beta \geq \left\lceil \frac{w+\alpha}{2} \right\rceil.$$

De plus, comme la probabilité que le décodage donne une erreur de poids strictement plus petit que t est quasi négligeable, on peut considérer que l'on a toujours $\beta = \left\lceil \frac{w+\alpha}{2} \right\rceil$. La probabilité que notre mot de poids $t+\alpha$ se décode en un mot de poids w s'exprime alors comme :

$$\mathcal{P}_{t+\alpha \to w} = \mathcal{N}_w \times \frac{\binom{w}{\beta}\binom{t+\alpha}{\beta}}{\binom{n}{t+\alpha}}.$$

En faisant la somme des probabilités de tous les événement possibles lors du décodage d'un mot de poids $t+\alpha$ on trouve donc :

$$\frac{1}{t!} = \sum_{w=\alpha}^{2t+\alpha} \mathcal{N}_w \frac{\binom{w}{\beta}\binom{t+\alpha}{\beta}}{\binom{n}{t+\alpha}} \quad \text{avec} \quad \beta = \left\lceil \frac{w+\alpha}{2} \right\rceil.$$

On peut en déduire (encore une fois pour $n = 2^m$) :

$$\begin{aligned}
\mathcal{N}_{2t+\alpha} &= \frac{\binom{n}{t+\alpha}}{\binom{2t+\alpha}{t+\alpha}} \times \left[\frac{1}{t!} - \sum_{w=\alpha}^{2t+\alpha-1} \mathcal{N}_w \frac{\binom{w}{\beta}\binom{t+\alpha}{\beta}}{\binom{n}{t+\alpha}} \right] \\
&= \frac{n^{t+\alpha}}{(2t+\alpha)!} \times \left[1 - t! \sum_{w=\alpha}^{2t+\alpha-1} \mathcal{N}_w \frac{\binom{w}{\beta}\binom{t+\alpha}{\beta}}{\binom{n}{t+\alpha}} \right]
\end{aligned}$$

Si on considère que n est grand, on peut alors négliger le terme contenu dans la somme et on trouve alors le résultat que l'on attend :

$$\mathcal{N}_{2t+\alpha} = \frac{\binom{n}{2t+\alpha}}{n^t}.$$

Il suffit donc de vérifier notre hypothèse initiale selon laquelle un mot de poids $t+\alpha$ a toujours une probabilité de $\frac{1}{t!}$ d'être décodable pour montrer que la distribution de poids d'un code de Goppa binaire coïncide avec celle d'un code aléatoire pour les mots de petit poids.

4-3.2 Résultats obtenus – interprétation des résultats

Le tableau 4-2 page suivante contient les moyennes obtenues pour différentes valeurs de α (et donc différents poids de mots $2t + \alpha$). Le cas $\alpha = 1$ est celui du tableau 4-1 page 46 concernant les mots de poids minimum uniquement.

Pour ces mesures, les même remarques que pour les mots de petit poids tiennent encore : les écarts types sont toujours très proches de $\frac{\Sigma}{10}$ (même si, certainement à cause du plus petit nombre de codes regardé, les écarts sont souvent plus importants) et Σ est toujours proche de $t!$ (ou, comme ce doit être le cas, un peu plus pour les petits m). À part cela, la valeur de α (le poids des mots cherchés) ne semble pas trop affecter les moyennes : du point de vue du décodage au moins, le syndrome d'un mot de petit poids (supérieur à $t + 1$ quand même) se comporte comme un syndrome aléatoire. On devrait donc bien retrouver une distribution binomiale autour de la distance minimale des codes de Goppa.

4-4 Conclusion

Dans sa thèse, Anne Canteaut [16, Chapitre 2, exemple 2.14] a étudié, entre autres, le nombre exact de mots de poids minimum dans des codes de Goppa corrigeant 3 erreurs. Les résultats qu'elle obtient sont reportés dans le tableau 4-3 page 51 et comparés à ceux attendus pour une vraie distribution binomiale. Plus la longueur du code augmente, plus on se rapproche de la distribution binomiale.

Les statistiques obtenues à partir des expérimentations décrites dans ce chapitre tendent à prouver exactement la même chose : plus les codes sont longs, plus on est proche de la distribution binomiale et, de façon générale, même pour des codes relativement courts, l'écart à la distribution binomiale est toujours très réduit.

On peut donc connaître le nombre exact de mots de poids minimum dans un code de Goppa corrigeant un petit nombre d'erreurs avec une recherche exhaustive (ou un algorithme un peu plus efficace [17]). Avec l'algorithme présenté ici on peut approximer ce nombre pour des codes corrigeant un peu plus d'erreurs. En revanche, pour des codes corrigeant un plus grand nombre d'erreurs comme ceux généralement utilisés, on ne peut qu'extrapoler les résultats obtenus pour les codes plus simples. Même s'il ne semble pas y avoir de raison pour que ce comportement change, une preuve reste toujours à donner.

poids $\longrightarrow$			$2t+2$		$2t+3$		$2t+4$	
m	t	Nb	Σ	σ	Σ	σ	Σ	σ
8	6	20	868	96	747	89	755	79
8	7	20	6 530	690	5 510	510	5 640	600
8	8	20	56 600	3 900	44 700	4 000	46 700	3 700
8	9	10	552 000	52 000	413 000	31 000	389 000	46 000
9	6	20	809	72	753	65	742	81
9	7	20	5 620	510	5 220	660	5 180	630
9	8	20	47 200	5 100	42 900	3 700	43 900	4 200
9	9	10	436 000	25 000	395 000	46 000	374 000	35 000
10	6	20	740	61	728	62	712	79
10	7	20	5 490	580	4 820	450	4 960	530
10	8	20	43 200	3 900	42 300	3 900	42 300	3 700
10	9	10	401 000	45 000	389 000	32 000	386 000	41 000
11	6	20	734	73	706	71	725	80
11	7	20	5 040	560	4 960	550	5 000	550
11	8	20	41 800	4 800	41 100	3 400	42 900	3 600
11	9	10	399 000	47 000	366 000	43 000	381 000	37 000
12	6	20	720	67	727	58	717	75
12	7	20	5 190	370	4 850	440	5 080	500
12	8	20	39 900	3 100	42 000	3 400	39 200	2 900
12	9	10	370 000	34 000	378 000	40 000	380 000	35 000
13	6	20	739	79	725	89	732	86
13	7	20	4 940	480	5 110	560	5 270	610
13	8	20	40 900	3 700	40 400	3 600	39 700	4 800
13	9	10	364 000	29 000	377 000	39 000	343 000	35 000
14	6	20	719	76	735	58	729	70
14	7	20	5 050	410	4 960	420	5 070	470
14	8	20	38 100	4 100	41 200	3 200	41 100	4 500
14	9	10	374 000	30 000	349 000	28 000	360 000	36 000
15	6	20	730	82	726	70	721	70
15	7	20	4 890	430	5 070	610	5 090	480
15	8	20	39 700	3 300	40 300	4 200	39 700	3 900
15	9	10	376 000	35 000	354 000	39 000	351 000	28 000
16	6	20	714	72	720	64	691	77
16	7	20	5 030	640	5 130	530	4 960	410
16	8	20	41 200	5 000	41 100	2 200	39 600	3 600
16	9	10	368 000	27 000	360 000	28 000	361 000	28 000

TAB. 4-2 – *Statistiques obtenues en cherchant des mots de poids faible – Nb : codes regardés ; Σ : moyenne des σ_i ; σ : écart type des σ_i.*

n	nombre exact	nombre attendu	écart
32	128	103	19.7%
64	$\sim$ 2 640	2 370	10.2%
128	47 616	45 073	5.3%
256	$\sim$ 806 000	784 510	2.7%
512	13 264 896	13 084 604	1.3%

TAB. 4-3 – *Nombre de mots de poids minimum dans les codes de Goppa corrigeant 3 erreurs.*

Chapitre 5

Signatures courtes utilisant le cryptosystème de McEliece

POUVOIR signer un document sur support numérique comme on le fait pour un document papier, mais avec en plus une véritable sécurité cryptographique, est un sujet qui intéresse les cryptographes depuis de nombreuses années. Le but d'une signature numérique est (à l'instar des signatures classiques) d'ajouter une petite quantité d'information à la fin d'un document afin de garantir qu'une personne bien identifiée est d'accord avec le contenu exact du document. Il faut ensuite que toute personne souhaitant vérifier que la signature est exacte puisse le faire. Une telle construction nécessite donc le même type de propriétés qu'un système de chiffrement à clef publique. C'est pour cela que dès l'apparition des premiers cryptosystèmes asymétriques ont commencé à apparaître les premiers schémas de signature numérique [33, 34, 73, 78, 84]. En effet, passer d'un cryptosystème à un schéma de signature est assez facile en utilisant la construction présentée en section 5-1.1.

Pour le cryptosystème de McEliece le problème est un peu différent : plusieurs modèles de schémas de signature utilisant des codes [4, 5, 49, 99] on été essayés, mais aucun n'était solide [3, 91, 94, 95, 100, 101] car il ne reposaient pas réellement sur la difficulté de décoder un code aléatoire. Le seul système qui se détache est celui de Kabatianskii, Krouk et Smeets [55] qui est solide, mais présente l'inconvénient de ne pouvoir être utilisé qu'une seule fois. Comme nous le verrons section 5-2 page 56, de façon générale, l'utilisation de codes correcteurs d'erreurs implique d'avoir un chiffré plus long que le message, et cela rend complexe l'utilisation d'une construction standard pour la signature.

Dans la suite de ce chapitre je montre comment, en utilisant des paramètres peu classiques du système de McEliece et avec la puissance de calcul des machines modernes, on peut contourner ces problèmes et obtenir deux schémas de signature efficace, presque identiques, qui, par la même occasion, s'avèrent donner les signatures les plus courtes connues à ce jour. En effet, ils permettent d'obtenir des signatures d'une longueur de 81 bits là où RSA [84], le système le plus couramment utilisé aujourd'hui, donne des signatures de 1536 bits pour une

sécurité équivalente. Les systèmes utilisant des courbes elliptiques permettent de descendre à 320 bits [54] mais cela reste encore nettement au dessus. Enfin, le concurrent le plus proche est Quartz [75] avec 128 bits dont la solidité repose sur le système HFE [74] dont le niveau de sécurité exact reste à établir [51].

Enfin, nous verrons section 5-6 page 72 que cette construction peut facilement passer à l'échelle et rester relativement efficace quand on fixe des contraintes de sécurité plus élevées, même si une puissance de calcul plus grande permettrait d'encore améliorer ses performances.

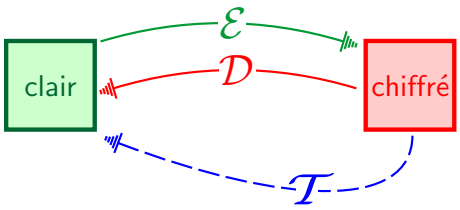
Les résultats présentés dans ce chapitre ont fait l'objet de plusieurs publications [22, 23, 24].

5-1 D'un cryptosystème à un schéma de signature

Historiquement, les cryptosystèmes à clef publique et les schémas de signature sont apparus en même temps, et les papiers fondateurs de la théorie [28, 33, 84] parlent en général des deux applications simultanément. En effet, les mécanismes utilisés dans les deux cas sont très proches et, la plupart du temps, passer de l'un à l'autre est très simple.

5-1.1 La construction usuelle

Supposons que l'on a le cryptosystème à clef publique suivant :



$\mathcal{E}$ est l'algorithme de chiffrement,

$\mathcal{D}$ le problème d'inversion, présumé calculatoirement difficile,

$\mathcal{T}$ est la trappe (secrète) qui permet de déchiffrer facilement.

On disposera aussi d'une fonction de hachage $\mathcal{M}$d que l'on suppose parfaite : on utilise le modèle de l'oracle aléatoire [9], ce qui permet de ne pas avoir à tenir compte de ses failles éventuelles dans l'étude de la sécurité du schéma signature. On peut alors construire un schéma (voir figure 5-1 page suivante) qui aura la même sécurité que le cryptosystème sous-jacent.

La signature se décompose en trois étapes :

1. calculer $h = \mathcal{M}\mathrm{d}(\mathsf{D})$,
2. calculer $s = \mathcal{T}(h)$
3. ajouter s à la suite de D.

Le document D est d'abord haché grâce à $\mathcal{M}$d pour obtenir h, de la même longueur qu'un chiffré du cryptosystème à clef publique utilisé. Ce haché est ensuite déchiffré en utilisant la trappe $\mathcal{T}$ du cryptosystème pour obtenir la signature s. Cette signature est ensuite apposée au document pour le transmettre.

La vérification est elle aussi faite de trois étapes :

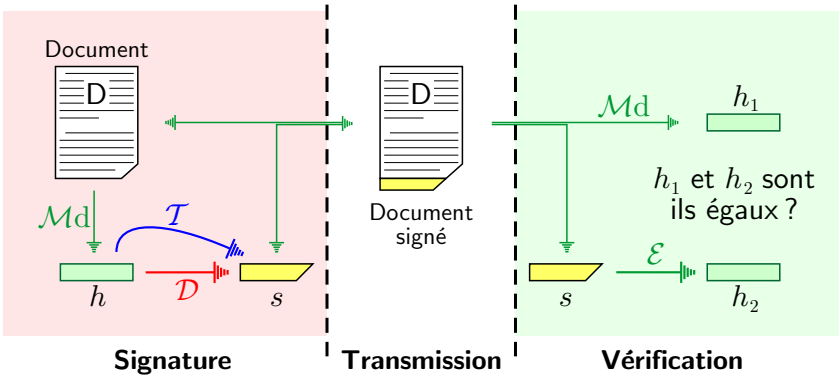


FIG. 5-1 – *Schéma général de construction d'un schéma de signature à partir d'un système de chiffrement à clef publique.*

1. calculer $h_1 = \mathcal{M}\mathrm{d}(\mathsf{D})$,
2. calculer $h_2 = \mathcal{E}(s)$,
3. comparer h_1 et h_2.

Le récepteur du document signé va d'une part, lui aussi, hacher le document D pour obtenir h_1 et d'autre part rechiffrer la signature s en utilisant $\mathcal{E}$ et la clef publique de la personne ayant signé le document. Il obtient ainsi h_2 qu'il va comparer au h_1 obtenu précédemment. Si les deux sont égaux c'est que la signature est valide.

5-1.2 Sécurité de cette construction

Créer une fausse signature (une contrefaçon) consiste toujours à trouver s et D tels que $\mathcal{E}(s) = \mathcal{M}\mathrm{d}(\mathsf{D})$. Pour cela trois choix d'attaques s'offrent à nous : soit on essaye de calculer $s = \mathcal{E}^{-1}(\mathcal{M}\mathrm{d}(\mathsf{D}))$, soit on essaye de calculer $\mathsf{D} = \mathcal{M}\mathrm{d}^{-1}(\mathcal{E}(s))$, soit on cherche une « collision ».

Le premier cas correspond exactement au problème $\mathcal{D}$ (on se donne un chiffré que l'on veut décoder) et devrait donc offrir la même sécurité. Un bon schéma de signature ne devrait pas offrir de meilleure attaque que celle ci.

Le deuxième cas correspond à l'inversion de la fonction de hachage. À partir du moment où nous nous sommes placés dans le modèle de l'oracle aléatoire cette attaque devient impossible : une fonction de hachage parfaite est calculatoirement non inversible.

Le dernier cas, la recherche de « collision », est donc la seule attaque nouvelle issue de la structure du schéma. L'idée est de construire deux listes, l'une de hachés et l'autre de chiffrés, et de chercher des éléments communs aux deux listes. Le célèbre paradoxe des anniversaires nous dit que si le haché et le chiffré sont de longueur r, alors deux listes d'environ $2^{\frac{r}{2}}$ éléments suffisent pour trouver en moyenne une telle collision. Cela prend un temps en $\mathcal{O}(r2^{\frac{r}{2}})$.

La plupart des cryptosystèmes à clef publique offrent une sécurité inférieure à $\mathcal{O}(r2^{\frac{r}{2}})$: par exemple, RSA nécessite une sortie de taille $r = 1536$ pour atteindre une sécurité supérieure à 2^{80} opérations. Sauf cas exceptionnel, cette

construction est donc aussi solide que le cryptosystème sous-jacent. Dans la pratique le modèle de l'oracle aléatoire n'est pas réaliste et il est toujours possible d'attaquer la fonction de hachage. Toutefois, une bonne fonction (comme celles utilisées couramment) offrira une sécurité aussi élevée que le coût de l'attaque par collision.

5-2 Application au cryptosystème de McEliece

Lorsque l'on essaye d'appliquer ce schéma de construction au cryptosystème de McEliece (voir la description du système section 2-2.1 page 25) on se heurte à un obstacle. En effet, le chiffrement McEliece consiste à transformer un message en mot d'un code de Goppa et à y ajouter une erreur décodable. La trappe $\mathcal{T}$ qui permet de décoder un nombre d'erreurs donné peut ainsi s'appliquer à tous les messages chiffrés par $\mathcal{E}$. Cependant, dans le schéma standard il est nécessaire d'appliquer $\mathcal{T}$ à un mot obtenu avec $\mathcal{M}\mathrm{d}$. À part pour un code parfait où tous les mots de l'espace seront décodables, rien ne garantit que $\mathcal{T}$ pourra s'appliquer à $h = \mathcal{M}\mathrm{d}(\mathsf{D})$.

Dans la pratique, la probabilité que h soit décodable est même très petite : avec les paramètres d'origine $[1024, 524, 101]$ pour le code de Goppa, cette probabilité est même de 2^{-216}. C'est pour cela que la signature a pu sembler impossible pendant de nombreuses années.

5-2.1 Densité des mots décodables

Pour rendre l'usage de la construction standard possible, il faudrait que cette probabilité soit très proche de 1, afin qu'une majorité des hachés soient décodables. Pour cela nous allons évaluer la densité des mots décodables.

Dans le cryptosystème de McEliece, le chiffrement consiste à convertir un message en mot de code (produit par la matrice génératrice du code) et à y ajouter une erreur de poids inférieur ou égal à la capacité de correction de la trappe. Avec les codes de Goppa on part donc d'un mot de longueur $k = n - mt$ que l'on transforme en mot de code de longueur $n = 2^m$, puis on y ajoute une erreur de poids t. La trappe (l'algorithme de décodage) permet de décoder n'importe quel mot construit de cette façon, c'est-à-dire, l'ensemble des mots à distance t ou moins d'un mot de code. Ces mots représentent donc un ensemble de boules de rayon t centrées sur les mots du code. Ces boules sont disjointes car la distance minimale construite du code est $2t + 1$ et elles ont donc une densité facile à calculer :

$$\mathcal{D}_{\mathrm{dec}} = \frac{2^k \times \binom{n}{t}}{2^n} = \frac{\binom{2^m}{t}}{2^{mt}} \approx \frac{1}{t!}$$

Cette densité est exactement la probabilité qu'un haché aléatoire soit décodable. C'est donc cette valeur que l'on voudrait voir proche de 1. Malheureusement cela voudrait dire qu'il faut prendre un code avec une capacité de correction d'une seule erreur et un tel code ne pourra jamais fournir une sécurité suffisante pour un usage dans le cryptosystème de McEliece.

Pour résoudre ce problème, les seuls moyens semblent être de modifier le cryptosystème pour avoir une fonction de chiffrement $\mathcal{E}$ surjective (et donc une

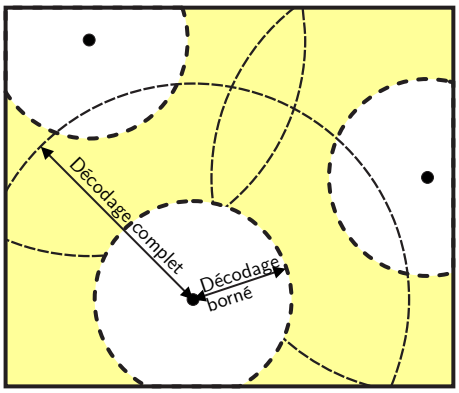


FIG. 5-2 – *Décodage complet.*

trappe $\mathcal{T}$ définie sur tout l'espace), ou alors de changer la construction du schéma de signature.

5-2.2 Utilisation d'un chiffrement surjectif : le schéma CFS_0

Pour rendre le chiffrement de McEliece surjectif il suffit, en conservant le même algorithme et la même matrice génératrice, d'augmenter le nombre d'erreurs que l'on ajoute. Ainsi, on fait grossir le rayon des boules de l'espace d'arrivée jusqu'à ce qu'elles recouvrent tout l'espace (voir figure 5-2). Le rayon minimal nécessaire pour cela est le rayon de recouvrement du code. Si on ajoute un nombre d'erreurs égal au rayon de recouvrement, la fonction de chiffrement devient alors surjective.

Cependant, cette méthode pose plusieurs problèmes. D'une part on ne sait pas calculer la valeur exacte du rayon de recouvrement d'un code de Goppa en temps polynomial. On connaît uniquement des bornes [61] et on sait qu'il est au moins égal à la sphere packing bound δ que l'on calcule ainsi :

$$\delta = \min \left\{ t \;\middle|\; \sum_{i=0}^{t} \binom{n}{i} \geq 2^{n-k} \right\}$$

Même si on ajoute δ erreurs, nous ne serons donc pas certains d'avoir une fonction $\mathcal{E}$ surjective et certains mots pourraient ne pas être décodables. Toutefois, la densité des mots décodables est statistiquement suffisante pour que cela ne pose pas vraiment de problèmes pour la signature.

D'autre part en faisant cela on sort du cadre classique du système de McEliece. Même si, a priori, cela ne change pas grand chose au problème, il n'y a jamais eu d'étude approfondie de la sécurité du système pour un nombre d'erreurs supérieur à celui de la construction classique. On peut imaginer que le fait d'augmenter le nombre d'erreurs et la diversité des chiffrés ne peut qu'augmenter la difficulté des attaques, cependant, le fait qu'un même chiffré puisse correspondre à deux clairs différents risque au contraire de la faire diminuer. Dans la pratique, les attaques sont en général considérées comme difficiles pour un nombre d'erreurs à corriger proche de la borne de Gilbert-Varshamov (voir section 2-1.1 page 22 ou [8, p. 655] pour plus de détails). Si en augmentant le

nombre d'erreurs on ne s'éloigne pas de cette borne on doit pouvoir considérer que la sécurité est maintenue et que cette modification ne représentera pas une faiblesse.

Enfin, et c'est peut-être là le problème le plus important de cette construction, décoder jusqu'au rayon de recouvrement revient à faire ce que l'on appelle un *décodage complet* : être capable de décoder n'importe quel mot de l'espace, y compris les plus éloignés des mots de code. Or, on ne sait a priori pas décoder efficacement au-delà de la capacité de correction construite : la trappe $\mathcal{T}$ ne s'applique qu'aux mots à distance $\leq t$ d'un mot de code. Comme la connaissance de cette trappe est le seul point qui distingue le signataire légitime d'un attaquant, il va donc falloir trouver un algorithme capable de ramener le problème de décodage des δ erreurs au décodage standard avec la trappe. Pour cela, la seule méthode disponible est plus ou moins une recherche exhaustive.

Si on a un algorithme $\mathcal{T}$ capable de décoder t erreurs, on peut alors décoder δ erreurs de la façon suivante :

1. on part d'un chiffré $c = \mathcal{E}(m) + e$, avec e une erreur de poids δ
2. on choisit (aléatoirement) une erreur e' de poids $\delta - t$
3. on essaye de décoder en calculant $\mathcal{T}(c + e')$
4. si cela échoue on retourne à l'étape 2
5. si le décodage réussit on a trouvé un clair m' correspondant au chiffré c (tel que $\mathcal{E}(m')$ soit à distance δ de c).

Pour calculer le nombre moyen d'essais de décodage à effectuer avec cette méthode il faut d'une part calculer le nombre moyen de solutions et d'autre part la probabilité de tomber sur une telle solution. Le volume total des boules de rayon δ centrées sur les mots de code est, pour un code de Goppa de longueur $n = 2^m$:

$$\mathcal{V}_\delta = 2^k \sum_{i=0}^{\delta} \binom{n}{i} \simeq 2^k \binom{2^m}{\delta}.$$

Le nombre moyen de solutions est donc :

$$\mathcal{N}_\delta = \frac{\mathcal{V}_\delta}{2^n} = \frac{2^{m\delta}}{\delta!\, 2^{n-k}} = \frac{2^{m(\delta-t)}}{\delta!}.$$

Pour chaque solution la probabilité de choisir l'erreur e' correctement est :

$$\mathcal{P}_{\delta-t} = \frac{\binom{\delta}{\delta-t}}{\binom{n}{\delta-t}}.$$

Le nombre total de décodages à effectuer en moyenne est donc :

$$\Sigma_{\text{dec}} = \frac{1}{\mathcal{P}_{\delta-t} \times \mathcal{N}_\delta} = \frac{\delta! \binom{n}{\delta-t}}{\binom{\delta}{\delta-t} 2^{m(\delta-t)}} \simeq t!$$

Comme on pouvait s'y attendre le nombre moyen d'essais à effectuer est exactement l'inverse de la densité de mots décodable. Cette méthode ne nous fait donc rien perdre de ce que l'on peut gagner en essayant de maximiser la

densité. Il est aussi important de noter que la valeur choisie pour δ n'intervient pas dans la complexité de la signature : un plus grand δ donnera cependant une signature plus longue avec une probabilité plus faible de tomber sur un haché non décodable. Nous verrons cela plus en détails section 5-4.2 page 65.

Pour plus de clarté, dans la suite de ce chapitre, nous intitulerons cette construction $\mathcal{CFS}_0$.

5-2.3 Modification de la construction : le schéma $\mathcal{CFS}_1$

Avec cette nouvelle méthode, intitulée $\mathcal{CFS}_1$, notre but est de modifier le schéma standard de construction d'un schéma de signature à partir d'un système de chiffrement à clef publique afin de permettre l'utilisation d'une trappe ne s'appliquant qu'à une petite proportion des chiffrés. Puisque certains hachés ne seront pas décodables il faut pouvoir retenter sa chance avec un autre haché. Cependant pour conserver la sécurité du système il est nécessaire que les différents hachés que l'on souhaite essayer soient toujours obtenus de façon non-inversible et sans collisions. Le plus simple pour réussir cela est de modifier le fichier avant de le hacher, par exemple en y ajoutant un compteur. À chaque essai raté il suffit d'incrémenter le compteur, de re-hacher le document et de réessayer de décoder. L'algorithme est le suivant :

1. initialiser le compteur $i = 0$
2. calculer le haché $h_i = \mathcal{M}\mathrm{d}(\mathsf{D}||i)$
3. essayer de décoder en calculant $\mathcal{T}(h_i)$
4. si cela échoue on retourne à l'étape 2
5. si le décodage réussit on a un clair qui peut servir de signature.

Dans le modèle de l'oracle aléatoire, les h_i obtenus de cette façon forment une suite de hachés sans aucune corrélation, dont une proportion $\frac{1}{t!}$ est décodable. Au bout de $t!$ essais en moyenne on doit donc trouver une valeur du compteur qui convient et un haché décodable. Cependant cela oblige à recalculer aussi $t!$ hachés. En ajoutant le compteur à la fin du document D on n'aura à refaire que le dernier tour (ou plutôt *les* derniers tours avec le padding) du hachage mais cela reste un travail de plus à effectuer.

On peut encore essayer d'économiser un peu de temps en n'utilisant pas un véritable hachage pour inclure le compteur. Il faut dans ce cas abandonner le modèle de l'oracle aléatoire et les hachés h_i que l'on produit ne pourront donc pas être considérés comme indépendants. Si on écrit $h_i = f(\mathcal{M}\mathrm{d}(\mathsf{D}), i)$, l'utilisation d'une fonction f beaucoup plus simple qu'un tour de hachage est envisageable, mais il faut être certain que cela n'affectera pas la sécurité de la construction. Les conditions minimales que doit vérifier f sont toutefois difficiles à déterminer précisément.

Remarquons aussi qu'avec cette construction la personne qui doit vérifier la signature aura besoin de connaître la valeur du compteur afin de hacher le document. Il est donc nécessaire d'inclure le compteur dans la signature. Sa valeur n'étant pas bornée dans l'algorithme décrit ci-dessus il faut donc accepter d'avoir une signature de longueur variable : plus longue pour les grandes valeurs du compteur. Si on veut avoir une signature de taille fixe il est nécessaire de

fixer un maximum au compteur : on se retrouve alors dans le même cas que la construction de la section 5-2.2 page 57 où certains messages ne pourront jamais être signés. Bien sûr, en augmentant le maximum on arrive très vite à une probabilité négligeable que cela arrive, en revanche, pour que ce soit le cas, la longueur de la signature (compteur compris) sera nettement plus grande que la longueur moyenne de la signature en taille variable. Ce point sera étudié plus en détail dans la section 5-4.3 page 66.

5-3 Étude de la sécurité de ces constructions

Les constructions $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$ semblent toutes deux faire reposer leur sécurité sur la difficulté d'inverser le chiffrement McEliece. Toutefois, dans les deux cas nous avons été obligés de modifier légèrement le schéma standard. Nous allons voir dans cette section les conséquences que cela peut avoir sur la sécurité du schéma, aussi bien d'un point de vue théorique que d'un point de vue pratique, en regardant comment s'adaptent les meilleures attaques connues.

5-3.1 Réduction de sécurité

Dans cette section nous allons montrer que s'attaquer à ce schéma est au moins aussi dur que de s'attaquer à deux problèmes difficiles bien identifiés : la distinguabilité des codes de Goppa (voir section 3-3.3 page 38) et *Goppa parameterized bounded decoding* (GPBD), un problème dérivé de *syndrome decoding* (voir section 6-1.2.c page 79). Pour cela nous nous plaçons toujours dans le modèle de l'oracle aléatoire afin de ne pas avoir à tenir compte des attaques sur la fonction de hachage.

Considérons donc un attaquant $\mathcal{A}$ qui souhaiterait créer un couple document/signature quelconque sans en avoir le droit (sans connaissance de la clef secrète). La fonction de hachage étant idéale, $\mathcal{A}$ n'aura aucun contrôle sur les hachés et devra donc créer un couple haché/signature à partir d'un haché comme tiré aléatoirement. Supposons maintenant que $\mathcal{A}$ sache attaquer le schéma en temps τ avec une probabilité de succès ε, et montrons qu'il saura alors résoudre l'un des deux problèmes cités.

$\mathcal{A}$ peut attaquer le système en temps τ, donc il ne pourra pas regarder plus de τ hachés différents (obtenus pour des fichiers différents dans le cas de $\mathcal{CFS}_0$ ou pour des valeurs du compteur différentes pour $\mathcal{CFS}_1$). Ces hachés étant aléatoires, cela signifie qu'en moyenne, sur l'ensemble de tous les hachés possibles, $\mathcal{A}$ sait attaquer une instance sur τ pour une clef publique donnée.

Deux cas de figure se présentent alors : soit cette attaque fonctionne avec la même probabilité de succès quelle que soit la matrice de parité $\mathcal{H}$ utilisée en clef publique (et donc aussi pour n'importe quelle matrice aléatoire), soit elle ne fonctionne avec une probabilité ε que pour certaines matrices de parité (dont celles d'un Goppa de support inconnu).

Dans le premier cas, cela signifie que quelle que soit la matrice binaire qu'on donne en entrée de l'algorithme, il est capable de résoudre une instance sur τ avec une probabilité d'au moins ε. Cet algorithme est donc capable de résoudre en temps τ une proportion d'au moins $\frac{\tau}{\varepsilon}$ des instances du problème GPBD.

Dans le deuxième cas, si on se donne une matrice quelconque $\mathcal{H}$ et que l'on essaye d'appliquer l'attaque pour une série de τ hachés, alors si $\mathcal{H}$ est une

matrice de parité d'un code de Goppa la probabilité de succès de l'attaque sera d'au moins ε alors que si ce n'est pas une matrice de Goppa permutée elle sera plus petite. En appliquant cette attaque sur la même matrice avec $\frac{1}{\varepsilon}$ ensembles de τ éléments différents, et donc pour un coût de $\frac{\tau}{\varepsilon}$ on peut différencier une matrice aléatoire d'une matrice de code de Goppa.

En conclusion, quelqu'un qui peut créer des fausses signatures saura soit résoudre une proportion non négligeable d'instances du problème GPBD, soit distinguer une matrice de parité aléatoire d'une matrice de parité de code de Goppa de support inconnu.

5-3.2 Sécurité pratique

Dans cette partie nous cherchons à évaluer la complexité concrète des attaques possibles sur chacun des deux schémas de signatures $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$. Comme nous l'avons vu dans la section précédente, la sécurité du système repose sur deux problèmes difficiles indépendants : il faut donc évaluer le coût des meilleures attaques sur chacun d'eux.

L'attaque structurelle, consistant à retrouver la clef privée à partir de la clef publique sera la même pour $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$: les deux versions utilisent le même couple clef publique/privée. En revanche, pour la création de fausses signatures, la complexité peut être différente dans les deux schémas : pour $\mathcal{CFS}_0$ il faut trouver un mot de poids δ ayant un syndrome donné (c'est exactement le problème de *syndrome decoding*) et pour $\mathcal{CFS}_1$ il faut trouver un compteur et un mot de poids t qui se correspondent.

La dernière attaque à prendre en compte s'applique à tous les schémas de signature : c'est une attaque sur la fonction de hachage. Dans la réduction de sécurité nous utilisons le modèle de l'oracle aléatoire, cependant, la fonction de hachage que l'on utilisera dans la pratique n'est pas un oracle aléatoire et a nécessairement des collisions. Les fonctions de hachages couramment utilisées en signature (SHA-1 en général) sont toutefois très bonnes et la meilleure attaque pour leur trouver des collisions fait toujours appel à une recherche exhaustive s'appuyant sur le paradoxe des anniversaires. Cette attaque a donc un coût très simple à évaluer : si on utilise des hachés de taille r (ici le haché a la longueur d'un syndrome, donc mt), l'attaque aura un coût de $\mathcal{O}\left(r2^{\frac{r}{2}}\right)$ opérations sur des hachés, donc $\mathcal{O}\left(r^2 2^{\frac{r}{2}}\right)$ opérations binaires. Si on veut obtenir, comme toujours, une sécurité de 2^{80} opérations binaires il sera donc nécessaire d'avoir $mt \geq 145$ (notons pour la suite que $mt = 144$ ne sera toutefois pas une réelle faiblesse).

a) Attaque structurelle

Une attaque permettant de retrouver la clef privée à partir de la clef secrète permettra nécessairement de distinguer une matrice de parité permutée de code de Goppa d'une matrice aléatoire. En effet, si on essaye d'appliquer une telle attaque à une matrice aléatoire, ne possédant pas de structure cachée, elle ne pourra pas aboutir. Cela permet donc de décider si une matrice donnée possède cette structure cachée et correspond donc à un code de Goppa. On peut donc minorer le coût de l'attaque structurelle par le coût du meilleur algorithme de distinction d'un code de Goppa.

Cependant, on ne sait que peu de choses de la distinguabilité des codes de Goppa et les seules attaques consistent à énumérer tous les codes de Goppa

possibles (ayant les même paramètres) et à tester leur équivalence avec la clef publique. L'algorithme le plus efficace est le suivant :

Entrée : une matrice de parité $\mathcal{H}$

1. choisir un polynôme g de degré t irréductible,
2. calculer une matrice de parité $\mathcal{H}'$ du code de Goppa généré avec g,
3. tester si les matrices $\mathcal{H}$ et $\mathcal{H}'$ sont équivalentes
4. – si non : retourner à l'étape 1.
   – si oui : trouver $\mathcal{P}$ et $\mathcal{Q}$ tels que $\mathcal{H} = \mathcal{Q}\mathcal{H}'\mathcal{P}$.

Il y a $\frac{2^{tm}}{t} = \frac{n^t}{t}$ polynômes irréductibles unitaires de degré t sur $\mathbb{F}_{2^m}$. On sait donc combien de fois au maximum on répétera la boucle. De plus, le code de Goppa étendu généré par un polynôme $g(z)$ est identique à celui généré par $(cz+d)^t g\left(\frac{az+b}{cz+d}\right)$ si $ad-bc \neq 0$ (voir [63, chap. 12, §4]). On peut donc ne tester qu'une partie des polynômes possibles : pour $n = 2^m$ il y a n^4 valeurs possibles pour (a, b, c, d) mais seules n^3 donnent des polynômes unitaires. De plus, en permutant le support de façon similaire avec le Fröbenius : $z \longrightarrow z^2$ (voir [38]) on peut encore diviser le nombre de polynômes à tester par m. On testera donc au total $\frac{n^{t-3}}{tm}$ polynômes.

Comme nous l'avons vu section 2-1.4 page 24, tester l'équivalence de $\mathcal{H}$ et $\mathcal{H}'$ est un problème qui peut être difficile, mais qui ne l'est quasiment jamais dans la pratique. De façon générale peu de calculs suffisent pour décider que les codes ne sont pas équivalents. Le coût de chaque tour sera quand même au moins celui d'un pivot de Gauss soit $\mathcal{O}\left(n(tm)^2\right)$.

Au bout du compte, pour un code de longueur maximale $n = 2^m$, il est nécessaire d'effectuer $\mathcal{O}\left(tmn^{t-2}\right)$ opérations. Dans tous les cas cette attaque aura donc un coût largement supérieur à celui d'une attaque par paradoxe des anniversaires sur la fonction de hachage. Elle ne sera jamais une vraie menace pour le système.

b) Attaques par décodage

Pour $\mathcal{CFS}_0$, contrefaire une signature revient à trouver un mot de poids δ ayant pour syndrome le haché d'un document donné. Cela revient exactement à résoudre une instance du problème de *syndrome decoding*. Cependant l'attaquant peut utiliser plusieurs documents différents s'il le veut et il lui suffit donc de trouver un mot ayant un syndrome présent dans une liste de hachés.

Pour $\mathcal{CFS}_1$, le problème est très similaire puisque l'on cherche à trouver un mot de poids t ayant pour syndrome un $h_i = \mathcal{M}\mathrm{d}(\mathsf{D}||i)$. On a donc encore une fois à trouver un mot de poids faible donné ayant un syndrome inclus dans une liste de syndromes précalculés. La seule différence est que pour $\mathcal{CFS}_0$ chaque syndrome correspond à plus d'une solution en moyenne alors qu'ici, pour $\mathcal{CFS}_1$, seule une proportion très petite des syndromes correspond effectivement à un mot du bon poids.

Dans les deux cas, il faudra donc résoudre le problème suivant :

List Syndrome Decoding : (LSD)
Entrée : une matrice binaire $\mathcal{H}$, des syndromes binaires $(\mathcal{S}_i)_{i \leq T}$ et un entier w
Propriété : il existe un mot e de poids w et $i_0 \leq T$ tels que $\mathcal{H}e = \mathcal{S}_{i_0}$.

Cependant aucune attaque connue ne permet de tirer profit de la présence de cette liste. Dans le cas de $\mathcal{CFS}_0$ on ne connaît donc pas de meilleure méthode que de choisir un haché arbitrairement dans la liste et d'appliquer la meilleure attaque connue dessus. Pour $\mathcal{CFS}_1$ il faut tenir compte du fait que certains hachés ne pourront pas être signés et la meilleure méthode sera de s'attaquer en parallèle à plusieurs instances avec un algorithme classique, et de s'arrêter dès que la première solution est trouvée. Il est cependant possible d'améliorer l'algorithme pour ne pas avoir à refaire l'ensemble des calculs pour chacun des hachés. Le coût de l'attaque sera donc certainement inférieur au coût de plusieurs attaques complètement indépendantes, mais restera de toute façon toujours au dessus d'une attaque sur une seule instance ayant une solution.

Les meilleures attaques sont expliquées en détail dans la section 6-3 page 82. Le meilleur algorithme est celui de Canteaut-Chabaud [17] dont l'évaluation du coût est relativement compliquée. Dans le cas où on considère un problème pour lequel on sait qu'il existe une solution unique on sait toutefois facilement trouver les paramètres optimaux de l'attaque et en déduire son facteur de travail.

Pour $\mathcal{CFS}_0$ le coût de l'attaque sera donc le coût de la recherche d'un mot de poids δ avec l'algorithme de Canteaut-Chabaud dans un code aléatoire, divisé par le nombre moyen de solutions. Pour $\mathcal{CFS}_1$ il devrait se situer entre le coût de la recherche d'un mot de poids t et ce même coût multiplié par le nombre moyen d'instances du problème à regarder (qui est toujours d'environ $t!$).

Le tableau 5-3 page suivante contient toutes les informations sur le coût de la meilleure attaque sur $\mathcal{CFS}_0$ ou $\mathcal{CFS}_1$ en fonction des paramètres $n = 2^m$ et t choisis. On constate que l'on peut sans problèmes obtenir une sécurité suffisante pour de petites valeurs de t. En particulier, deux jeux de paramètres semblent intéressants : $n = 2^{15}$ et $t = 10$ ou $n = 2^{16}$ et $t = 9$. Le premier jeu de paramètres donne une meilleure sécurité, mais le temps de signature étant lié à $t!$, le deuxième va permettre de signer à peu près 10 fois plus vite. Pour gagner encore sur le temps de signature il faudrait encore faire diminuer t mais déjà avec $t = 8$ il faudra prendre $n \geq 2^{19}$, ce qui rendrait la clef publique (de taille nmt) un peu trop grande. Il semble donc que ce soient les paramètres $(2^{16}, 9)$ qui soient les mieux adaptés. Ces paramètres donnent des syndromes de longueur 144 et donc des hachés de même taille : c'est exactement la taille minimale que l'on s'est fixée pour résister aux attaques par collision sur la fonction de hachage.

5-4 Implémentation

Plusieurs aspects de l'implémentation de chacun des schémas ne sont pas encore clairs. Tout le mécanisme d'inversion de McEliece est bien défini une fois les paramètres choisis, en revanche il reste plusieurs choix pour δ dans $\mathcal{CFS}_0$et pour l'écriture même de la signature et du compteur.

5-4.1 Codage bijectif en mot de poids constant

Aussi bien dans $\mathcal{CFS}_1$ que dans $\mathcal{CFS}_0$, la signature contient un mot binaire e de longueur n et de poids t ou δ fixé. Afin d'avoir une signature courte il va falloir écrire ce mot e de la façon la plus compacte possible, même si cela doit coûter un peu plus cher (en temps de calcul). Si on utilise directement l'écriture

		CFS0				*CFS1*	
		δ	coût	nombre	coût	coût	coût max.
n	t	optimal	CC	solu.	attaque	CC	attaque
2^{14}	7	9	65.3	$2^{9.5}$	55.8	54.5	66.8
	8	10	69.5	$2^{6.2}$	63.3	58.3	73.6
	9	12	80.6	$2^{13.2}$	67.4	64.4	82.9
	10	13	84.9	$2^{9.5}$	75.5	69.4	91.2
2^{15}	7	9	71.2	$2^{11.5}$	59.6	59.2	71.5
	8	10	74.1	$2^{8.2}$	65.9	61.9	77.2
	9	12	86.2	$2^{16.2}$	70.0	70.2	88.7
	10	13	92.8	$2^{12.5}$	80.3	74.6	96.2
2^{16}	7	9	76.3	$2^{13.5}$	62.8	63.3	75.6
	8	10	78.8	$2^{10.2}$	68.6	65.6	80.9
	9	11	88.7	$2^{6.7}$	82.0	76.1	94.6
	10	13	100.6	$2^{15.5}$	85.2	79.2	100.9
2^{17}	7	9	81.4	$2^{15.5}$	65.8	67.4	79.7
	8	10	83.6	$2^{12.2}$	71.4	69.4	84.7
	9	11	95.0	$2^{8.7}$	86.3	81.4	99.8
	10	13	108.4	$2^{18.5}$	90.0	83.8	105.6
2^{18}	7	9	86.5	$2^{17.5}$	68.9	71.5	83.8
	8	10	88.5	$2^{14.2}$	74.3	73.3	88.6
	9	11	101.1	$2^{10.7}$	89.4	86.5	104.9
	10	12	103.5	$2^{7.2}$	96.3	88.6	110.4
2^{19}	7	9	91.6	$2^{19.5}$	72.0	75.5	87.8
	8	10	93.4	$2^{16.2}$	77.2	77.2	92.5
	9	11	107.2	$2^{12.7}$	94.5	91.5	110.0
	10	12	109.3	$2^{9.2}$	100.2	93.5	115.3

TAB. 5-3 – *Logarithmes des facteurs de travail (en opérations binaires) de l'algorithme de Canteaut-Chabaud en fonction des différents paramètres. Voir section 5-4.2 page ci-contre pour le choix du δ optimal.*

binaire de e on obtient une signature de n bits, ce qui n'est absolument pas satisfaisant. On peut à la place uniquement écrire les indices des positions non nulles de e. On obtient alors une signature de mt ou $m\delta$ bits : c'est mieux mais pas encore assez.

En effet, il y a exactement $\binom{n}{t}$ mots de longueur n et poids t. En les numérotant et en utilisant comme signature l'indice du mot trouvé on peut obtenir une signature de longueur $\log_2 \binom{n}{t}$ ou $\log_2 \binom{n}{\delta}$. Pour calculer cet indice, si on note $(i_1, \ldots, i_t)$ les positions des bits non nuls de e (avec $i_1 < i_2 < \ldots < i_t$) on peut alors simplement calculer l'indice I_e de e :

$$I_e = \binom{i_1}{1} + \binom{i_2}{2} + \ldots + \binom{i_t}{t}.$$

Malheureusement, en pratique ce calcul n'est pas si simple qu'il n'y parait car il fait en effet nécessairement appel à des calculs sur des grands entiers. Pour la signature cela ne pose pas de problèmes puisque cette étape répétée une seule fois aura de toute façon un coût négligeable par rapport au déchiffrement répété $t!$ fois. En revanche cela peut ralentir la vérification de la signature : il faut donc trouver un algorithme le plus économique possible pour reconvertir cet indice en mot de poids constant.

La méthode basique consiste à d'abord chercher le plus grand $\binom{i_t}{t}$ plus petit que I_e qui nous donne l'indice i_t, puis on calcule $I_e - \binom{i_t}{t}$ et on cherche de la même façon i_{t-1} et ainsi de suite. On peut améliorer cela pour ne pas avoir à faire appel à des calculs directs de binomiaux [42, 25] mais uniquement des divisions sur des grands entiers.

Enfin, on peut aussi précalculer tous les coefficients binomiaux que l'on risque de rencontrer (tous les $\binom{i}{j}$ pour $i = 0 \ldots n-1$ et $j = 1 \ldots t$) et procéder par dichotomie. Cette dernière méthode sera certainement plus efficace si on doit faire beaucoup de vérifications et présente l'avantage de ne pas du tout nécessiter de divisions ou de multiplications lors du recodage du mot ou des précalculs (si on calcule les binomiaux avec le triangle de Pascal). De plus, pour $\mathcal{CFS}_1$, avec les paramètres $(2^{16}, 9)$, la longueur de l'indice est $\log_2 \binom{2^{16}}{9} = 125.5$ et peut donc s'écrire sur un mot machine de 128 bits. Une implémentation logicielle du codage en mot de poids constant pourra donc se contenter de faire des comparaisons et des additions de mots 128 bits, ce qui peut être très rapide. Toutefois, avec ces paramètres, la table à garder en mémoire a une taille d'environ 8Mo : cette méthode ne sera donc valable que sur des architectures disposant d'une quantité suffisante de mémoire.

5-4.2 Le choix de δ pour $\mathcal{CFS}_0$

Nous avons vu que la valeur choisie pour δ n'intervenait ni pour la sécurité du système (tant qu'on lui conserve une petite valeur), ni pour le coût de la signature. En revanche, plus il est choisi grand, plus le mot à coder dans la signature sera de poids élevé, et donc comme vu précédemment, plus la signature sera longue. De même, plus δ est grand, plus la probabilité de tomber sur un document dont le haché ne peut être décodé (et qui ne peut donc pas être signé) sera petite. Il faut donc prendre le plus petit δ possible, garantissant une probabilité d'échec suffisamment petite (en dessous de 2^{-80} par exemple). Le ta-

δ	9	10	11	12
$\mathcal{P}_{\text{echec}}$	0.999997	0.982	2^{-155}	$2^{-846915}$
longueur (bits)	126	139	151	164

TAB. 5-4 – *Probabilité de ne pas pouvoir signer un document et longueur de la signature $\mathcal{CFS}_0$ pour différentes valeurs de δ, avec $n = 2^{16}$ et $t = 9$.*

bleau 5-4 contient les longueurs de signatures et les probabilités correspondantes à différents choix de δ pour les paramètres choisis $(2^{16}, 9)$.

Pour cette table, la probabilité d'échec a été calculée en supposant que tous les mots décodables étaient choisis aléatoirement, un par un, dans l'espace des mots. Dans la pratique, les mots sont rassemblés en boules autour des mots de code et ne sont pas dispersés au hasard. Cependant les mots du code étant, eux, bien répartis dans l'espace pour respecter la distance minimale, l'estimation ne doit pas être trop fausse. On constate que pour $\delta = 11$ la probabilité de ne pas pouvoir signer est déjà négligeable. Il est donc inutile d'utiliser une valeur plus grande et c'est donc cette valeur qu'il convient d'utiliser pour $\mathcal{CFS}_0$. Elle donne des signatures de 151 bits (et de longueur fixe).

5-4.3 Codage du compteur pour $\mathcal{CFS}_1$

L'autre point d'implémentation nécessitant un choix est la façon d'inclure le compteur dans la signature. Trois choix s'offrent à nous, plus ou moins valables selon les situations. Dans tous les cas nous considérerons que l'on sait exactement où s'arrête le document que l'on a signé et où commence la signature. De même, l'indice I_e du mot de poids t obtenu étant de longueur fixe (126 bits pour les paramètres que l'on a choisi), on sait exactement où commence l'écriture du compteur z. Ce qu'on ne sait pas, en revanche, c'est où elle s'arrête...

a) Compteur de taille variable

Ce premier choix est bien adapté dans le cas où le fichier est destiné à être stocké sur un disque ou n'importe quel système de fichier standard où la taille totale des fichiers est connue : si on connaît la position de début du compteur et la longueur totale du fichier on sait quelle longueur fait le compteur. On peut donc simplement écrire l'indice sans aucun en-tête et sans terminaison : par exemple pour un compteur $z = 13$ on pourra simplement ajouter 1101 à la fin de la signature. La longueur moyenne du compteur est alors 18.1 bits pour $t = 9$.

b) Compteur de taille fixe

Ce deuxième choix correspond aux conditions où l'on a le plus de contraintes : par exemple si on doit écrire la signature dans un registre de carte à puce de longueur fixée. Dans ce cas on ne peut en aucun cas se permettre d'avoir une signature dont la taille dépasse la capacité du registre et il est nécessaire de fixer la taille. Si on fixe par exemple le compteur à une longueur de 7 bits (bien sûr ce n'est pas une taille réaliste pour nos paramètres) on aurait pour $z = 13$ une écriture 0001101.

Cette méthode présente cependant deux inconvénients : d'une part la signature sera plus longue que la taille moyenne avec une autre méthode et d'autre

l	18	19	20	21	22	23	24	25
$\mathcal{P}_{\text{echec}}$	0.49	0.24	0.06	$2^{-8.3}$	$2^{-16.7}$	$2^{-33.3}$	$2^{-66.7}$	$2^{-133.4}$

TAB. 5-5 – *Probabilité de ne pas pouvoir signer un message en fonction de la longueur l fixée pour l'écriture du compteur, pour les paramètres $n = 2^{16}$ et $t = 9$.*

part cela fixe une valeur maximale pour l'indice à utiliser, et cela introduit donc une probabilité de ne pas pouvoir signer un document. Le tableau 5-5 donne cette probabilité pour $n = 2^{16}$ et $t = 9$: la probabilité d'échec est élevée au carré à chaque bit ajouté et tend donc très vite vers 0. Un compteur de 24 ou 25 bits est cependant nécessaire.

c) *Compteur de taille variable à terminaison décidable*

La dernière solution est une solution intermédiaire : on a une signature de taille variable, mais en la lisant on connaît exactement le moment où l'écriture du compteur est terminée. C'est l'écriture qui sera la mieux adaptée pour un système où la signature est transmise dans un flot d'information où la taille des messages est fixe : on connaît le début de la signature et si on sait quand elle s'arrête on pourra connaître le début du message suivant sans avoir à utiliser d'en-tête supplémentaire.

Ce problème est un problème classique de codage de source : on veut coder en binaire, et de la façon la plus courte, tous les entiers $i \in \mathbb{N}$ munis d'une loi de probabilité $\mathcal{P}(i)$. On sait qu'un codage optimal donnera une taille moyenne égale à l'entropie de la source et qu'on ne peut pas faire mieux. Dans notre cas, $\mathcal{P}(i) = p(1-p)^{i-1}$ avec $p = \frac{1}{t!}$ et l'entropie est :

$$H = \sum_{i \geq 1} \mathcal{P}(i) \log_2 \frac{1}{\mathcal{P}(i)} = \frac{p \log_2 p + (1-p) \log_2 (1-p)}{p} \approx 19.91$$

On sait donc qu'il n'existe pas de codage utilisant moins de 19.9 bits en moyenne. Pour essayer de s'en approcher au mieux on va utiliser la technique suivante : on fixe un nombre de bits λ (il faudra trouver la valeur optimale de ce paramètre) et si le compteur est plus grand que 2^λ on écrit un 1 et on soustrait 2^λ au compteur. On continue ainsi jusqu'à avoir un compteur plus petit que λ et on écrit alors un 0, suivi de l'écriture sur λ bits du compteur restant. Il suffit donc de faire une division entière de i par 2^λ : $i = q2^\lambda + r$ et l'écriture sera :

$$i \longmapsto \overbrace{1 \ldots 1}^{q}\, 0\, \overbrace{\boxed{r}}^{\lambda}$$

On sait donc que l'on peut arrêter de lire le compteur λ bits après avoir lu le premier 0. Si on fixe par exemple $\lambda = 3$ on écrira $z = 13$ comme 1 0 101. Pour $\lambda = 2$ on l'écrirait à la place 111 0 01. Pour $t = 9$, le tableau 5-6 page suivante permet de constater que le meilleur λ donne un codage très proche de l'optimal et rallonge donc la signature de moins de 2 bits en moyenne : on écrit le compteur sur 19.94 bits au lieu de 18.1.

λ	12	13	14	15	16	17	18	19	20
Nb bits	101.1	57.8	36.7	26.6	22.1	20.3	19.94	20.3	21.1

TAB. 5-6 – *Longueur moyenne du compteur (en bits) en fonction de λ pour $t = 9$ en utilisant le codage en taille variable à terminaison décidable.*

5-5 Raccourcir la signature

Dans la section précédente nous avons vu exactement comment, dans la pratique, la signature devait être écrite. Selon le schéma utilisé et les différents choix faits on peut obtenir des longueurs de signatures très différentes. On trouve :

- pour $\mathcal{CFS}_0$: 151 bits,
- pour $\mathcal{CFS}_1$ en taille variable : 144.1 bits en moyenne,
- pour $\mathcal{CFS}_1$ en taille fixe : 150 ou 151 bits,
- pour $\mathcal{CFS}_1$ en taille variable à terminaison décidable : 146 bits en moyenne.

Notons que les schémas $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$ en taille fixe sont quasiment identiques, aussi bien pour la longueur que pour la probabilité d'échec. Cependant, comme nous allons le voir dans cette section, une signature effectuée en utilisant $\mathcal{CFS}_0$ pourra se compresser un peu mieux.

En effet, les signatures obtenues avec ces deux constructions font partie des plus petites connues, beaucoup plus petites que des signatures RSA fournissant une sécurité équivalente. Il est toutefois possible de faire encore mieux : dans cette section nous allons voir comment, en augmentant le coût d'une vérification, nous pouvons encore diminuer la taille de ces signatures.

5-5.1 Le compromis standard

Dans tout système de signature, il est possible de raccourcir la signature transmise en augmentant la part du travail faite par le vérificateur. On peut transmettre un bit de moins et au moment de la vérification il suffit d'essayer les deux valeurs possibles pour ce bit : si l'une des deux possibilités correspond à une signature valide, la signature sera acceptée.

Avec cette technique on peut ainsi diminuer la longueur de la signature transmise de ζ bits en multipliant le coût de la vérification par 2^ζ. Ce compromis est donc très coûteux mais dans le cas de $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$ la vérification est suffisamment rapide pour que cela puisse en valoir la peine. De plus, comme nous allons le voir section 5-5.2 page ci-contre, il y a moyen de faire cela beaucoup plus efficacement pour ces systèmes.

Sécurité d'un tel compromis

Du point de vue de la sécurité on peut imaginer que transmettre une signature plus courte, ne contenant pas toute l'information nécessaire à la vérification, puisse simplifier la tâche d'un attaquant. Il n'en est pourtant rien : tant que la vérification passe par la reconstitution de la signature complète les deux attaques auront un coût similaire.

Supposons que le temps nécessaire à la vérification de la signature raccourcie soit $\mathcal{T}_{\text{vérif}}$ et le temps nécessaire à la réalisation d'une fausse signature complète $\mathcal{T}_{\text{comp}}$. Un attaquant capable de réaliser une fausse signature raccourcie en temps

$\mathcal{T}_{\text{court}}$ peut ensuite simplement essayer de vérifier sa fausse signature et obtient alors une fausse signature en longueur totale. On a donc nécessairement :

$$\mathcal{T}_{\text{court}} + \mathcal{T}_{\text{vérif}} \geq \mathcal{T}_{\text{comp}},$$

et donc :

$$\mathcal{T}_{\text{court}} \geq \mathcal{T}_{\text{comp}} - \mathcal{T}_{\text{vérif}}.$$

Tant que le temps nécessaire à la vérification reste plus petit que le temps d'une attaque (ce qui semble être assez nécessaire), le coût des deux sortes d'attaques est donc identique.

5-5.2 Transmettre moins de positions d'erreur

Dans le cas des signatures $\mathcal{CFS}_0$ et $\mathcal{CFS}_1$ il est possible d'utiliser le compromis précédent de façon beaucoup plus rentable. Au lieu d'omettre quelques bits qui n'ont pas réellement de sens pour la signature nous allons diminuer le poids des mots transmis : par exemple pour $\mathcal{CFS}_0$, au lieu de donner les indices des δ positions nécessaires à la vérification, le signataire en transmettra un peu moins. En faisant cela, le vérificateur aura toujours une recherche exhaustive à effectuer, mais cette recherche étant liée à la structure même du système elle pourra aller beaucoup plus vite.

a) En oubliant une position

Si le signataire n'enlève qu'une erreur, le vérificateur a à sa disposition le document D et les indices de toutes les positions non nulles sauf une (plus le compteur pour $\mathcal{CFS}_1$). Il suffit alors de hacher le document ($\mathcal{M}\text{d}(\mathsf{D})$ ou $\mathcal{M}\text{d}(\mathsf{D}||i)$) et d'y ajouter les $t-1$ (ou $\delta-1$ pour $\mathcal{CFS}_0$) colonnes de la matrice de parité $\mathcal{H}$ (la clef publique) correspondantes. Si la signature est valide, la somme que l'on vient de calculer doit être égale à une colonne de $\mathcal{H}$. Il suffit donc d'avoir rangé la matrice dans une table de hachage appropriée pour pouvoir effectuer cette vérification en temps constant.

Transmettre une position de moins ne va donc pas augmenter le travail du vérificateur qui aura juste à avoir effectué quelques précalculs. La taille de la signature va cependant elle bien diminuer. Pour $\mathcal{CFS}_0$ on passe de 151 à $\log_2 \binom{n}{\delta-1} = 138.2$ bits (12 bits gagnés) et pour $\mathcal{CFS}_1$ de 144 bits en moyenne à $\log_2 \binom{n}{t-1} + 18.1 \approx 131$ bits en moyenne (13 bits gagnés, quelle que soit la façon d'écrire le compteur).

b) En oubliant deux positions

Si le signataire omet maintenant 2 des indices des positions non nulles, on va procéder de la même façon que précédemment : on calcule le haché et on y ajoute les colonnes connues et on cherche dans la matrice $\mathcal{H}$ une somme de 2 colonnes qui y soit égale. L'idéal serait de pouvoir précalculer toutes les combinaisons de 2 colonnes et les ranger dans une table de hachage, mais pour $n = 2^{16}$ cela ferait une table de plusieurs centaines de gigabits. On peut à la place faire une recherche exhaustive : on ajoute une colonne de la matrice au haché et colonnes connues et on cherche si cette nouvelle colonne est dans la matrice, cette fois avec la même méthode que pour une seule position omise. Avec cette technique, le coût d'une vérification devient alors essentiellement

positions omises	coût de la vérification		longueur $\mathcal{CFS}_0$	$\mathcal{CFS}_1$	$\mathcal{CFS}_1$ fixe	$\mathcal{CFS}_1$ termin.
0	t	9	151	144	151	146
1	t	9	139	131	138	133
2	$2\frac{n}{t}$	2^{14}	126	118	125	120
3	$3\binom{n}{2}/\binom{t}{2}$	2^{27}	113	105	112	107
4	$3\binom{n}{3}/\binom{t}{3}$	2^{40}	100	91	99	93

TAB. 5-7 – *Complexité de la vérification (en nombre d'opérations sur les colonnes) et taille des signatures en fonction du nombre w d'erreurs omises.*

celui de la recherche exhaustive, avec donc, en moyenne, $\frac{n}{2}$ XORs de colonnes et $\frac{n}{2}$ recherches dans la table de hachage.

Si de plus le signataire choisit d'oublier toujours les positions qui ont le plus petit indice, le vérificateur peut alors faire sa recherche exhaustive dans l'ordre et le nombre moyen d'essais à effectuer passe alors de $\frac{n}{2}$ à $\frac{n}{t+1}$. Avec cela, les signatures ont alors une longueur de 126 bits pour $\mathcal{CFS}_0$ et 118 bits en moyenne pour $\mathcal{CFS}_1$.

c) En oubliant plus de positions

En omettant plus de positions la taille va encore diminuer mais le coût de la recherche exhaustive va augmenter très vite. En effet, pour chercher les w premières positions exhaustivement il faut de l'ordre de $\mathcal{O}\left(\frac{n^w}{t^w}\right)$ opérations sur les colonnes. En revanche, la taille diminue elle aussi assez vite. Le tableau 5-7 contient les tailles et les coûts de vérification en fonction du nombre d'erreurs omises.

On constate que l'on peut, si on accepte que la vérification dure quelques secondes, obtenir des signatures juste au dessus de 100 bits. En revanche, à part sur la première étape, le gain n'est pas tellement meilleur qu'avec le compromis standard : on gagne 13 bits pour une vérification 2^{13} fois plus longue.

5-5.3 Partitionner le support

Cette méthode va consister à encore supprimer de l'information de la signature, mais cette fois, en ajoutant un temps constant à la vérification, quel que soit le nombre de positions retirées. On va accepter de perdre un peu d'information pour chaque position transmise en ne donnant pas exactement l'indice de cette position, mais un intervalle dans lequel elle se trouve.

a) Fonctionnement

On partitionne le support en $\frac{n}{\ell}$ blocs de ℓ positions chacun et on ne transmet que l'indice du bloc. Ainsi, la taille de la signature ne sera plus $\log_2\binom{n}{t}$ mais $\log_2\binom{n/\ell}{t}$. Pour vérifier la signature il faudra vérifier qu'il existe un mot ayant le bon syndrome, avec un 1 dans chaque bloc dont on connaît l'indice et un poids total de t. Cette opération peut se faire relativement facilement en procédant comme sur la figure 5-8 page ci-contre :

– on regroupe tous les blocs de colonnes dont on connaît les indices en début de matrice,

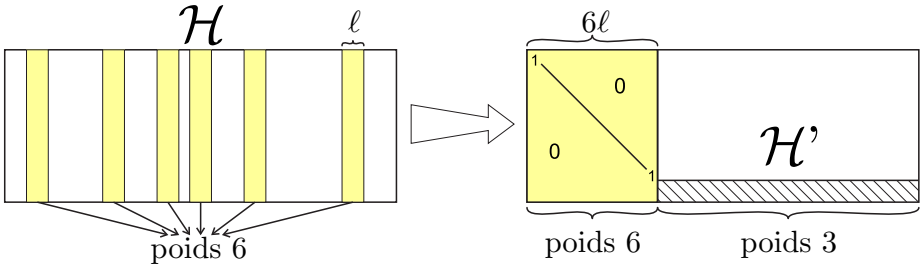


FIG. 5-8 – *Vérification de la signature avec partition du support dans le cas où on ne transmet que 6 positions sur 9.*

- on effectue un pivot de Gauss sur la matrice pour obtenir l'identité dans les $t-w$ colonnes de gauche (on applique le pivot sur le haché du document en même temps),
- reste à trouver w colonnes de $\mathcal{H}'$ ayant pour somme la fin du haché pivoté et telles que la somme de ces w colonnes avec le haché soit de poids 6.

Le travail à effectuer est donc exactement le même que précédemment, avec le même avantage procuré par l'omission de colonnes du début de la matrice, mais sur une matrice un petit peu plus petite (de taille $n-(t-w)\ell$ par $mt-(t-w)\ell$), plus un pivot de Gauss sur toute la matrice. Ce pivot aura un coût de l'ordre de $(mt)^2 n$ opérations binaires soit en gros le même coût que mnt opérations sur des colonnes, soit pour nos paramètres 2^{24} opérations de colonnes. Cette technique sera donc surtout avantageuse dans le cas où l'on oublie 3 positions puisque le coût du pivot sera inférieur à celui de la recherche des positions manquantes.

Notons toutefois que selon la valeur de ℓ la matrice $\mathcal{H}'$ peut n'avoir que très peu de lignes. Cela aurait pour effet de faire augmenter le nombre de solutions de w colonnes dans $\mathcal{H}'$ et donc le coût total de l'algorithme puisque pour chaque solution trouvée dans $\mathcal{H}'$ il faut vérifier si l'on obtient un mot de poids 6 dans la matrice complète, ce qui est relativement coûteux. Il sera donc préférable de conserver ℓ suffisamment petit pour que le nombre moyen de solutions dans $\mathcal{H}'$ reste voisin de 1.

b) Gain de taille

Le choix de ℓ est un problème assez délicat puisque, comme nous l'avons vu, un grand ℓ risque de nuire à l'efficacité de la recherche exhaustive et en même temps, un grand ℓ permet de mieux réduire la taille de signature. De façon générale, si on prend $\ell = m = \log_2 n$, la matrice $\mathcal{H}'$ aura mw lignes, ce qui correspond exactement au cas où il y a en moyenne une solution indésirable. Une valeur plus petite ne sera donc pas optimale du point de vue du gain de taille, et une valeur plus grande fera nécessairement augmenter le temps de vérification en faisant augmenter les collisions inutiles.

Le tableau 5-9 page suivante contient la mise à jour des tailles de signature et des coûts de vérification en utilisant le partitionnement du support en blocs de taille m. On constate que l'on peut atteindre des tailles de signature de l'ordre de 80 bits, toujours avec un temps de vérification de quelque secondes. Notons aussi qu'avec cette modification le schéma $\mathcal{CFS}_0$ donne des signatures de taille fixe de longueur égale à celle obtenues avec $\mathcal{CFS}_1$ en taille variable.

positions omises	coût de la vérification	longueur			
		$\mathcal{CFS}_0$	$\mathcal{CFS}_1$	$\mathcal{CFS}_1$ fixe	$\mathcal{CFS}_1$ termin.
0	2^{24}	107	108	115	110
1	2^{24}	99	99	106	101
2	2^{24}	90	90	97	92
3	2^{27}	81	81	88	83
4	2^{40}	72	72	79	74

TAB. 5-9 – *Coût de la vérification (en nombre d'opérations sur les colonnes) et taille des signatures en fonction du nombre w d'erreurs omises, en partitionnant le support en blocs de taille $\ell = m$.*

5-6 Comportement asymptotique

Avec les paramètres choisis $(2^{16}, 9)$, la signature a un coût de 9! décodages dans le code, ce qui prend environ 1 minute sur un pentium 4 à 2.6 GHz avec une implémentation peu optimisée. Comme nous l'avons vu ce sont les paramètres minimum pour avoir une sécurité suffisante. Si la sécurité attendue avait été supérieure, le coût d'une signature aurait certainement été trop élevé et aurait rendu le schéma impraticable. C'est peut-être aussi une raison pour expliquer que cette méthode n'ait pas été essayée avant : si on remonte 10 ans en arrière la puissance de calcul étant beaucoup plus faible, la sécurité obtenue avec un schéma praticable alors n'aurait de toute façon pas été suffisante. Au contraire, si on regarde vers le futur, un système offrant une sécurité adaptée aux besoins à venir correspondra à des paramètres donnant un temps de signature plus court, ce qui est plutôt bon signe pour le passage à l'échelle.

Le tableau 5-10 page ci-contre donne les équivalents asymptotiques des différentes valeurs entrant en jeu dans les schémas de signature proposés. On constate que seule la sécurité est exponentielle en le produit mt. En faisant donc grandir les valeurs de m et t en parallèle, la sécurité augmentera très vite alors que le temps de signature (exponentiel en t) et la taille de clef (exponentielle en m) ne grandiront que modérément vite. De même, en raccourcissant la signature le temps de vérification devient vite exponentiel en m, mais reste polynomial en t.

Les paramètres m et t sont donc à faire varier avec précautions mais il n'est pas difficile d'obtenir le temps de signature ou la taille de clef voulue pour une sécurité donnée. Les meilleurs compromis seront en revanche toujours pour des valeurs de m et t assez voisines (comme ici 9 et 16).

Si on vise une sécurité plus élevée comme par exemple 2^{100} on peut facilement redimensionner les paramètres, même si le résultat devient peu praticable avec les moyens actuels. En effet, cela nous oblige à choisir mt plus grand que 184 environ. On peut alors choisir $t = 10$ et $m = 19$ pour un temps de signature d'une dizaine de minutes et une clef de 95Mbits, ou alors $t = 9$ et $m = 21$ pour garder un temps de signature d'une minute et avoir une clef de 378Mbits. Dans les deux cas on arrivera quand même à obtenir des signatures de l'ordre de 110 bits mais pour un temps de vérification de plusieurs minutes. Avec de telles longueurs on se rend compte qu'on ne pourra pas autant réduire la taille de signature, ou alors il faut garder des valeurs de t et m plus équilibrées, ce qui va rendre le temps de signature beaucoup plus grand.

	attaque par décodage	$2^{tm(\frac{1}{2}+o(1)}$
	attaque structurelle	$tm2^{m(t-2)}$
	temps de signature	$t!t^2m^3$
	taille de clef	$tm2^m$
$\mathcal{CFS}_1$	coût de vérification	t^2m
	taille de signature	tm
$\mathcal{CFS}_1-1$	coût de vérification	t^2m
	taille de signature	$(t-1)m+\log_2 t$
$\mathcal{CFS}_1-2$	coût de vérification	$m2^m$
	taille de signature	$(t-2)m+2\log_2 t$
$\mathcal{CFS}_1-2+P$	coût de vérification	$t^2m^22^m$
	taille de signature	$\log_2 \binom{\frac{2^m}{m}}{t-2}$
$\mathcal{CFS}_0-3+P$	coût de vérification	$\frac{2^{2m}}{t^2}+t^2m^22^m$
	taille de signature	$\log_2 \binom{\frac{2^m}{m}}{\delta-3}$

TAB. 5-10 – *Équivalents asymptotiques des différentes grandeurs entrant en jeu dans les deux schémas en fonction de m et t quand on utilise un support de taille maximale $n = 2^m$. La colonne de gauche indique le schéma utilisé : $\mathcal{CFS}_0$ ou $\mathcal{CFS}_1$, $-i$ indique le nombre i de positions omises et $+P$ indique le partitionnement du support en blocs de taille m.*

5-7 Conclusion

Nous avons vu dans ce chapitre deux schémas de signature très voisins donnant des signatures très courtes et très rapides à vérifier. Nous avons ensuite vu deux méthodes pour encore raccourcir les signatures obtenues au détriment du temps de vérification. Pour obtenir une sécurité satisfaisante de 2^{80} opérations nous avons vu que les paramètres $m = 16$ et $t = 9$ convenaient. Parmi les compromis possibles il y en a quatre qui semblent particulièrement intéressants pour ces paramètres, chacun donnant des signatures un peu plus courtes et un temps de vérification un peu plus long.

a) $\mathcal{CFS}_1$-1

Il correspond au cas où on conserve un temps de vérification le plus court possible en enlevant juste une position de la signature avant de la transmettre.

vérification	$\sim 1\mu$s
signature	131 bits (en moyenne)

b) $\mathcal{CFS}_1$-2

On accepte maintenant de prendre un peu plus de temps pour vérifier la signature et on enlève les deux erreurs du début.

vérification	$\sim$1ms
signature	118 bits (en moyenne)

c) $\mathcal{CFS}_1$-2+$\mathcal{P}$

C'est le même schéma que précédemment mais avec le partitionnement du support en plus. Le coût de la vérification sera donc essentiellement celui du pivot initial, la recherche exhaustive qui le suit est négligeable.

vérification	$\sim$1s
signature	90 bits (en moyenne)

d) $\mathcal{CFS}_0$-3+$\mathcal{P}$

Pour avoir les signatures les plus courtes on enlève 3 positions et on partitionne le support. Maintenant c'est le coût du pivot qui devient négligeable et seule la recherche compte. De plus, le schéma $\mathcal{CFS}_0$ permet d'obtenir des signatures aussi courtes que la moyenne des signature $\mathcal{CFS}_1$. On utilise donc plutôt $\mathcal{CFS}_0$ qui sera plus simple à mettre en place dans la plupart des cas.

vérification	$\sim$20s
signature	81 bits (toujours)

Avec des signatures de 81 bits on obtient les signatures les plus courtes connues à ce jour et offrant une sécurité acceptable. Cela vient toutefois au prix d'une vérification relativement longue, d'un temps de signature non négligeable et d'une taille de clef assez conséquente. Notons aussi que le temps de vérifier la signature sera encore beaucoup plus long s'il faut vérifier que la signature est fausse : on perd l'avantage procuré par le choix d'omettre les premières positions du support et la vérification prends alors plusieurs minutes avec $\mathcal{CFS}_0$-3+$\mathcal{P}$.

Troisième partie

Autour du problème de Syndrome Decoding

Chapitre 6

Le problème de Syndrome Decoding

Au cœur de la plupart des cryptosystèmes utilisant des codes correcteurs d'erreurs, on retrouve un même problème difficile : *syndrome decoding* (SD). Comme l'ont montré Berlekamp, McEliece et van Tilborg en 1978 [12] ce problème est NP-complet. C'est en partie sur sa difficulté que repose la sécurité de plusieurs constructions comme les systèmes de McEliece [66] ou Niederreiter [70], ou comme nous allons le voir dans les deux prochains chapitres, à peu près tout nouveau système. Ce qui en fait un problème de choix pour les applications cryptographiques des codes correcteurs d'erreurs n'est pas seulement le fait d'être NP-complet, mais aussi le fait que, en général, une instance aléatoire de ce problème est toujours difficile. Il est compliqué de prouver des résultats précis à ce sujet, mais il semblerait qu'en termes de complexité « en moyenne », ce problème soit presque toujours difficile. C'est-à-dire que l'on doit pouvoir définir une mesure sur l'ensemble des instances pour laquelle celles qui sont faciles représentent un volume aussi petit que l'on veut.

De plus, comme nous allons le voir dans la suite de ce chapitre, cette bonne densité d'instances difficiles fait que beaucoup de sous-problèmes, consistant en un groupe d'instances particulières du problème SD, sont aussi majoritairement constitués d'instances difficiles. Pour certains on peut même refaire une preuve de NP-complétude similaire à celle de [12]. De cette façon, on peut faire reposer la sécurité d'un système sur un nombre restreint d'instances particulières du problème SD sans pour autant prendre un grand risque de n'avoir que des instances faciles.

Enfin, et c'est là l'argument le plus fort quand à la difficulté du problème de *syndrome decoding*, depuis que ce problème est étudié (cela fait déjà quelques dizaines d'années) aucun algorithme sous exponentiel n'a été trouvé. Cet argument peut ne pas paraître très fiable, mais c'est le même qui nous pousse à croire que factoriser de grands entiers est un problème difficile, alors même que des algorithmes bien plus efficaces existent pour factoriser que pour résoudre le problème SD. La dernière partie de ce chapitre recense quelques unes des attaques les plus connues sur ce problème et plus particulièrement celles basées sur des ensembles d'information.

6-1 Description

6-1.1 Syndrome Decoding

Le problème de *syndrome decoding* s'énonce de la façon suivante :

Syndrome Decoding (SD)
Entrée : une matrice binaire $\mathcal{H}$ de taille $r \times n$, un syndrome $\mathcal{S}$ binaire de longueur r et un entier w,
Propriété : il existe une erreur e de poids $\leq w$ telle que $\mathcal{H}e = \mathcal{S}$.

On peut rencontrer ce problème sous le nom de *Coset Weight*, puisqu'il consiste à chercher un mot de petit poids dans un coset du code. Dans tous les cas, c'est un énoncé assez général qui englobe une grande partie des problèmes de décodage. Il faut noter qu'il correspond au problème de décodage borné (trouver un mot à distance plus petite qu'une borne donnée) et non pas au problème de trouver le mot le plus proche, qui lui n'est pas dans NP puisqu'il est impossible de vérifier qu'un mot donné est le plus proche en temps polynomial.

La preuve de NP-complétude utilise une réduction au problème de Three-Dimensional Matching qui s'énonce comme suit :

Three-Dimensional Matching : (3DM)
Entrée : un ensemble fini T et $U \subseteq T \times T \times T$.
Propriété : il existe $V \subseteq U$ tel que $|V| = |T|$ et aucune paire d'éléments de V n'a de coordonnées communes.

Vu sous cet angle ce problème ne parait pas très proche du problème de décodage. En revanche, si on regarde un exemple on constate qu'on peut le réécrire de façon beaucoup plus appropriée. Considérons par exemple : $T = \{1, 2, 3\}$ et $|U| = 5$

$$\begin{aligned} U_1 &= (1, 2, 2) \\ U_2 &= (2, 2, 3) \\ U_3 &= (1, 3, 2) \\ U_4 &= (2, 1, 3) \\ U_5 &= (3, 3, 1) \end{aligned}$$

On peut voir qu'un ensemble V constitué de U_1, U_4 et U_5 vérifie la propriété. En revanche, dès que l'on retire U_1 de U il n'existe plus aucune solution. Dans notre cas il sera plus simple de considérer ce problème de la façon suivante : on associe une matrice d'incidence A binaire de taille $3|T| \times |U|$ à l'instance. Sur notre exemple cela donne la matrice de la figure 6-1 page suivante.

Une solution au problème est un sous-ensemble de $|T|$ colonnes dont la somme est le vecteur tout à 1. Si on a un algorithme capable de résoudre n'importe quelle instance de SD il est alors clair qu'il sera capable de résoudre une telle instance de 3DM. De ce fait SD est lui aussi NP-complet.

6-1.2 Problèmes proches

On peut écrire une multitude de sous-problèmes de SD correspondant à certains types d'instances particulières à résoudre. Ceux qui suivent font partie des problèmes que l'on rencontre le plus fréquemment. On trouvera section 8-4 page 112 deux autres exemples de sous-problèmes pour lesquels on a ajouté

$A =$

	122	223	132	213	331
1	1	0	1	0	0
2	0	1	0	1	0
3	0	0	0	0	1
1	0	0	0	1	0
2	1	1	0	0	0
3	0	0	1	0	1
1	0	0	0	0	1
2	1	0	1	0	0
3	0	1	0	1	0

FIG. 6-1 – *Matrice d'incidence associée à une instance de 3DM*

une contrainte au problème original : ces deux problèmes sont aussi NP-complets et servent à étayer la sécurité de la fonction de hachage construite chapitre 8.

a) Subspace Weight

Ce problème est certainement celui que l'on rencontre le plus souvent : il correspond en fait simplement au cas où on cherche un mot de petit poids (cette fois non nul en plus) ayant un syndrome nul. C'est donc le cas où on ne cherche pas dans n'importe quel coset puisque l'on cherche directement dans le code. Dans leur papier de 1978 Berlekamp, McEliece et van Tilborg montrent aussi que ce problème est NP-complet.

Subspace Weight :
Entrée : une matrice binaire $\mathcal{H}$ et un entier $w > 0$.
Propriété : il existe un mot c de poids w tel que $\mathcal{H}c = 0$.

b) Bounded-Distance Decoding

Ce problème s'énonce ainsi :

Bounded-Distance Decoding :
Entrée : une matrice binaire $\mathcal{H}$ et un mot $\mathcal{S}$.
Promesse : tout ensemble de $d-1$ colonnes de $\mathcal{H}$ est linéairement indépendant.
Propriété : il existe une erreur e de poids $< \frac{d}{2}$ telle que $\mathcal{H}e = \mathcal{S}$.

Ce problème est une variante un peu particulière du problème SD puisqu'il s'agit en fait d'un problème à promesse. Ici, la personne proposant une instance s'engage sur la promesse, mais il est impossible de la vérifier en temps polynomial (la vérification de cette promesse est même un problème NP-complet). De ce fait le problème ne peut pas être dans NP, en revanche il est conjecturé NP-dur.

c) Goppa Parameterized Bounded Decoding

Ici encore il s'agit d'une particularisation du problème SD. On souhaite fixer la valeur de w en fonction des dimensions de $\mathcal{H}$. Ainsi on se restreint à l'ensemble des instances de SD ayant une géométrie donnée.

En particulier il peut être intéressant de montrer que l'ensemble des instances ayant des dimensions propres aux codes de Goppa est lui aussi NP-complet. On énonce donc ce problème ainsi :

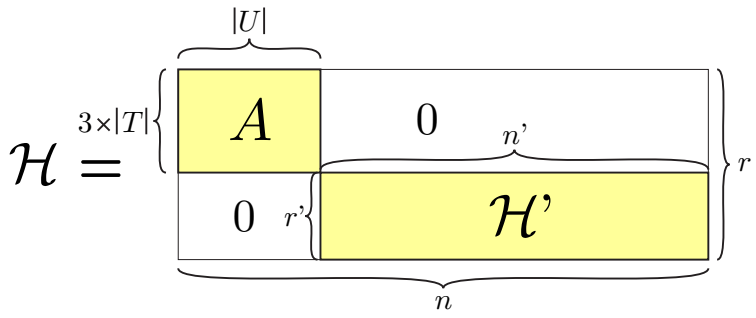


FIG. 6-2 – *Matrice de parité construite pour la réduction de 3DM à GPBD*

Goppa Parameterized Bounded Decoding : (GPBD)
Entrée : une matrice binaire $\mathcal{H}$ de dimension $r \times n$ et un syndrome $\mathcal{S}$.
Propriété : il existe un mot e de poids $\leq \frac{r}{\log_2 n}$ tel que $\mathcal{H}e = \mathcal{S}$.

Pour prouver que ce problème est NP-complet il va s'agir de prouver que savoir résoudre toutes les instances de ce problème permet de résoudre n'importe quelle instance de 3DM et ce avec une réduction polynomiale. Supposons donc que l'on sait résoudre toutes les instances de GPBD. On se donne une instance T, U de 3DM et on va essayer de ramener sa résolution à une instance de GPBD.

Commençons par calculer la matrice d'incidence A de notre instance. Comme nous l'avons vu, si on essaye de résoudre une instance de SD sur cette matrice avec les paramètres $w = |T|$ et $\mathcal{S} = (1, \ldots, 1)$ on peut donner une réponse à notre instance de 3DM. Ici, pour se ramener à une instance de GPBD à la place, nous allons construire une matrice telle que celle de la figure 6-2.

Supposons que la matrice $\mathcal{H}$ ainsi construite ait les bonnes proportions quand on prend $w = |T|$ (c'est-à-dire qu'il faut $|T| = \frac{r}{\log_2 n}$). Dans ce cas, si on essaye de résoudre GPBD sur cette instance avec $\mathcal{S} = (1, \ldots, 1, 0, \ldots, 0)$ ($3 \times |T|$ fois 1 suivis de zéros) et que $\mathcal{H}'$ est la matrice de parité d'un code de distance minimale strictement supérieure à w, alors une solution de cette instance donnera une solution à l'instance de 3DM contenue dans A.

Il ne reste donc qu'à prouver que l'on peut construire une telle matrice. Il nous faut pour cela distinguer trois cas :

1. si A a la bonne forme : $|T| = \frac{3\times|T|}{\log_2 |U|} \quad \Leftrightarrow \quad \log_2 |U| = 3$
2. si A est trop longue : $|T| > \frac{3\times|T|}{\log_2 |U|} \quad \Leftrightarrow \quad \log_2 |U| > 3$
3. si A est trop courte : $|T| < \frac{3\times|T|}{\log_2 |U|} \quad \Leftrightarrow \quad \log_2 |U| < 3$

Dans le premier cas on peut directement appliquer la réduction, comme pour SD, juste en cherchant à décoder le syndrome $(1, \ldots, 1)$.

Dans le deuxième cas, il va falloir choisir une matrice $\mathcal{H}'$ moins allongée que la proportion requise : le plus simple est de choisir $n' = 1$ et de simplement choisir $\mathcal{H}'$ égal à une colonne de 1. Il suffit alors d'avoir la bonne proportion finale soit : $|T| = \frac{r'+3\times|T|}{\log_2(|U|+1)}$. On utilise donc :

$$r' = \log_2(|U| + 1) - 3$$

Enfin, dans le troisième cas, on va utiliser pour $\mathcal{H}'$ une matrice de code de Goppa. Pour cela nous allons choisir $r' = 3 \times |T|$ (la même hauteur que A) et

$n' = n - |U| = 2^{\frac{6\times|T|}{|T|}} - |U| = 64 - |U|$ (on le choisit pour que la matrice $\mathcal{H}$ ait les bonnes proportions). Le code défini par $\mathcal{H}'$ a une distance minimale de $d = 2 \times \frac{r'}{\log_2 n'} + 1 > 2 \times \frac{r'}{\log_2 n} + 1 = |T| + 1$. C'est exactement ce qu'il fallait pour finir la preuve.

On peut donc construire la matrice $\mathcal{H}'$ dans tous les cas de façon à ce qu'elle ait une distance minimale adéquate. La réduction est donc toujours possible et GPBD est donc NP-complet.

d) Le cas général : $w \leq f(n, r)$

On peut reprendre le même raisonnement que pour GPBD pour n'importe quelle forme de matrice, à condition de toujours pouvoir trouver une matrice $\mathcal{H}'$ adéquate. Le problème général peut alors s'énoncer ainsi :

Parameterized Bounded Decoding : (PBD)
Entrée : une matrice binaire $\mathcal{H}$ de dimension $r \times n$ et un syndrome $\mathcal{S}$.
Propriété : il existe un mot e de poids $w \leq f(n, r)$ tel que $\mathcal{H}e = \mathcal{S}$.

Cependant, pour un f donné le problème n'est pas toujours NP-complet (par exemple pour la fonction nulle). Il va donc falloir établir des bornes sur f pour lesquelles le problème est toujours NP-complet. L'idéal serait de pouvoir utiliser la borne de Gilbert-Varshamov (GV). Dans le cas binaire elle s'énonce ainsi :

Théorème :
Si n, r et d vérifient $\sum_{i=0}^{d-1} \binom{n}{i} < 2^{r+1}$ alors un code $[n, n-r, d]$ existe.

Il suffirait donc de chercher n et r tels que n', r' et $|T|+1$ vérifient la borne de Gilbert-Varshamov et donc de vérifier les équations suivantes :

$$\begin{cases} r = r' + 3 \times |T| \\ n = n' + |U| \\ \dfrac{r}{\log_2 n} = |T| \\ \displaystyle\sum_{i=0}^{|T|} \binom{n'}{i} < 2^{r'+1} \end{cases}$$

Malheureusement cette borne n'est pas constructive : on sait qu'il existe des codes ayant ces paramètres mais, comme nous l'avons vu à la section 2-1 page 22, le problème de construire un code de paramètres donnés est difficile et n'a en général pas de solution polynomiale.

Pour une réduction polynomiale il n'est pas possible d'utiliser un code dont la construction n'est elle-même pas polynomiale. Il faudra donc se résoudre à utiliser les meilleures bornes connues sur les paramètres de codes constructibles en temps polynomial [8, p. 672–675]. On pourrait, par exemple, penser utiliser les bornes de Tsfasman et Vlădut pour des codes géométriques constructibles en temps polynomial [56, 93], mais elles ne sont pas meilleures que la borne de GV dans le cas binaire qui nous intéresse ici.

Dans tous les cas, trouver les condition nécessaires les plus faibles sur f n'est certainement pas un problème facile et même si on peut aisément donner des fonctions (comme pour GPBD) ou des familles de fonctions qui conviennent, le cas général reste un problème ouvert.

6-2 Sécurité en moyenne

Nous avons vu que SD était NP-complet et que beaucoup des problèmes voisins l'étaient aussi. En revanche, rien ne prouve que ce problème sera bien adapté à des application cryptographiques. En effet, les cryptosystèmes reposant sur des problèmes de sac à dos [19, 68, ...] (qui est aussi un problème NP-complet) ont, par exemple, tous été cassés au bout du compte [14, 72, 87]. Il se trouve que pour ce problème de sac à dos, toutes les instances dans lesquelles une trappe a été introduite correspondent à des instances relativement faciles à résoudre.

Dans le cas de SD le même problème pourrait se poser. Cependant, même si rien n'a encore pu être prouvé à ce sujet, les spécialistes du sujet [86] tendent à penser que ce problème serait « NP-complet en moyenne ». Même si cette notion est difficile à définir précisément [46, 60], cela pourrait signifier que l'ensemble des instances faciles du problème est de mesure très petite, et que donc, en prenant une instance aléatoire, la probabilité d'obtenir une instance facile est très faible.

D'un point de vue cryptographique la nuance entre les notions de NP-complétude et de complétude en moyenne est assez importante puisque, par exemple, dans le chapitre 8 page 105, la sécurité de la fonction de hachage construite va reposer sur la difficulté de résoudre une instance totalement aléatoire de SD. Avec des propriétés de difficulté en moyenne on peut supposer sans risques que ce système sera solide ; avec une simple propriété de NP-complétude cela serait moins vrai. Pour le cryptosystème de McEliece le problème est encore un peu différent puisqu'il faudrait prouver que pour une instance moyenne du système de McEliece (donc pour une instance de SD utilisant comme matrice une matrice de parité de code de Goppa permutée) la complexité de résolution est élevée. Cela semble cependant encore plus difficile à prouver.

6-3 Meilleures attaques connues sur SD

Il existe une grande variété d'attaques sur SD, ayant toutes leurs particularités, certaines passant beaucoup de temps en précalculs et peu sur l'instance, d'autres utilisant essentiellement de la mémoire... Elles ont cependant toute un point commun : leur coût est exponentiel. Nous ne nous intéresserons donc ici qu'à la variété d'attaque la plus efficace : les attaques utilisant le décodage par ensemble d'information. Les autres types d'attaques (*split syndrome decoding*, *gradient-like decoding*...) sont détaillées dans [8, p. 692–702].

Bien sûr il reste aussi toujours le décodage par recherche exhaustive qui pourra servir de référence pour la complexité. Pour décoder un syndrome donné, cette attaque revient à regarder, dans l'ordre, toute les colonnes de $\mathcal{H}$, puis toutes les combinaisons de deux colonnes, puis de 3 colonnes... et ainsi de suite jusqu'à tomber sur le syndrome recherché. Pour effectuer un décodage borné (problème SD non modifié) il suffit de s'arrêter aux combinaisons de w colonnes et la complexité de l'attaque est donc :

$$\sum_{i=1}^{w} \binom{n}{i} = \mathcal{O}\left(\frac{n^w}{w!}\right)$$

En acceptant d'utiliser beaucoup de mémoire et en utilisant le paradoxe des anniversaires on peut faire baisser cette complexité. Il suffit de construire une grande liste de mots de poids $\frac{w}{2}$ et de chercher des collisions (des mots ayant le même syndrome) dans cette liste. On construit alors des listes de taille :

$$\sum_{i=0}^{\lceil \frac{w}{2} \rceil} \binom{n}{i} = \mathcal{O}\left(\frac{n^{\frac{w}{2}}}{\frac{w}{2}!}\right).$$

L'attaque a alors un coût total en $\mathcal{O}\left(\frac{w}{2}\log_2 n \times (n^{\frac{w}{2}}/\frac{w}{2}!)\right)$.

6-3.1 Décodage par ensemble d'information

Un *ensemble d'information* est un ensemble de k éléments du support du code tel que les k colonnes correspondantes d'une matrice génératrice forment une matrice $k \times k$ inversible.

Le décodage d'un mot $c' = c + e$ par ensemble d'information va consister à choisir aléatoirement un ensemble d'information I, inverser la sous-matrice $\mathcal{G}_I$ de la matrice génératrice et calculer un pseudo-inverse de c' sur ces k positions : $m' = c'_I \times \mathcal{G}_I^{-1}$. Si en recodant le pseudo-inverse m' on trouve un mot de code à distance moins de w de c' alors on a réussi à décoder. Pour que ce soit le cas il suffit que l'ensemble d'information choisi ne contienne aucune des positions non nulles de e. Cela permet donc de tester plusieurs combinaisons d'erreurs en une seule opération : on ajoute une inversion (d'un coût maximum de $\mathcal{O}\left(k^3\right)$) pour regarder d'un seul coup $\binom{n-k}{w}$ motifs d'erreur. En supposant que l'on a une seule solution (ce qui est le cas quand on s'attaque au cryptosystème de McEliece), la complexité finale est de l'ordre de :

$$\text{Complexité} = \mathcal{O}\left(k^3 \frac{\binom{n}{w}}{\binom{n-k}{w}}\right) \simeq \mathcal{O}\left(k^3 \left(\frac{n}{n-k}\right)^w\right)$$

Si on cherche maintenant à résoudre une instance du problème SD, c'est-à-dire trouver un mot de petit poids ayant un syndrome donné cela se passe exactement de la même façon, mais pour la matrice de parité : on choisit un ensemble d'information, on prend la sous-matrice carrée correspondant au $n-k$ colonnes complémentaires de la matrice de parité et on calcule un pseudo inverse du syndrome avec cette matrice. S'il a un poids plus petit que w c'est gagné, autrement on recommence avec un autre ensemble d'information. On va tester exactement le même nombre de mots de petits poids que le nombre de motifs d'erreur dans le cas du décodage, et le coût supplémentaire est $\mathcal{O}\left((n-k)^3\right)$. On a donc une complexité similaire (en supposant qu'il n'y a qu'une solution) :

$$\text{Complexité} = \mathcal{O}\left((n-k)^3 \left(\frac{n}{n-k}\right)^w\right)$$

Les deux problèmes sont donc quasiment identiques, mais de façon générale, la deuxième façon de voir les choses est mieux adaptée pour affiner les attaques. Notons aussi que cette méthode va apporter un gain considérable par rapport à la recherche exhaustive quand il y a une solution au problème. Dans le cas où il n'y a pas de solution elle risque en revanche de devenir plus coûteuse si on ne choisit pas bien les ensembles d'information que l'on teste : pour bien faire il

faudrait que leurs complémentaires forment un recouvrement minimal de tous les ensembles de w positions du support.

6-3.2 Attaques raffinées

Les attaques que je présente ici sont des évolutions du décodage par ensemble d'information utilisant des calculs supplémentaires pour chaque ensemble choisi (dans le but de tester encore plus de motifs d'erreur à la fois) et reposant sur un choix des ensembles d'information permettant de ne pas recalculer tout le pivot à chaque fois. Certains de ces algorithmes ont été inventés directement pour servir d'attaque contre le système de McEliece et permettre d'établir plus précisément des paramètres offrant une bonne sécurité, d'autres ont été mis au point pour chercher des mots de petit poids dans un code ou déterminer la distance minimale d'un code. Dans les deux cas les applications sont les mêmes et les algorithmes permettent tous d'attaquer un peu mieux le problème SD général.

Les algorithmes que je détaille ici ne sont que les principales étapes dans l'amélioration des attaques contre le problème SD. D'autres papiers traitant de la sécurité de ce problème et du cryptosystème de McEliece ont bien sûr été publiés pendant ce temps-là [1, 2, 59].

a) Lee & Brickell

Cette attaque a été décrite en 1988 par Lee et Brickell [58]. C'est le premier papier à avoir été publié qui améliore nettement la complexité du décodage par ensemble d'information tel que décrite précédemment. L'idée est de faire quelques calculs supplémentaires pour chaque ensemble d'information choisi. Les auteurs partent du principe qu'il est beaucoup plus facile de trouver un ensemble d'information qui contient quelques erreurs, qu'un ensemble n'en contenant aucune. Ils procèdent donc de la façon suivante :

1. tirer un ensemble d'information I et calculer $c' + c'_I \times \mathcal{G}_I^{-1}\mathcal{G}$
2. énumérer toutes les erreurs e_I de poids $\leq j$ dans l'ensemble I
3. pour chaque e_I, regarder si le poids de $c' + c'_I\mathcal{G}_I^{-1}\mathcal{G} + e_I\mathcal{G}_I^{-1}\mathcal{G}$ est plus petit que w
4. retourner à l'étape 1 pour un autre I

Libre à l'attaquant de choisir la valeur de j la mieux adaptée. Le coût de l'attaque devient alors :

$$\text{Complexité} = \mathcal{O}\left(\frac{1}{\mathcal{P}_j}(k^3 + k\sum_{i=0}^{j}\binom{k}{i}\right),$$

où $\mathcal{P}_j$ désigne la probabilité que I convienne et contienne donc j erreurs ou moins. On a donc :

$$\mathcal{P}_j = \sum_{i=0}^{j}\frac{\binom{w}{i}\binom{n-w}{k-i}}{\binom{n}{k}}$$

L'optimal sera à peu près toujours pour $j = 2$, quand le coût du pivot est voisin de celui de l'énumération des erreurs. Dans le cas des paramètres initiaux

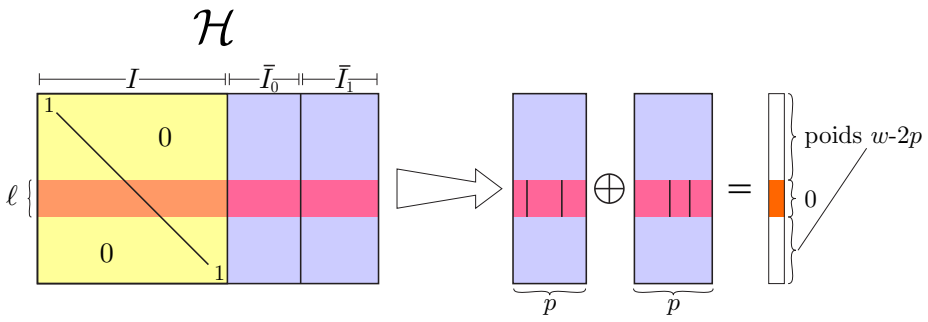


FIG. 6-3 – *L'algorithme de Stern pour résoudre SD.*

de McEliece ($n = 1024$, $k = 524$ et $w = 50$) le facteur de travail d'une attaque tombe de 2^{84} à 2^{73}.

Lee et Brickell proposent ensuite de ne pas recalculer entièrement l'étape 2 et de regarder le poids obtenu au fur et à mesure. Il suffit donc en moyenne de calculer $2w$ lignes pour dépasser le poids de w. Ils gagnent ainsi un facteur 10 en moyenne.

De même, ils évoquent la possibilité de ne pas refaire entièrement le pivot entre les différents choix d'ensembles d'information en choisissant des ensembles qui ne diffèrent qu'en un point et espèrent ainsi gagner encore un facteur 10.

b) Stern

Cette deuxième méthode, présentée par Stern [88], a d'abord été conçue pour chercher des mots de petits poids dans un code. Comme je l'ai dit, ce problème est quasiment équivalent au problème de décodage et l'algorithme peut s'adapter très facilement.

La technique employée par cet algorithme repose sur les mêmes idées que celui de Lee et Brickell : encore une fois on commence par choisir un ensemble d'information I, puis on effectue un pivot (sur la matrice de parité $\mathcal{H}$ ici) et on va chercher des solutions ayant un poids $2p$ hors de l'ensemble d'information (dans son complémentaire $\bar{I}$) et $\leq w - 2p$ dans l'ensemble.

C'est à ce moment-là que vont intervenir les modifications. L'algorithme de Lee et Brickell consisterait en gros à essayer toutes les erreurs de poids $2p$ dans $\bar{I}$, calculer le syndrome correspondant et regarder si son poids est $\leq w - 2p$ (ou, dans le cas d'un décodage, la distance entre ce syndrome et celui que l'on cherche). Ici à la place on va économiser un peu en utilisant le paradoxe des anniversaires : on sépare $\bar{I}$ en deux ensembles $\bar{I}_0$ et $\bar{I}_1$ (pas nécessairement de tailles égales), on calcule toutes les combinaisons de p colonnes de $\bar{I}_0$ et toutes celles de $\bar{I}_1$ et il suffit ensuite de trouver deux colonnes (une de chaque ensemble) dont la somme est de poids $\leq w - 2p$. C'est là qu'intervient le deuxième raffinement de l'algorithme, en utilisant une idée de Leon [59] : au lieu de regarder les colonnes en entier on va s'intéresser uniquement à une « fenêtre » de ℓ lignes de la matrice $\mathcal{H}$, et au lieu de chercher toutes les combinaisons de poids faible on va uniquement regarder les combinaisons qui s'annulent sur cette fenêtre (voir figure 6-3). On réduit ainsi la probabilité de succès, mais on diminue considérablement la quantité de travail à fournir.

Le probabilité de succès de cet algorithme va donc dépendre de p et ℓ que

l'on pourra ajuster pour trouver le meilleur compromis. La probabilité d'avoir un choix de I, $\bar{I}_0$, $\bar{I}_1$ et de la fenêtre de taille ℓ qui convienne quand on a une seule solution de poids w au problème est :

$$\mathcal{P} = \frac{\binom{w}{2p}\binom{n-w}{k-2p}}{\binom{n}{k}} \times \frac{\binom{2p}{p}}{4^p} \times \frac{\binom{n-k-w+2p}{\ell}}{\binom{n-k}{\ell}}$$

Le premier facteur est la probabilité de bien choisir l'ensemble d'information, le deuxième celui de bien choisir $\bar{I}_0$ et $\bar{I}_1$ et le troisième la fenêtre de taille ℓ.

Le pivot de Gauss a un coût de $\mathcal{O}\left(n(n-k)^2\right)$ et le calcul des combinaisons de colonnes de $\bar{I}_0$ et $\bar{I}_1$ un coût de $\mathcal{O}\left(\ell p\binom{k/2}{p}\right)$. Enfin, le coût de la comparaison des éléments des deux ensembles et du calcul du poids total de chaque solution est $\mathcal{O}\left(2p(n-k) \times \binom{k/2}{p}^2 / 2^\ell\right)$.

Idéalement il faudra ajuster p et ℓ pour que les trois étapes aient des coûts similaires.

c) *Canteaut-Chabaud*

Cette dernière version [17] est l'algorithme le plus performant pour trouver une solution au problème SD. Elle est construite exactement comme la méthode de Stern vue précédemment, mais tient compte de la remarque suivante : l'étape la plus coûteuse dans l'algorithme est le calcul du pivot, si on peut réduire son coût on pourra alors choisir des p et ℓ encore mieux adaptés et diminuer le coût total de l'algorithme.

Pour diminuer le coût du pivot, Canteaut et Chabaud vont reprendre l'idée évoquée dans l'article de Lee et Brickell : au lieu de choisir des ensembles d'information complètement indépendants les uns des autres on va choisir des ensembles qui ne diffèrent que sur une seule position. La mise à jour du pivot est alors beaucoup plus rapide, mais en revanche parmi les solutions que l'on va examiner il y en aura nécessairement certaines de communes avec l'étape précédente. C'est donc un compromis qui va diminuer le coût de chaque étape en diminuant un peu la probabilité de succès. De plus, le fait que les ensembles ne soient pas indépendants oblige à utiliser des chaînes de Markov dans le calcul de la probabilité de succès, et l'évaluation de la complexité totale est de ce fait beaucoup plus compliquée. Avec cet algorithme, le coût d'une attaque sur le cryptosystème de McEliece utilisant les paramètres d'origine $[1024, 524, 101]$ est alors de $2^{64.2}$ opérations binaires.

Chapitre 7

Le système de chiffrement à clef publique Augot-Finiasz

YNDROME decoding est le problème NP-complet qui correspond au problème de décodage d'un code aléatoire. Cependant, n'importe quel code donné est, en général, difficile à décoder. De ce fait, même des codes construits pour être faciles à décoder jusqu'à leur distance construite, se trouvent souvent difficiles à décoder au-delà. Comme nous l'avons vu en introduction (voir section 1-2.3.b page 18), c'est le cas par exemple pour les codes de Reed-Solomon : on les construit pour pouvoir décoder de manière unique jusqu'à $\frac{n-k}{2}$ erreurs. Au-delà, l'algorithme de Guruswami-Sudan [47, 92] permet encore de décoder en temps polynomial, mais plus de façon unique (il renvoie la liste des solutions possibles) jusqu'à $n - \sqrt{kn}$ erreurs. Encore au-delà, leur décodage devient un problème difficile.

Ce problème est aussi connu sous le nom de Polynomial Reconstruction, un problème très proche du problème PolyAgree qui est NP-dur. De plus, récemment, Guruswami et Vardy [48] ont prouvé que le problème général de décodage des codes de Reed-Solomon était lui aussi NP-dur. Faire reposer un cryptosystème sur la difficulté de décoder un code de Reed-Solomon semble donc une bonne idée.

Dans ce chapitre, j'expose une construction permettant de cacher une instance facile de Polynomial Reconstruction à l'intérieur d'une instance difficile. La connaissance d'une solution de cette instance difficile étant la trappe du cryptosystème. Malheureusement, la sécurité de cette construction repose sur le problème de décoder dans un code engendré par un Reed-Solomon et un mot aléatoire qui, contrairement à ce qui était cru initialement, n'est pas difficile. Comme nous le verrons section 7-6 page 99, ce cryptosystème est cassé. L'idée de base consistant à cacher une instance facile d'un problème difficile dans une autre instance pourrait cependant être réutilisée avec d'autres problèmes.

Les résultats présentés dans ce chapitre ont fait l'objet d'une publication à EUROCRYPT'03 [6].

7-1 Polynomial Reconstruction – décodage des Reed-Solomon

Le problème de Polynomial Reconstruction s'énonce de la façon suivante :

Polynomial Reconstruction : (PR)
Entrée : n, k et t des entiers et n couples $(x_i, y_i)_{i=1\dots n}$ d'éléments de $\mathbb{F}_q$ (les x_i tous distincts).
Propriété : il existe un polynôme $p(X)$ tel que $\deg p(X) < k$ et $p(x_i) \neq y_i$ pour au plus t valeurs de i.

Le problème de décodage des Reed-Solomon, lui, s'énonce ainsi :

Décodage des Reed-Solomon : (DRS)
Entrée : un code de Reed-Solomon RS_k de paramètres $[n, k]$ sur $\mathbb{F}_q$, un entier t et un mot $y \in \mathbb{F}_q^n$.
Propriété : il existe un mot du code RS_k à distance t ou moins de y.

Dans le problème DRS, si on choisit les x_i comme étant les éléments du support du code on se retrouve exactement avec une instance de PR. De même, une instance de PR peut se ramener à une instance de DRS pour un code ayant comme support l'ensemble des x_i. Les deux problèmes sont donc équivalents et sont simplement deux formulations différentes d'un même problème.

Dans [40, 41], les auteurs prouvent que le problème PolyAgree ci après est NP-dur :

PolyAgree :
Entrée : n, k et t des entiers et n couples $(x_i, y_i)_{i=1\dots n}$ d'éléments de $\mathbb{F}_q$.
Propriété : il existe un polynôme $p(X)$ tel que $\deg p(X) < k$ et $p(x_i) \neq y_i$ pour au plus t valeurs de i.

Cependant, leur preuve repose en partie sur la seule différence entre ce problème et PR : le fait qu'ici les x_i peuvent être égaux. Leur preuve ne semble donc pas pouvoir s'adapter directement au problème de PR. Heureusement, très récemment, Guruswami et Vardy [48] ont pu prouver le même résultat pour le problème général de décodage des codes de Reed-Solomon.

Il est cependant à noter que les valeurs des paramètres pour lesquels les instances de PR ou DRS sont difficiles sont assez restreintes. Pour chacun, si $t \geq n-k$ on peut résoudre facilement l'instance par interpolation, si $t \leq \frac{n-k}{2}$ on peut trouver la réponse unique à l'instance en temps polynomial avec l'algorithme de décodage des codes de Reed-Solomon (algorithme d'Euclide, Berlekamp-Welch ou Berlekamp-Massey), et enfin si $t \leq n - \sqrt{nk}$ on peut trouver une liste de solutions en utilisant l'algorithme de Guruswami-Sudan. En conclusion, les seuls cas où ces problèmes sont effectivement difficiles sont les cas où $n - \sqrt{kn} < t < n - k$. Ce sont ces instances qui font de ces problèmes des problèmes difficiles et il sera donc nécessaire de toujours rester dans cet intervalle pour notre cryptosystème.

7-2 Construction du cryptosystème

Comme nous l'avons vu, le problème PR n'est difficile à résoudre que pour des valeurs des paramètres assez précises. Il est donc très facile de basculer d'une

instance facile à résoudre vers une instance difficile, et réciproquement. L'idée de ce cryptosystème est donc de mélanger les deux types d'instances : cacher une instance facile dans une instance difficile, et cela de façon à ce que la résolution de l'instance facile nécessite la connaissance d'une solution de l'instance difficile.

Dans notre système, nous allons donc mélanger deux instances de DRS, toutes deux issues du même code de dimension k et de support de longueur n sur le corps $\mathbb{F}_q$. L'une de ces instances consiste à décoder un mot de code auquel on a ajouté w erreurs pour un w plus petit que $\frac{n-k}{2}$, l'autre à décoder un mot auquel on a ajouté W erreurs de façon à rendre cette instance difficile.

Le principal problème qui se pose est de déterminer comment mélanger ces instances de façon à les lier du mieux possible.

7-2.1 Description du système

Nous nous plaçons ici dans le corps $\mathbb{F}_q$ qui est une extension de $\mathbb{F}_2$ de façon à ce que $\log_2 q$ soit un entier. La clef publique sera l'instance difficile de DRS, soit un mot $K = c + E$ où c est un mot du code RS_k (évaluation d'un polynôme $\mathcal{P}_c$ de degré $k-1$) et E une erreur de poids W. La clef secrète quand à elle sera la solution à cette instance : le couple (c, E). Pour chiffrer un message m de $(k-1)\log_2 q$ bits nous procédons de la façon suivante :

1. convertir les $(k-1)\log_2 q$ bits de m en un polynôme $\mathcal{P}_m$ de degré $k-2$ sur $\mathbb{F}_q$,
2. calculer c_m le mot de code associé à m : l'évaluation de $\mathcal{P}_m$,
3. choisir α aléatoirement dans $\mathbb{F}_q$,
4. choisir une erreur e de poids w et de longueur n,
5. calculer le chiffré $y = c_m + \alpha K + e$.

Le processus de chiffrement est donc relativement simple et peu coûteux : le plus cher est le calcul de c_m : l'évaluation d'un polynôme de degré $k-2$ sur $\mathbb{F}_q$ en n points. Notons que quelqu'un qui souhaiterait directement décoder y dans le code de Reed-Solomon sera obligé de décoder à la fois E et e et donc une erreur de poids trop grand.

Pour déchiffrer, il suffit d'utiliser l'information contenue dans la clef secrète : on connaît c et E. Le chiffré y peut donc s'écrire : $y = c_m + \alpha c + e + \alpha E$. Introduisons le code $\overline{\mathrm{RS}}_k$ correspondant au code de Reed-Solomon raccourci sur les positions non nulles de E. Ce code est de longueur $n - W$ et toujours de dimension k. Pour déchiffrer on procède donc comme suit :

1. écrire $\bar{y} = \bar{c}_m + \alpha\bar{c} + \bar{e} + \alpha\bar{E} = \bar{c}_m + \alpha\bar{c} + \bar{e}$,
2. $\bar{c}_m$ et $\bar{c}$ sont dans $\overline{\mathrm{RS}}_k$ donc si $w \leq \frac{n-W-k}{2}$ on peut décoder $\bar{e}$.
3. on retrouve un polynôme $\mathcal{P}$ de degré k s'évaluant en $\bar{c}_m + \alpha\bar{c}$.
   $\bar{c}$ est l'évaluation de $\mathcal{P}_c$ et $\bar{c}_m$ celle de $\mathcal{P}_m$ donc $\mathcal{P} = \alpha\mathcal{P}_c + \mathcal{P}_m$
4. $\mathcal{P}_m$ de degré $k-2$ et on connaît $\mathcal{P}_c$: on peut retrouver α à partir du coefficient dominant de $\mathcal{P}$,
5. on en déduit $\mathcal{P}_m = \mathcal{P} - \alpha\mathcal{P}_c$ et donc m.

Comme on peut le constater, le déchiffrement n'est possible qu'à deux conditions : avoir pris $\mathcal{P}_m$ de degré strictement inférieur à $\mathcal{P}_c$ afin de pouvoir retrouver α, et avoir choisi w suffisamment petit pour permettre un décodage en temps polynomial (il faut que l'instance sur le code raccourci soit une instance facile). Une fois ces deux conditions rassemblées les deux algorithmes de chiffrement et déchiffrement sont relativement simples et peu coûteux en temps de calcul comme en mémoire.

7-2.2 Quelques remarques

Avec cette construction le choix de w n'est pas optimal. En effet, si on prend $w \leq \frac{n-W-k}{2}$ on peut, comme prévu dans l'algorithme de déchiffrement, décoder dans le code raccourci avec les techniques de décodage classique. On pourrait cependant utiliser l'algorithme de Sudan pour effectuer ce décodage : il ne garantit pas d'avoir une réponse unique, mais selon le code de Reed-Solomon utilisé, la probabilité d'avoir plusieurs réponses peut être négligeable. Ainsi, cela permettrait de choisir n'importe quel $w \leq n - W - \sqrt{k(n-W)}$, ou même un peu plus, si on considère que parmi les W positions sur lesquelles on raccourcit le code avant de faire le décodage il y a de fortes chances d'avoir quelques-unes des w erreurs ajoutées pendant le chiffrement. Dans tous les cas, cela permet d'utiliser un w un peu plus grand qui donnera une meilleure sécurité pour le système. Cependant, cela nécessite d'effectuer un décodage avec l'algorithme de Sudan qui est quand même plus coûteux que l'algorithme classique.

Remarquons aussi que lorsque l'on raccourcit le code pour passer de RS_k à $\overline{\mathrm{RS}}_k$ on a de bonnes chances de supprimer quelques erreurs de e. L'erreur raccourcie $\bar{e}$ est donc en général de poids inférieur à w. Selon la valeur de W il est peut-être possible d'augmenter un peu w en ayant une probabilité négligeable que le poids de $\bar{e}$ soit supérieur à la capacité de correction de $\overline{\mathrm{RS}}_k$. Une étude précise de la valeur optimale de w tenant compte de ce fait est disponible dans [57].

7-3 Sécurité théorique

7-3.1 Liens entre m, α et e

Avant d'étudier les différentes sortes d'attaques applicables au système il est intéressant de remarquer comment m, α et e sont liés. Effectivement, la connaissance de n'importe lequel de ces trois éléments permet de facilement retrouver les deux autres.

a) Si on connaît m

Quand on connaît m (et donc $\mathcal{P}_m$) on peut calculer $y - c_m = \alpha(c + E) + e$. Il suffit alors de regarder $2w + 1$ positions de ce mot et parmi ces positions une majorité sont proportionnelles à $c + E$. On peut donc retrouver α, le coefficient de proportionnalité majoritaire. Une fois m et α connus on retrouve e.

b) Si on connaît α

On calcule $y - \alpha(c + E) = c_m + e$, et on retrouve e et c_m simplement en appliquant l'algorithme de décodage des Reed-Solomon. En effet, on est

nécessairement dans une instance facile puisque w est choisi pour donner une instance facile une fois dans le code raccourci.

c) Si on connaît e

On calcule ici aussi $y-e = c_m+\alpha(c+E)$. Ce mot est un mot du code RS+1, le code engendré par le Reed-Solomon et la clef publique $c + E$. On connaît la matrice génératrice de ce code et on a un mot non bruité, on peut donc facilement retrouver le message correspondant à ce mot de code en calculant un pseudo inverse de la matrice génératrice. Ce message sera de longueur $k + 1$, avec les k premiers coefficients correspondants exactement à m et le $(k + 1)^{\text{ème}}$ étant α.

Les éléments m, α et e étant liés, toutes les attaques visant à retrouver l'un de ces paramètres pourront être rangées dans la même catégorie d'attaques. Cela va donc beaucoup simplifier l'inventaire des attaques possibles.

7-3.2 Les problèmes sous-jacents

Avant de regarder les meilleures attaques sur le système il est important de bien identifier les différents problèmes que l'on doit savoir résoudre.

Le première remarque à faire est que la connaissance de couples clair/chiffré n'apporte aucune information sur la clef secrète. Cependant, dans ce système, un mot y aléatoire n'est pas nécessairement déchiffrable. Il est donc important de savoir si le fait de savoir si un mot donné est déchiffrable ou non apporte des informations quand à la clef secrète. C'est malheureusement le cas pour ce système : si on essaye de déchiffrer un message construit avec une erreur e de poids un peu supérieur à w, il ne pourra être déchiffré que si une partie de l'erreur e est contenue dans l'erreur E et a été supprimée en raccourcissant le code. En faisant déchiffrer successivement plusieurs messages avec des e bien choisis, un attaquant peut ainsi retrouver E. Il faut donc noter que ce système ne résiste, a priori, à aucune forme d'attaque à chiffré choisi, adaptative ou non (une attaque adaptative permettra juste d'utiliser moins de chiffrés différents).

Maintenant, si on considère des attaques ne faisant pas appel à un oracle de déchiffrement, on constate qu'il n'y a que deux façons de s'attaquer au système pour déchiffrer un message : soit on essaye de retrouver la clef secrète, soit on essaye de déchiffrer le message sans connaître cette clef secrète.

Nous appellerons RS+1, Reed-Solomon augmenté d'un mot, un code engendré par RS_k et un mot quelconque hors de RS_k. Ainsi, les chiffrés y sont des mots d'un code RS+1 auquel on a ajouté une erreur de poids w.

Retrouver la clef secrète consiste exactement à résoudre une instance difficile de DRS. En revanche, essayer de déchiffrer sans connaître la clef secrète correspond à un problème beaucoup moins clair. En effet plusieurs problèmes se posent : le problème de décoder avec une clef publique telle qu'elle est construite est-il différent du problème où on remplacerait cette clef par un mot aléatoire ? Décoder w erreurs dans le code RS+1 est-il équivalent à décoder dans un code aléatoire ?

Il est clair que par rapport à un code complètement aléatoire, dans notre cas on peut toujours faire une recherche exhaustive sur α pour décoder. Mais peut-on faire mieux qu'une recherche exhaustive ? Il faudra donc distinguer deux sortes d'attaques : les attaques visant à retrouver α qui prennent en compte le

fait que RS+1 n'est pas un code aléatoire, et les autres attaques qui traiteront RS+1 comme un code aléatoire. Le deuxième type d'attaque correspond exactement à des attaques sur le problème SD : décoder dans un code aléatoire. En revanche, le premier type ne correspond à aucun problème difficile connu. Dans la section 7-3.3 nous considérons que la meilleure attaque est la recherche exhaustive. Cependant, comme nous le verrons dans la section 7-6 page 99, cette considération n'est pas raisonnable et retrouver α peut se faire, en général, en temps polynomial.

7-3.3 Les attaques possibles

Comme nous l'avons vu, si on laisse de côté les attaques visant à retrouver directement α, il y a deux types d'attaques possibles : retrouver la clef secrète en s'attaquant à une instance de DRS, ou retrouver le clair correspondant à un chiffré en s'attaquant à une instance de SD.

a) Attaques sur la clef

Pour s'attaquer à une instance difficile de DRS on peut encore une fois procéder de deux façons. Soit on considère que n'ayant pas d'algorithme corrigeant suffisamment d'erreurs, la structure de Reed-Solomon du code ne nous apporte rien et dans ce cas on essaye de résoudre cette instance comme une instance de SD. Dans ce cas les meilleures attaques seront les attaques basées sur la recherche d'ensembles d'information comme l'algorithme de Canteaut-Chabaud [17]. Autrement, on peut essayer d'utiliser la structure du code de Reed-Solomon pour s'aider. Dans ce cas il faudra essayer de modifier l'instance difficile afin de pouvoir appliquer un algorithme de décodage connu. La meilleure méthode sera d'essayer de raccourcir le code en des positions aléatoires en espérant supprimer des valeurs non nulles de E. On peut ainsi faire diminuer W de façon à se ramener à une instance facile.

b) Attaques sur une instance

Dans RS+1, α peut être vu comme une partie du message : soit on décide de le traiter à part en utilisant le fait que l'on a une structure de Reed-Solomon sur le reste, soit on le traite comme les autres coefficients. Dans le premier cas on aura une attaque exhaustive qui va consister à énumérer tous les éléments du corps sur lequel est construit le Reed-Solomon et pour chaque valeur, essayer de décoder le complément. Dans le deuxième cas on va devoir décoder w erreurs dans un code "aléatoire". On se retrouve donc encore une fois dans un cas où le meilleur algorithme sera celui de Canteaut et Chabaud, mais avec un code de dimension 1 de plus et une erreur de poids w au lieu de W.

7-4 Sécurité pratique

Pour déterminer des paramètres adéquats pour le système il est nécessaire de bien évaluer le coût de chacune des attaques énumérées précédemment. Comme toujours, nous essayerons de viser une sécurité de 2^{80} opérations binaires.

7-4.1 Coût des attaques

a) Retrouver la clef secrète en utilisant la structure de RS ($\mathcal{ESD}_W$)

On essaye ici de résoudre une instance difficile de DRS en se ramenant à une instance plus simple, que l'on sait résoudre. Pour cela on peut raccourcir le code en β positions, et si ces β positions sont incluses dans les W positions non nulles de E on n'aura plus qu'à décoder une erreur de poids $W-\beta$ dans un code de longueur $n-\beta$ et de dimension k. Si on a réussi à se ramener à une instance facile la résolution peut se faire en temps polynomial.

Le coût de cette attaque sera donc le produit du coût (polynomial) de l'algorithme de décodage de Sudan et de l'inverse de la probabilité de choisir les β positions correctement. Pour plus de simplicité nous négligerons la partie polynomiale pour ne regarder que la probabilité qui est, elle, exponentielle.

Il est à noter que la valeur optimale de β n'est pas forcément facile à déterminer : il se peut qu'en prenant un β plus grand on augmente suffisamment la probabilité de tomber parmi les W erreurs pour compenser la perte de longueur (et donc de capacité de correction). Nous allons donc évaluer la probabilité de trouver β positions qui conviennent en fonction de la valeur choisie pour β et ainsi déterminer l'optimal.

Supposons que parmi les β positions choisies, β_0 soient incluses dans les W positions non nulles de l'erreur. On se retrouve donc à résoudre le problème DRS avec les paramètres $W' = W - \beta_0$, $n' = n - \beta$ et $k' = k$. Pour pouvoir décoder il faut donc $W' < n' - \sqrt{n'k'}$ et donc aussi $\beta_0 > W - n' + \sqrt{n'k'}$. La probabilité de choisir correctement les β positions est donc égale à la probabilité d'avoir un β_0 dans l'intervalle $\left[W - n' + \sqrt{n'k'}; \beta\right]$ (si cet intervalle est non vide).

On peut calculer la probabilité que β_0 vaille i :

$$\mathcal{P}_{\beta_0=i} = \frac{\binom{W}{i}\binom{n-W}{\beta-i}}{\binom{n}{\beta}},$$

et donc la probabilité de choisir β positions permettant de décoder :

$$\mathcal{P}_\beta = \sum_{i=W+\beta-n+\sqrt{(n-\beta)k}}^{\beta} \frac{\binom{W}{i}\binom{n-W}{\beta-i}}{\binom{n}{\beta}} = \sum_{i=0}^{n-W-\sqrt{(n-\beta)k}} \frac{\binom{W}{\beta-i}\binom{n-W}{i}}{\binom{n}{\beta}}.$$

Le coût de l'attaque sera l'inverse de cette somme et un attaquant choisira donc β afin de maximiser cette probabilité.

Sur la figure 7-1 page suivante on remarque que le coût de l'attaque est croissant (en dents de scie) avec β. L'optimal sera donc toujours soit la valeur minimale (la plus petite valeur pour laquelle $n - W - \sqrt{(n-\beta)k} \geq 0$), soit un de plus (selon que cela ajoute un terme à la somme ou pas). Dans tous les cas la probabilité sera presque la même et toujours choisir le plus petit β ne coûtera rien à l'attaquant. On pourra donc utiliser $\beta = n - \frac{(n-W)^2}{k}$.

b) Retrouver la clef secrète en résolvant une instance de SD (ISD_W ou CC_W)

Cette attaque va utiliser la recherche d'ensembles d'information (ISD_W). Sa complexité sera donc de l'ordre de l'inverse de la probabilité de trouver un ensemble d'information valide pour notre instance. Ici, il faut trouver un ensemble

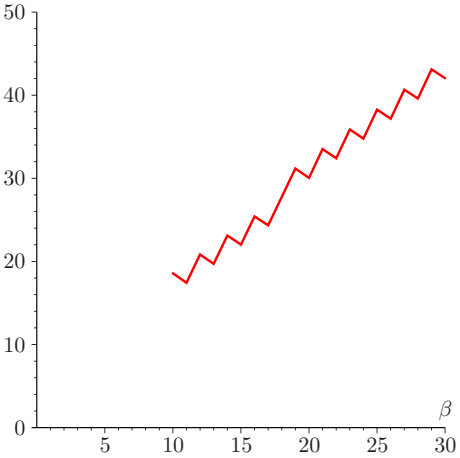


FIG. 7-1 – *Complexité (logarithme de l'inverse de la probabilité de succès) de l'attaque $\mathcal{ESD}_W$ en fonction de la valeur de β, ici pour les paramètres $n = 1024$, $k = 900$ et $W = 65$.*

de k positions parmi n ne contenant aucune des W erreurs. La probabilité de bien tomber est donc :

$$\mathcal{P}_{\text{valide}} = \frac{\binom{n-W}{k}}{\binom{n}{k}} = \frac{\binom{n-k}{W}}{\binom{n}{W}}$$

Avec un algorithme de base le coût de l'attaque serait l'inverse de cette probabilité multiplié par un coût polynomial (le coût d'un pivot de Gauss sur la matrice de parité du code). En utilisant un algorithme plus évolué comme celui de Canteaut-Chabaud [17] ($\mathcal{CC}_W$) on peut éliminer ce facteur polynomial en ne recalculant pas entièrement le pivot à chaque fois, de plus il est aussi possible d'améliorer la probabilité de succès en faisant un peu de calcul supplémentaire à chaque instance. Cependant, comme nous le verrons sur la figure 7-2 page 96 cette attaque a asymptotiquement un coût très proche de celui de l'attaque élémentaire.

c) Retrouver α pour une instance

Comme nous l'avons remarqué à la section précédente, il ne semble a priori pas évident de pouvoir faire mieux pour retrouver α qu'une recherche exhaustive parmi les éléments du corps utilisé. La complexité de cette attaque serait donc la taille du corps multipliée par un facteur polynomial correspondant au coût d'un décodage classique.

d) Décoder e dans $\mathcal{RS}$+1 comme dans un code aléatoire ($\mathcal{ISD}_w$ ou $\mathcal{CC}_w$)

Ici, comme dans l'attaque de b) page précédente, il s'agit de décoder dans un code aléatoire. On utilise donc l'algorithme de Canteaut-Chabaud, mais maintenant avec les paramètres n, $k+1$ et w. La dimension est donc un de plus et le poids différent. On trouve une probabilité de choisir un ensemble d'information valide ($\mathcal{ISD}_w$) égale à :

$$\mathcal{P}_{\text{valide}} = \frac{\binom{n-k-1}{w}}{\binom{n}{w}} \quad \text{avec} \quad w = n - W - \sqrt{(n-W)k}$$

L'attaque de Canteaut-Chabaud (CC_w) aura donc un coût un peu inférieur à l'inverse de cette probabilité.

7-4.2 Courbes des coûts

Les attaques possibles étant bien identifiées il faut maintenant trouver des paramètres tels que la sécurité de la meilleure attaque soit en 2^{80} opérations binaires. Les paramètres que nous pouvons choisir sont :

- q la taille du corps $\mathbb{F}_q$ dans lequel on travaille,
- n la longueur du code,
- k la dimension du code,
- W le poids de l'erreur E,
- w le poids de la petite erreur e.

L'attaque par recherche exhaustive sur α oblige à avoir au moins 2^{80} éléments dans le corps. Il faut donc $\log(q) \geq 80$. De plus, plus le poids de e est élevé, plus cette erreur sera difficile à décoder pour un attaquant (tant qu'elle ne dépasse pas $n-k$). Comme $w \leq n - W - \sqrt{(n-W)k} < n-k$, le mieux est de choisir $w = n - W - \sqrt{(n-W)k}$. Rappelons que l'on peut aussi choisir $w = \frac{n-W-k}{2}$ si on ne veut pas utiliser l'algorithme de Sudan pour décoder : cela donnera une moins bonne sécurité mais un algorithme un peu plus rapide.

Il ne reste donc qu'à choisir les valeurs de n, k et W. La longueur n va en même temps donner la taille de la clef publique (c'est un mot de longueur n sur $\mathbb{F}_q$) et idéalement des petites valeurs seront plus commodes à utiliser. De même, plus k est grand, plus le taux de transmission du chiffrement (égal à $\frac{k-1}{n}$) est élevé. Cependant plus k est grand, plus la valeur du w sera petite et cela rend plus facile les attaques sur une instance. Pour bien voir comment se comportent les complexités des différentes attaques on peut tracer quelques courbes. La figure 7-2 page suivante montre ce que cela donne pour différentes valeurs de n et k. On peut constater en particulier que pour $n = 1024$ la valeur $k = 900$ semble donner une sécurité suffisante pour $W = 74$.

7-4.3 Comportement asymptotique

En prenant en compte les 3 types d'attaques exposés précédemment on constate qu'il n'est pas dur de trouver un jeu de paramètres correspondant au niveau de sécurité que l'on désire. Asymptotiquement il se trouve que trouver de tels paramètres reste toujours aussi simple.

a) Complexités

Afin de choisir des paramètres asymptotiques corrects il est nécessaire de regarder la complexité asymptotique des différentes grandeurs entrant en jeu dans le système. On a :

- taille de clef : $n \log q$
- taux de transmission : $\frac{k-1}{n}$
- taille de bloc : $(k-1)\log q$ (en entrée).
- coût du chiffrement : $\mathcal{O}\left(nk \log^2 q\right)$ opérations binaires

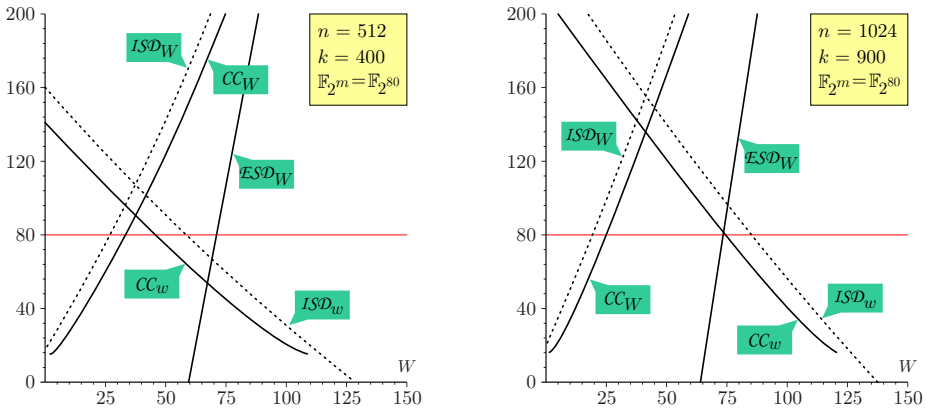


FIG. 7-2 – *Logarithme du facteur de travail binaire des différentes attaques en fonction de W. De façon générale, ce sont les attaques ESD_W (retrouver la clef secrète en raccourcissant le code) et CC_w (attaque de Canteaut-Chabaud sur une instance) qui vont déterminer de bons paramètres.*

– coût du déchiffrement : $\mathcal{O}\left((n-W)^2 \log^2 q\right)$ opération binaires

Il convient maintenant de déterminer le coût asymptotique des attaques. La première chose à remarquer est que tant que l'on choisit W de façon à avoir une instance difficile de DRS pour retrouver la clef secrète on a $W > n - \sqrt{nk}$ et ceci fait que w est nécessairement plus petit que W. Ainsi, ISD_W sera toujours moins efficace que ISD_w (et de même pour les attaques CC). Il n'est donc pas nécessaire de regarder le coût de cette attaque. De même, asymptotiquement CC_w et ISD_w sont presque équivalentes, pour plus de simplicité nous ne regarderons donc que ISD_w. Comme on le voit sur la figure 7-2 les attaques ISD_w et ESD_W étant monotones elles se croisent au point de sécurité maximale. Il suffit de déterminer ce point d'intersection pour connaître la sécurité du système.

On cherche donc la solution de :

$$\frac{\binom{n}{k}}{\binom{n-w}{k}} = \frac{\binom{n}{\beta}}{\binom{W}{\beta}} \quad \text{avec} \quad \beta = n - \frac{(n-W)^2}{k} \quad \text{et} \quad w = n - W - \sqrt{(n-W)k}$$

Il faut donc :

$$(n-w-k)!(n-\beta)!W! = (W-\beta)!(n-k)!(n-w)!$$

Il n'est malheureusement pas possible de déduire une expression simple du W optimal en fonction des autres paramètres de cette équation. Il n'est donc pas non plus possible de donner la complexité asymptotique des attaques en fonction de n et k quand on choisit le W optimal.

6) *Passage à l'échelle*

Il semble donc difficile de trouver une formule exacte déterminant la sécurité optimale du système. En revanche, on peut constater que si on a une solution (n_0, k_0, W_0) optimale pour la sécurité, si on multiplie par un facteur λ tous les paramètres on aura encore une solution optimale.

En effet, en multipliant n, k et W par λ, w et β sont eux aussi multipliés par λ, et on obtient alors :

$$f_\lambda(n,k,W) = \frac{(\lambda n - \lambda w - \lambda k)!(\lambda n - \lambda\beta)!(\lambda W)!}{(\lambda W - \lambda\beta)!(\lambda n - \lambda k)!(\lambda n - \lambda w)!}.$$

Il suffit donc de vérifier que f_λ est toujours égal à 1 (au premier ordre). On utilise l'équivalent donné par la formule de Stirling en négligeant tous les termes polynomiaux :

$$n! \approx \frac{n^n}{e^n},$$

et on obtient :

$$f_\lambda(n,k,W) = \frac{(\lambda n - \lambda w - \lambda k)^{\lambda n - \lambda w - \lambda k}(\lambda n - \lambda\beta)^{\lambda n - \lambda\beta}(\lambda W)^{\lambda W}}{(\lambda W - \lambda\beta)^{\lambda W - \lambda\beta}(\lambda n - \lambda k)^{\lambda n - \lambda k}(\lambda n - \lambda w)^{\lambda n - \lambda w}}$$

Les termes e^n ont tous disparu du fait que les sommes des éléments présents au numérateur et au dénominateur sont égales. De même on peut diviser tous les termes par λ pour obtenir :

$$f_\lambda(n,k,W) = \frac{(n - w - k)^{\lambda n - \lambda w - \lambda k}(n - \beta)^{\lambda n - \lambda\beta}W^{\lambda W}}{(W - \beta)^{\lambda W - \lambda\beta}(n - k)^{\lambda n - \lambda k}(n - w)^{\lambda n - \lambda w}}$$

Enfin, si on factorise le λ en puissance on trouve :

$$f_\lambda(n,k,W) = \left[\frac{(n - w - k)^{n - w - k}(n - \beta)^{n - \beta}W^{W}}{(W - \beta)^{W - \beta}(n - k)^{n - k}(n - w)^{n - w}}\right]^\lambda$$

Donc si des paramètres vérifient $f_1(n_0,k_0,W_0) = 1$, alors pour tout λ on aura $f_\lambda(n_0,k_0,W_0) = 1^\lambda = 1$.

On peut donc augmenter linéairement tous les paramètres du système et rester au point optimal. En utilisant la même approximation grossière que précédemment, la sécurité s'exprime en fonction de λ comme :

$$\text{Complexité}_\lambda = \frac{\binom{\lambda n}{\lambda k}}{\binom{\lambda(n-w)}{\lambda k}} \approx \left[\frac{\binom{n}{k}}{\binom{n-w}{k}}\right]^\lambda = \text{Complexité}_1^\lambda$$

En conclusion, on peut donc dire que quand on a un jeu de paramètres qui convient bien, si on multiplie tous ces paramètres linéairement par λ, le logarithme de la sécurité sera lui aussi multiplié par λ.

Notons qu'avec les complexités obtenues précédemment, l'introduction de λ va faire augmenter la taille de clef et de bloc linéairement en λ, le taux de transmission reste constant et les coûts de chiffrement et de déchiffrement augmentent eux quadratiquement.

c) Choix des paramètres initiaux

On sait que l'on peut facilement faire passer à l'échelle un jeu de paramètres afin d'obtenir la sécurité que l'on veut. En revanche il reste plusieurs choix pour les paramètres initiaux. Les paramètres $n = 1024$, $k = 900$ et $W = 74$ sont ceux qui offrent le meilleur taux de transmission pour une sécurité de 2^{80} en longueur 1024. On pourrait en revanche chercher à maximiser la sécurité pour une même longueur, ou encore augmenter W pour diminuer le coût du décodage.

Pour maximiser la sécurité il faut choisir $n = 1024$ et k autour de 650, ce qui donne, pour le W optimal, une sécurité autour de 2^{120}. Pour minimiser le coût du déchiffrement il faut prendre un petit k. Par exemple pour $k = 320$ on peut prendre $W = 470$ et on obtient encore une sécurité de 2^{80}. Le coût du déchiffrement est en revanche divisé par 3 (et le taux de transmission aussi).

Il reste donc aussi beaucoup de liberté dans le choix des paramètres initiaux que l'on veut faire passer à l'échelle, selon ce que l'on vise pour le système final.

7-5 Réduction de la taille de clef

La taille de clef et la taille de bloc sont les deux principaux points faibles de ce système. On ne peux pas diminuer la taille de bloc facilement car n et q sont déjà choisis proches de leur minimum : un n plus petit rend le système vulnérable aux attaques par décodage et un q plus petit va rendre la recherche exhaustive sur α trop facile. En revanche, comme vu dans la section précédente, les attaques type ISD_W pour retrouver la clef sont bien plus difficiles que les autres : on peut certainement gagner quelque chose tout en les rendant plus faciles.

7-5.1 Utilisation d'un sous-corps

Pour réduire la taille de clef, on peut choisir de prendre la clef publique dans un sous-corps de $\mathbb{F}_q$. Cela oblige aussi à choisir le support du code dans ce même sous-corps, mais cela ne devrait rien changer à la sécurité du système : on continue à choisir α et m dans $\mathbb{F}_q$.

Soit q_0 tel que $\log_2 q_0$ divise $\log_2 q$, la taille du sous-corps utilisé. On prend les n éléments du support dans $\mathbb{F}_{q_0}$ et on prend pour la clef un polynôme $\mathcal{P}_c$ unitaire de degré $k-1$ sur $\mathbb{F}_{q_0}$. Cela nous donne donc un c appartenant à $\mathbb{F}_{q_0}^n$ auquel on n'a plus qu'à ajouter une erreur de poids W elle aussi dans le sous-corps. La taille de clef passe alors d'une taille nq à une taille nq_0 qui reste cependant nécessairement plus grande que n^2 puisque tout le support doit être dans $\mathbb{F}_{q_0}$. Avec cette méthode seules les attaques ISD_W et CC_W sont rendues plus faciles, la modification est transparente pour les autres attaques.

Si on veut diminuer la taille de clef encore plus c'est possible en prenant la clef dans un sous-code sur un sous-corps de $\mathbb{F}_{q_0}$. Si on prend $\mathbb{F}_{q_1}$ un sous-corps de $\mathbb{F}_{q_0}$ en conservant toujours un support dans $\mathbb{F}_{q_0}$, cela induit un sous-code de dimension k' plus petite que k. On a $(n-k') = \log_{q_1} q_0(n-k)$ ce qui peut faire chuter la dimension du code dramatiquement et rendre les attaques très faciles. En revanche, encore une fois, cela n'affecte a priori pas les autres attaques sur le système. Donc si q_0 et q_1 sont choisis de façon à conserver les attaques ISD_W et CC_W au dessus de 2^{80} opérations cela ne devrait pas poser de problèmes. On peut alors avoir une taille de clef de nq_1.

Par exemple, avec les paramètres précédents $n = 1024$, $k = 900$ et $W = 74$, au lieu de prendre $\mathbb{F}_q = \mathbb{F}_{2^{80}}$ on peut choisir $\mathbb{F}_q = \mathbb{F}_{2^{84}}$, $\mathbb{F}_{q_0} = \mathbb{F}_{2^{12}}$ et $\mathbb{F}_{q_1} = \mathbb{F}_{2^3}$ en conservant la sécurité de 2^{80}. Au passage, on réduit la taille de clef publique de 80 kbits à 3 kbits, ce qui devient tout à fait acceptable.

7-5.2 Attaque utilisant la trace

Le problème de cette réduction de taille de clef est que, même si elle n'avantage effectivement en rien un attaquant utilisant $\mathcal{ESD}_W$ ou $\mathcal{CC}_w$, elle va par contre rendre beaucoup plus facile la recherche exhaustive sur α. L'attaquant va pour cela utiliser l'application trace $\mathrm{Tr} : \mathbb{F}_q \to \mathbb{F}_{q_0}$. En l'appliquant à un chiffré on obtient par linéarité :

$$\mathrm{Tr}(y) = \mathrm{Tr}(c_m) + \mathrm{Tr}(\alpha(c+E)) + \mathrm{Tr}(e)$$

De plus, comme (c+E) est dans $\mathbb{F}_{q_0}$ on a :

$$\mathrm{Tr}(y) = \mathrm{Tr}(c_m) + \mathrm{Tr}(\alpha) \times (c+E) + \mathrm{Tr}(e)$$

La trace d'un mot du code de Reed-Solomon sur $\mathbb{F}_q$ étant un élément du Reed-Solomon sur $\mathbb{F}_{q_0}$, on se retrouve avec un problème tout à fait similaire au problème de départ mais sur $\mathbb{F}_{q_0}$. Si on fait alors une recherche exhaustive sur $\mathrm{Tr}(\alpha)$ on a une attaque de complexité q_0 qui permet de retrouver $\mathrm{Tr}(\alpha)$ et donc aussi $\mathrm{Tr}(e)$ et $\mathrm{Tr}(c_m)$. Une fois que l'on a $\mathrm{Tr}(e)$ on connaît les positions non nulles de e (ou du moins une bonne partie) et on peut alors décoder l'instance initiale du problème et retrouver m.

Même si cette réduction de taille de clef semblait attirante il faudra donc s'en passer.

7-6 Attaque de Coron

Cette attaque, présentée par Jean-Sébastien Coron [20], montre qu'il y a mieux à faire pour retrouver α qu'une simple recherche exhaustive. Elle a l'horrible inconvénient de toujours pouvoir s'appliquer au système si w est choisi de façon à pouvoir décoder de façon unique (c'est-à-dire quand $w \leq \frac{n-W-k}{2}$). Elle ne s'applique a priori pas quand on utilise l'algorithme de Sudan pour déchiffrer, mais Kiayias et Yung [57] ont montré qu'une modification de l'algorithme de Guruswami-Sudan permettait aussi d'attaquer tous les cas où $w \leq n - W - \sqrt{(n-W)k}$.

7-6.1 Description de l'attaque

Cette attaque est une variante de l'algorithme de décodage de Berlekamp-Welch [13]. L'idée est de réécrire le système sous forme polynomiale, coordonnée du support par coordonnée du support. On écrira $(x_i)_{i=1..n}$ les éléments du support et K_i, y_i et e_i les coordonnées respectives de la clef publique $K = c+E$, du chiffré y et de l'erreur e. On a alors :

$$\forall i \in [1..n] \quad y_i = \mathcal{P}_m(x_i) + \alpha K_i + e_i.$$

Soit $\mathcal{L}_e$ le polynôme localisateur de e. Ce polynôme inconnu est unitaire de degré w et contient donc w coefficients inconnus. Si on multiplie l'équation précédente par ce polynôme on trouve :

$$\forall i \in [1..n] \quad y_i \mathcal{L}_e(x_i) = \mathcal{P}_m(x_i)\mathcal{L}_e(x_i) + \alpha K_i \mathcal{L}_e(x_i),$$

et donc :

$$\forall i \in [1..n] \quad (y_i - \alpha K_i)\mathcal{L}_e(x_i) = \mathcal{P}_m(x_i)\mathcal{L}_e(x_i). \tag{7-1}$$

Afin de linéariser cette équation on introduit le polynôme $\mathcal{N} = \mathcal{P}_m \times \mathcal{L}_e$. Il est de degré $k-2+w$ et possède donc $k-1+w$ coefficients inconnus. L'équation 7-1 peut alors s'écrire :

$$\forall i \in [1..n] \quad (y_i - \alpha K_i)\mathcal{L}_e(x_i) - \mathcal{N}(x_i) = 0. \tag{7-2}$$

Toutes les solutions de 7-1 donnent une solution de 7-2, en revanche certaines solutions de 7-2 peuvent ne pas correspondre à une solution de 7-1. Il faudra donc trouver toutes les solutions de 7-2 pour ensuite faire le tri.

Si on considère α comme un paramètre on se retrouve avec un système linéaire avec pour inconnues les $w+1$ coefficients de $\mathcal{L}_e$ (on suppose qu'il n'est pas nécessairement unitaire pour garder un système linéaire au lieu d'un système affine) et les $k-1+w$ coefficients de $\mathcal{N}$. On dispose de n équations dont au minimum $n-W$ sont indépendantes (tous les éléments du support pour lesquels E_i est nul donnent des équations indépendantes), et comme $w \leq \frac{n-W-k}{2}$ on a juste ce qu'il faut comme équations. On cherche comme solution un élément non nul du noyau de ce système, il est donc nécessaire que le noyau soit non trivial et donc que le déterminant du système soit nul.

Le déterminant du système est un polynôme de degré $w+1$ en α que l'on peut calculer. Les seuls α donnant une solution non nulle au système font partie des racines de ce polynôme. On trouve donc au plus $w+1$ valeurs possibles de α que l'on n'a plus qu'à essayer l'une après l'autre afin de déchiffrer le message.

Un problème peut tout de même se poser : on sait qu'il y a au moins une solution au système 7-2 puisqu'il y en a au moins une au système 7-1, en revanche rien ne dit que le déterminant n'est pas identiquement nul. Si tel est le cas on n'a alors plus aucune information sur α car tous donneront une solution non nulle pour $\mathcal{N}$ et $\mathcal{L}_e$.

Dans la pratique, étant donné que l'on travaille sur un corps très grand, la probabilité de tomber sur un déterminant toujours nul est négligeable. En revanche il est peut-être possible de construire des instances du problème pour lesquelles le polynôme est toujours nul, à savoir : le système est toujours de rang trop petit, quelle que soit la valeur de α. Rien ne dit cependant que cela n'apportera pas de nouvelles faiblesses au système. Cette voie reste tout de même à explorer.

7-6.2 Une version multi-clefs

Pour parer cette attaque on peut imaginer utiliser plusieurs clefs publiques ayant la même localisation de l'erreur. On transformerait ainsi le système de l'équation 7-2 en un système à plusieurs paramètres. Sa résolution serait donc beaucoup plus compliquée.

a) Construction

On va utiliser ν couples clef publique/clef privée. On génère donc ν couples $(c^{(i)}, E^{(i)})_{i=1..\nu}$ distincts pour des erreurs $E^{(i)}$ ayant toute le même support (c'est nécessaire pour pouvoir raccourcir le code et déchiffrer) et des mots de code $c^{(i)}$ évaluation de polynômes unitaires de degré $k-i$ (on a besoin de degrés étagés pour facilement retrouver tous les $\alpha^{(i)}$).

Le chiffrement est alors :

$$y = c_m + \sum_{i=1}^{\nu} \alpha^{(i)} \left(c^{(i)} + E^{(i)} \right) + e.$$

Le déchiffrement se passe comme précédemment en raccourcissant pour retrouver e puis en retrouvant tous les $\alpha^{(i)}$ successivement. Pour que cela fonctionne il faut bien sûr prendre un message de plus bas degré que précédemment : ici il faut que $\mathcal{P}_m$ soit de degré au plus $k - \nu - 1$.

b) Nouvelle attaque sur la clef

On fournit ν clefs correspondant à des mots de codes bruités sur les mêmes positions. De ce fait, retrouver ces positions est plus simple que pour un seul mot. En modifiant légèrement l'algorithme de Berlekamp-Welch on peut alors non plus décoder $\frac{n-k}{2}$ erreurs mais $\frac{\nu}{\nu+1}(n-k)$ erreurs. L'algorithme de Guruswami-Sudan doit lui aussi certainement pouvoir s'adapter pour corriger plus d'erreurs dans ce cas-là, mais j'ignore encore comment pour l'instant. Dans tous les cas cela nous oblige à prendre un W beaucoup plus grand qu'avant et donc, par la même occasion un w plus petit.

c) Taille de clef

On a aussi multiplié la taille de la clef par ν dans cette opération. On peut éviter cela en modifiant un peu le chiffrement et en faisant encore une fois intervenir la trace. Cela permet de n'utiliser qu'une seule clef, mais qui sera projetée de différentes façons sur un sous-corps pour rendre le même effet que plusieurs clefs sans a priori faciliter aucune des autres attaques. En revanche la remarque précédente tient encore et cela oblige aussi à prendre un W plus grand. De plus cela limite la valeur de ν car il faut toujours que le support du code soit inclus dans le sous-corps ν fois plus petit que $\mathbb{F}_q$ et donc $n \leq 2^{\frac{\log_2 q}{\nu}}$.

7-6.3 Adaptation de l'attaque de Coron

Malheureusement, même avec plusieurs clefs l'attaque de Coron tient encore [21]. On réécrit les équations coordonnée par coordonnée et on trouve :

$$\forall i \in [1..n] \quad y_i = \mathcal{P}_m(x_i) + \sum_{j=1}^{\nu} \alpha^{(j)} K_i^{(j)} + e_i. \tag{7-3}$$

Encore un fois nous allons introduire $\mathcal{L}_e$ le polynôme localisateur de e et $\mathcal{N} = \mathcal{L}_e \times \mathcal{P}_m$ et multiplier l'équation par $\mathcal{L}_e$. On obtient :

$$\forall i \in [1..n] \quad y_i \mathcal{L}_e(x_i) = \mathcal{N}(x_i) + \sum_{j=1}^{\nu} \alpha^{(j)} \mathcal{L}_e(x_i) K_i^{(j)}.$$

Maintenant, si comme précédemment on considère les $\alpha^{(j)}$ comme des paramètres on va obtenir des équations de degré ν, qui risquent de poser quelques problèmes de résolution. Pour simplifier cela on va de nouveau linéariser le système en introduisant les polynômes $(\mathcal{R}^{(j)})_{j=1..\nu}$ avec $\mathcal{R}^{(j)} = \alpha^{(j)} \mathcal{L}_e$. Ces polynômes sont de degré w et introduisent donc chacun $w+1$ inconnues au lieu

d'une seule inconnue $\alpha^{(j)}$. En revanche, on a maintenant un système linéaire :

$$\forall i \in [1..n] \quad y_i \mathcal{L}_e(x_i) = \mathcal{N}(x_i) + \sum_{j=1}^{\nu} K_i^{(j)} \mathcal{R}^{(j)}(x_i). \tag{7-4}$$

Notons qu'encore une fois une solution de l'équation 7-3 induit toujours une solution à l'équation 7-4. Le système 7-4 a donc nécessairement une solution non nulle dans son noyau qu'il reste à déterminer.

Le système est constitué de n équations en $k + w - \nu + (\nu + 1)(w + 1) = k + w(\nu + 2) + 1$ inconnues ($\mathcal{N}$ est de degré au plus $k + w - \nu - 1$ et $\mathcal{L}_e$ et les $\mathcal{R}^{(j)}$ sont de degré w). Le système n'est pas de rang maximal puisque l'on sait qu'il y a au moins une solution. En revanche, s'il est de rang exactement $k + w(\nu + 2)$ (un de moins que le nombre d'inconnues), l'ensemble des solutions sera de dimension 1. Dans ce cas, n'importe quelle solution permet de retrouver m, simplement en calculant $\mathcal{N}/\mathcal{L}_e$ avec les valeurs trouvées par cette solution (on supprime le coefficient linéaire que peut avoir introduit l'espace vectoriel).

On sait donc que si le système 7-4 est de rang $k + w(\nu + 2)$ on pourra alors attaquer le système. Cela n'est pas garanti car la présence des $K_i^{(j)}$ peut faire tomber le rang à $n - W$ (la longueur du code raccourci sur l'erreur). Cependant, vu que l'on travaille sur un corps de très grande taille, la probabilité que cela fasse chuter le rang même de 1 est négligeable si les $K_i^{(j)}$ sont choisis aléatoirement. On peut donc considérer que le rang est maximal et il suffit donc pour pouvoir utiliser cette attaque d'avoir :

$$n \geq k + (\nu + 2)w \tag{7-5}$$

Cependant d'autre conditions doivent aussi être remplies par les variables. On a vu à la section 7-6.2 page 100 que pour résister aux attaques sur la clef il fallait :

$$W > \frac{\nu}{\nu + 1}(n - k) \tag{7-6}$$

Enfin, pour pouvoir déchiffrer le message il faut aussi :

$$w \leq \frac{n - W - k}{2} \tag{7-7}$$

En mettant ensemble les 3 équations 7-5, 7-6 et 7-7 on aboutit à la constatation suivante : il est toujours possible d'appliquer une des deux attaques si on veut pouvoir déchiffrer. On aboutit donc à la même conclusion que pour la version à une seule clef publique : si on ne se place pas volontairement dans un cas dégénéré, il y a toujours une attaque qui permet de déchiffrer les messages.

7-7 Conclusion

Nous avons construit un nouveau système de chiffrement basé sur les codes qui est assez efficace en vitesse de chiffrement et déchiffrement, mais qui, comme tous les systèmes utilisant des codes, possède les désavantages d'avoir une clef relativement grosse et un taux de transmission plus petit que 1. Ce système est malheureusement cassé et semble difficile à réparer sans modifications majeures. Il introduit cependant un nouveau concept de chiffrement à clef publique dont

la sécurité réside dans la façon de cacher une instance facile d'un problème dans une instance difficile. Ici nous utilisons le problème de Polynomial Reconstruction pour lequel la limite entre les instances faciles et difficiles a l'avantage d'être bien connue. Malheureusement la façon de cacher l'instance facile dans l'instance difficile possède une faille. Cependant, il est peut-être possible d'adapter cette idée à d'autres problèmes possédant aussi des instances faciles et d'autres difficiles : il n'y a pas alors besoin d'introduire une trappe dans une instance difficile, mais juste une trappe dans la façon de cacher l'instance facile.

Chapitre 8

Une famille de fonctions de hachage cryptographiques rapides à sécurité prouvée

Une grande majorité des fonctions de hachage utilisées aujourd'hui en cryptologie est construite sur un même modèle : l'itération d'une fonction de compression selon le schéma présenté en 1989 par Damgård et Merkle [26, 67]. Dans la plupart des cas, ces fonctions de compression sont construites dans le but d´être très rapides et utilisent pour cela des mécanismes empruntés aux systèmes de chiffrement symétrique [83, 69]. D'autres constructions, empruntant des techniques de la cryptographie à clef publique, permettent d'obtenir des fonction de compression dont la sécurité est prouvée [37], cependant, cela se fait toujours au détriment de la vitesse et ces fonctions sont beaucoup trop lentes pour être d'une utilité pratique.

Dans ce chapitre je présente une famille de fonctions de hachage présentant les deux propriétés précédemment citées : la rapidité et une sécurité s'appuyant sur des problèmes biens définis. La construction utilisée est encore celle du schéma de Damgård et Merkle et la fonction de compression peut être vue comme une variante du cryptosystème de Niederreiter [70].

Nous verrons dans un premier temps comment construire une telle fonction, puis comment la rendre à la fois plus rapide et plus sûre. Ensuite une étude des meilleures attaques sur cette construction nous permettra de déterminer les paramètres minimums permettant d'obtenir une sécurité suffisante.

Les résultats présentés dans ce chapitre n'étant pas encore officiellement publiés, seule une version partielle est disponible via le serveur d'ePrint de l'IACR [7].

8-1 Généralités sur les fonctions de hachage cryptographiques

Une fonction de hachage $\mathcal{H}$ est une fonction de ${\mathbb{F}_2}^*$ dans ${\mathbb{F}_2}^r$ qui prend en entrée un document de longueur quelconque et retourne en sortie une suite binaire de longueur donnée. Le but d'une telle fonction est de garantir l'*intégrité* d'un document, c'est-à-dire, permettre d'être certain qu'un document donné correspond bien à ce que l'on attend et qu'il n'a pas été modifié. Pour cela, la personne qui cherche à diffuser un document va rendre public le *haché* du document : son image par la fonction de hachage. Si la fonction de hachage vérifie les deux propriétés suivantes, alors il suffira de vérifier que le document reçu a le bon haché pour être certain de son intégrité. Pour cela elle doit donc être :

- *non-inversible* : pour un haché h donné, il doit être calculatoirement impossible de trouver un document D tel que $\mathcal{H}(\mathsf{D}) = h$.
- *sans collisions* : il doit être calculatoirement impossible de trouver deux documents D_1 et D_2 tels que $\mathcal{H}(\mathsf{D}_1) = \mathcal{H}(\mathsf{D}_2)$.

Dans certains cas on peut être amené à ne considérer que la première de ces deux propriétés et à ne chercher que la non-inversibilité, mais on n'a plus alors à proprement parler une fonction de hachage.

8-1.1 La construction de Damgård et Merkle

C'est sur cette construction que sont basées à peu près toutes les fonctions de hachage utilisées de nos jours. Au cœur du schéma se trouve une *fonction de compression* : une fonction qui prend une entrée de longueur donnée s et retourne une sortie de longueur r. Comme son nom l'indique cette fonction compresse et il faut donc que s soit strictement plus grand que r. C'est en itérant cette fonction de compression que l'on obtient une fonction qui prend une entrée de taille quelconque.

Dans ce schéma (présenté figure 8-1 page suivante) chaque élément a son importance :

- **le document** va être lu par tranches de longueur fixe en entrée de la fonction de compression,
- **le chaînage** consiste à réutiliser la sortie du tour $n-1$ de la fonction de compression en entrée du $n^{\text{ième}}$ tour. Cela fait que la sortie du $n^{\text{ième}}$ tour dépend de tout le début du document D et pas seulement de la dernière tranche lue.
- **l'initial value** (I.V.) sert uniquement à remplacer les bits de chaînage lors du premier tour. Elle peut être choisie de n'importe quelle façon : par exemple entièrement nulle.
- **le padding** permet de gérer des document de longueur quelconque. En effet la fonction de compression prend s bits d'entrée, dont r de chaînage. Si la longueur du document n'est pas multiple de $s-r$ on n'a pas un nombre de tours entier. Le padding consiste à ajouter des 0 à la fin du document, puis un denier bloc de longueur $s-r$ contenant la longueur initiale du document (afin d'éviter de créer des collisions avec le padding) pour atteindre une longueur adéquate.

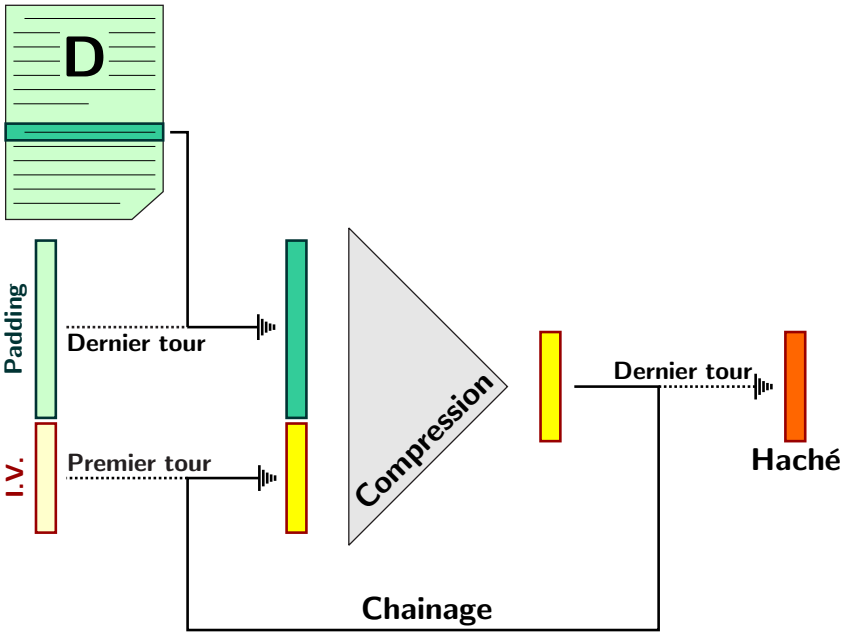


FIG. 8-1 – *Schéma général de construction d'une fonction de hachage de Damgård et Merkle*

Enfin, l'une des plus intéressantes propriétés de cette construction concerne la sécurité : il est prouvé que la sécurité d'une fonction de hachage construite avec ce schéma est au moins aussi élevée que la sécurité de la fonction de compression utilisée. Cela signifie qu'inverser le schéma global nécessite forcément d'inverser au moins une fois la fonction de compression, et trouver des collisions nécessite de trouver des collisions sur la fonction de compression. Cela permet donc de prouver la sécurité du schéma complet en ne prouvant que la sécurité de la fonction de compression.

En effet, on peut imaginer facilement que si on a une collision sur la fonction globale on a aussi une collision sur le dernier tour de la fonction, ou alors, si les deux documents sont égaux, sur celui d'avant... On peut ainsi prouver que si les deux documents sont différents, il y a nécessairement une collision sur un tour de la fonction de compression et on en déduit que si on sait trouver des collisions sur la fonction globale, on saura donc aussi le faire sur la fonction de compression.

8-1.2 Quelques fonctions connues

Les fonctions de hachage les plus utilisées de nos jours sont MD5 et SHA-1. Elles utilisent toutes les deux le schéma de chaînage de la figure 8-1 : MD5 une sortie de 128 bits pour une entrée de 128+512 et SHA-1 avec une sortie de 160 bits pour une entrée de 160+512.

MD5 (Message Digest 5) a été proposée par R. Rivest en 1992 [83] et est une évolution de MD4 [82] (pour laquelle des collisions ont été trouvées depuis [30]). Quelques attaques existent sur cette fonction [27, 29] mais elle est encore très

utilisée pour des applications où la sécurité n'est pas primordiale. Elle est de nos jours essentiellement utilisée pour garantir l'authenticité de fichiers sur internet. Sa construction repose sur le schéma de Davies-Meyer, c'est-à-dire que le fonction de compression est en fait une fonction de chiffrement de 128 bits vers 128, utilisant une clef de 512 bits.

SHA-1 est elle une évolution de SHA (Secure Hash Algorithm) publiée par le gouvernement américain en 1993 [69]. Sa construction repose sur celle de MD5, mais est plus sûre puisqu'elle utilise 160 bits. De ce fait c'est cette fonction qui est la plus utilisée dans les schémas de signature. Il en existe aussi trois autres versions : SHA-256, SHA-384 et SHA-512 hachant respectivement en 256, 384 et 512 bits. Une faiblesse présente dans SHA et mise à jour par F. Chabaud et A. Joux [18] a disparu dans chacune de ces versions.

Une grande variété d'autres fonctions de hachage existent mais sont beaucoup moins utilisées. Certaines font appel à des principes similaires [31, 65, 102], d'autres constructions cherchent, au contraire, à atteindre des buts différents : par exemple, la fonction décrite dans [37], en s'appuyant sur des problèmes de logarithmes discrets, permet d'obtenir une preuve de sécurité au détriment de la vitesse de hachage.

8-2 Une nouvelle fonction de compression

Notre but étant de construire une fonction de hachage rapide à sécurité prouvée il va falloir construire une fonction de compression dont on peut prouver la sécurité et qui soit suffisamment rapide. Pour cela il faut non seulement qu'un tour de la fonction de compression soit rapide, mais aussi que la fonction compresse beaucoup (prenne beaucoup de bits de entrée) afin de ne pas avoir à faire un trop grand nombre de tours au total.

8-2.1 Utilisation du chiffrement de Niederreiter

Le cryptosystème de Niederreiter [70] a, par rapport à la plupart des autres cryptosystèmes à clef publique, l'avantage d'avoir un chiffrement très rapide. On peut donc imaginer utiliser une fonction similaire dans notre fonction de compression et ainsi bénéficier à la fois de la rapidité du système et de sa sécurité.

Comme vu section 2-2.2 page 27, dans le système de Niederreiter le chiffrement se fait en utilisant une matrice de parité d'un code de Goppa. On convertit le message à chiffrer en un mot de longueur n et de poids w donné que l'on multiplie par la matrice de parité. Le résultat est un syndrome qui va servir de chiffré. La matrice de parité est supposée être indistinguable d'une matrice aléatoire et seule la connaissance de sa structure de code de Goppa et de la permutation utilisée pour la brouiller permet de déchiffrer le syndrome.

Pour utiliser cette fonction de chiffrement en tant que fonction de compression on n'a plus besoin de la trappe (au contraire, on veut même être certain que personne ne peut inverser la fonction de compression) et on peut donc prendre une véritable matrice aléatoire $\mathcal{H}$ de taille $r \times n$. La fonction de compression se décompose alors en deux étapes :

1. convertir les $s = \log_2 \binom{n}{w}$ bits d'entrée en un mot de longueur n et poids w,
2. multiplier ce mot par la matrice $\mathcal{H}$ pour obtenir un syndrome de longueur r.

8-2.2 Sécurité d'une telle fonction de compression

Pour une fonction de hachage, la sécurité se mesure en terme de résistance à l'inversion et à la recherche de collisions. L'inversion consiste à retrouver une entrée de la fonction de compression correspondant à une sortie donnée. Dans notre cas il s'agit, à partir d'un syndrome $\mathcal{S}$ donné de longueur r, de retrouver un mot e de poids w tel que $\mathcal{H} \times e^T = \mathcal{S}$. C'est exactement le problème de *syndrome decoding* (SD) énoncé chapitre 6 page 77.

Pour ce qui est de la recherche de collisions, il va s'agir de trouver deux mots e_1 et e_2 de poids w ayant le même syndrome. Cela revient à trouver un mot $e = e_1 + e_2$ de poids $2w$ (ou moins si e_1 et e_2 ont des positions communes) ayant un syndrome nul. Ici aussi on se retrouve exactement avec une instance du problème SD à résoudre.

D'un point de vue théorique, la sécurité de la fonction de compression (et donc de la fonction de hachage construite avec) repose sur un problème NP-complet bien identifié : le problème SD. Cependant cela ne suffit pas pour garantir la sécurité du schéma, il faut aussi choisir des paramètres tels que les instances du problème SD qu'un attaquant aurait à résoudre soient suffisamment difficiles pour être considérées comme calculatoirement impossibles.

a) Pour l'inversion

On est dans le cas le plus général du problème SD, c'est-à-dire qu'on s'attaque à des instances quelconques, sans aucune particularité : la matrice est bien quelconque, le syndrome aussi. Les meilleures attaques pour l'inversion seront donc les mêmes attaques que sur le problème SD général.

b) Pour les collisions

On ne s'intéresse qu'à la résolution du problème SD avec des syndromes nuls. C'est donc à un sous-ensemble des instances du problème que l'on s'attaque et des algorithmes plus spécifiques pourraient donc exister. Cependant, on peut en fait ramener n'importe quelle instance au cas où le syndrome est nul et ces instances ne sont pas plus faciles que les autres. Encore une fois on sera donc amené à utiliser des attaques génériques.

8-2.3 Efficacité de cette fonction de compression

Comme pour le cryptosystème de Niederreiter, en utilisant cette construction, la partie la plus coûteuse d'un tour de la fonction de compression est la conversion en mots de poids constant. En effet, pour lire $\log_2 \binom{n}{w}$ bits et les convertir, de façon bijective, en w indices compris entre 0 et n, il est nécessaire d'effectuer des calculs sur des grands entiers (voir section 5-4.1 page 63). Le hachage à proprement parler n'étant constitué que de quelques additions (XORs) de colonnes de $\mathcal{H}$ ira beaucoup plus vite.

Si on veut pouvoir aller plus vite, il va falloir abandonner la conversion bijective et se contenter de codages moins efficaces. Se posent alors deux problèmes :

- on va lire moins de bits en entrée et on risque de devoir faire plus de tours de hachage,
- l'espace image des hachés possibles sera un sous-ensemble de l'espace des syndromes. On sort donc du contexte général du problème SD.

8-3 Variations sur le codage en poids constant

Pour accélérer notre fonction de compression nous voulons utiliser un codage en poids constant non bijectif le plus rapide possible. Deux choix s'offrent alors à nous : essayer de conserver un rendement le meilleur possible pour ne pas trop augmenter le nombre de tours et ne pas trop s'éloigner du problème SD ; ou alors choisir d'aller le plus vite possible même avec un mauvais rendement.

Dans le premier cas on peut utiliser des techniques de codage de source pour obtenir des rendements les plus élevés possibles, dans le deuxième il faudrait lire les positions dans le fichier avec le minimum de calculs supplémentaires.

8-3.1 Technique utilisant un codage de source

On considère que le fichier à hacher est une source binaire qui émet avec la même probabilité des 1 ou des 0. Notre but est d'interpréter cette source comme la description de mots binaires de longueur n et poids w.

Dans le cas du codage bijectif, il est nécessaire de lire un bloc entier de la source pour pouvoir commencer les calculs et trouver les w positions codées par ce bloc. Pour diminuer les calculs il faudrait pouvoir déduire chaque position d'une partie seulement du bloc. Si on code directement chaque position par un certain nombre de bits il va être difficile de gérer correctement les permutations d'un même lot de w positions. Le plus simple pour éviter ce problème est de coder les écarts des positions. Ainsi les premiers bits vont coder la première position, les suivants la deuxième par rapport à la première et ainsi de suite.

Se pose alors un deuxième problème : si on décide d'utiliser un nombre fixe de bits pour chaque écart de positions et que l'on veut être sûr de ne pas dépasser la longueur totale du mot, la première position ne pourra jamais se trouver au-delà de la position n/w. De même, la dernière position se trouvera en moyenne à la moitié de la longueur. On va donc, pour améliorer cela, être obligés de coder les positions avec un nombre variable de bits. De plus, pour conserver une distribution des mots équiprobable, il ne faut pas que le fait d'avoir une première position très proche du début du mot ne tende à rapprocher toutes les autres positions du début. Ce nombre variable devra donc s'adapter à la fois au nombre de positions à choisir dans le mot et à la longueur encore disponible.

Des solutions pour faire cela de façon efficace (avec une très bonne densité de mots représentée) existent [86] mais nécessitent de refaire quelques calculs après la lecture de chaque position et aboutissent à des codages en longueur variable : le nombre de bits lus pour obtenir un mot de poids w n'est pas toujours le même. Dans notre cas cela donnerait une fonction de compression à entrée de taille variable. Cela ne semble pas poser de problèmes de sécurité, mais risque de s'avérer assez compliqué à mettre en œuvre au niveau du padding.

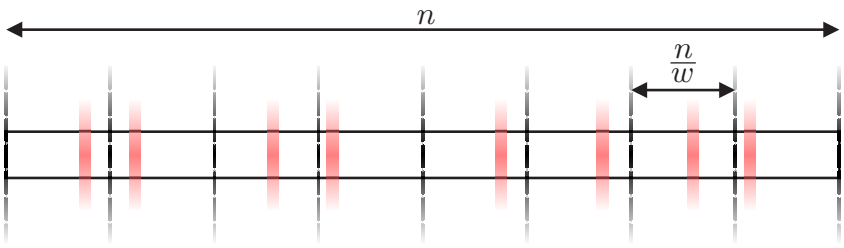


FIG. 8-2 – *Forme générale d'un mot régulier.*

On peut donc avoir des rendements très proches de celui de la fonction bijective, mais cela nécessite encore des calculs qui vont ralentir le hachage et cela risque aussi de poser des problèmes de mise en œuvre.

8-3.2 Méthode sans calculs

Le but de cette méthode et de connaître directement les w positions non nulles du mot en lisant le document, et sans aucun calcul. On peut pour cela utiliser, comme précédemment, les écarts entre les positions et dire qu'un nombre fixe de bits $s = \log_2\left(\frac{n}{w}\right)$ va coder la distance entre la $i^{\text{ème}}$ position et la $(i+1)^{\text{ème}}$. Cependant, cette technique fait que certaines positions (vers la fin du mot) ont une probabilité très faible d'être atteintes, et cela va beaucoup simplifier le travail d'un attaquant éventuel : la sécurité du système sera équivalente à la sécurité du même système mais en longueur plus courte.

On peut garder exactement le même rendement que cette méthode mais avec une probabilité égale pour chaque position. Pour cela il suffit d'encore une fois lire $s = \log_2\left(\frac{n}{w}\right)$ bits en entrée pour chaque position, et dire que ces s bits codent la position de la colonne dans le $i^{\text{ème}}$ intervalle de $\frac{n}{w}$ positions. L'ensemble des mots de poids w ainsi construit sera appelé ensemble des *mots réguliers* : le support du mot est coupé en w intervalles qui contiennent chacun une et une seule position non nulle (voir figure 8-2).

Cette méthode de conversion présente l'énorme avantage de ne comporter aucun calcul puisque l'indice de chaque position correspond simplement à la conversion des s bits du document en un entier entre 0 et $2^s - 1$ auquel on ajoute i fois $\frac{n}{w}$, l'indice du début d'intervalle.

Si on désigne par $\mathcal{H}_i$ la $i^{\text{ème}}$ colonne de la matrice $\mathcal{H}$ utilisée pour le hachage, la fonction de compression se décompose alors en ces quelques étapes élémentaires :

1. Initialiser $h = 0$
2. Pour i allant de 0 à $w - 1$
   - lire $s = \log_2\left(\frac{n}{w}\right)$ bits en entrée, les convertir en un entier α pour obtenir l'indice $\alpha_i = \alpha + i \times \frac{n}{w}$
   - calculer $h = h \oplus \mathcal{H}_{\alpha_i}$
3. Renvoyer le haché h.

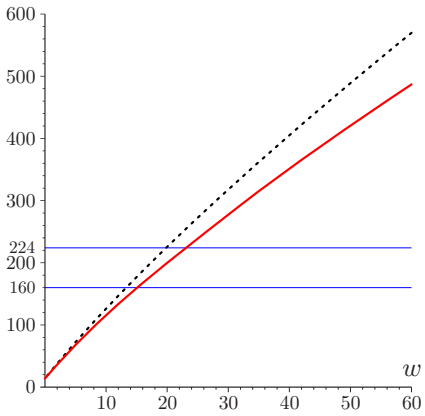


FIG. 8-3 – *Nombre de bits lus en entrée de la fonction de compression en fonction de w pour $n = 2^{14}$. En pointillés la version avec codage bijectif, en trait continu la méthode utilisant des mots réguliers. Les lignes* 160 *et* 224 *représentent des tailles de chaînage possibles.*

Du point de vue du rendement (taux de compression) de la fonction, cette méthode sera en revanche beaucoup moins efficace que le codage bijectif ou la technique utilisant le codage de source. En effet, pour un tour, le nombre de bits lus en entrée devient $w \log_2\left(\frac{n}{w}\right)$ au lieu de $\log_2\binom{n}{w}$. On lit donc à peu près $\log_2(w!)$ bits en moins par tour. On voit sur la figure 8-3 que cet écart est assez important, surtout une fois que l'on déduit les bits issus du chaînage, fixes quelle que soit la méthode. Le hachage d'un document nécessitera donc plus de tours avec le codage en mots réguliers, mais le gain en temps de calcul est tel qu'il compense largement ce surplus.

8-4 Sécurité théorique du codage en mots réguliers

Le codage en mots réguliers simplifie beaucoup le hachage et permet de gagner un facteur important sur le temps nécessaire. Malheureusement, l'utilisation d'un tel codage nous fait sortir du contexte le plus général du problème SD. En effet, en restreignant les solutions aux mots réguliers on change complètement le problème. Cependant, comme nous allons le voir, ce problème est lui aussi NP-complet.

Les attaques auxquelles la construction doit pouvoir résister sont, comme toujours, l'inversion et la recherche de collisions. Pour chacune de ces attaques on peut énoncer un problème s'y ramenant et prouver qu'il est NP-complet.

Pour l'inversion on se donne un syndrome et on cherche simplement un mot régulier qui y correspond. Le problème est le suivant :

Regular Syndrome Decoding : (RSD)
Entrée : une matrice binaire $\mathcal{H}$ de taille $r \times n$, un syndrome $\mathcal{S}$ et un poids w.
Propriété : il existe un mot régulier e de poids w tel que $\mathcal{H}e = \mathcal{S}$.

Pour les collisions, il s'agit de trouver deux mots réguliers distincts e_1 et e_2 ayant le même syndrome, et donc un mot $e = e_1 + e_2$, non nul, ayant un

syndrome nul. Ce mot ne sera pas à proprement parler régulier puisque chaque intervalle peut contenir deux positions ou zéro si e_1 et e_2 coïncidaient dans cet intervalle. J'appellerai un tel mot un mot *2-régulier*. Il a donc un poids $2w'$ avec $0 < w' \leq w$ et w' intervalles parmi les w contiennent deux positions non nulles.

2-Regular Null Syndrome Decoding : (2-RNSD)
Entrée : une matrice binaire $\mathcal{H}$ de taille $r \times n$ et un poids w.
Propriété : il existe un mot 2-régulier e de poids $2w'$ tel que $\mathcal{H}e = 0$.

Pour prouver que ces deux problèmes sont NP-complets, on utilise la même démarche que dans la preuve de NP-complétude du problème SD due à Berlekamp, Massey et van Tilborg [12]. Comme vu à la section 6-1.1 page 78 nous utilisons donc une réduction au problème *Three-Dimensional Matching* (3DM).

a) Réduction de 3DM à RSD

Notre but est de prouver que si l'on sait résoudre toutes les instances de RSD, alors on peut résoudre n'importe quelle instance de 3DM. Pour cela nous avons besoin, à partir d'une instance de 3DM, de pouvoir construire une instance de RSD pour laquelle une solution donne aussi une solution à l'instance de 3DM. Ainsi on aura construit une réduction de 3DM à RSD et prouvé que résoudre RSD est au moins aussi difficile que de résoudre 3DM.

Soit une entrée $U \subseteq T \times T \times T$ du problème 3DM. Soit A la matrice d'incidence $3|T| \times |U|$ décrite à la section 6-1.1 page 78 associée à cette instance. On construit la matrice suivante :

$$\mathcal{H} = \overbrace{\begin{array}{|c|c|c|}\hline A & A & \\ \hline \end{array}}^{|T|\times A} \quad A$$

Si on essaye de résoudre RSD sur cette matrice avec $w = |T|$ et $\mathcal{S} = (1, \ldots, 1)$ une solution existe si et seulement si il est possible de trouver w colonnes de A (possiblement plusieurs fois la même) dont la somme est $\mathcal{S}$. Comme toutes les colonnes de A contiennent exactement trois 1, la seule façon d'obtenir à la fin un poids total de $3 \times |T|$ est qu'aucune paire des w colonnes choisies n'ait de 1 sur une même ligne (chaque fois que deux colonnes ont un 1 sur la même ligne, le poids total final diminue de 2). Dans le cas où une solution existe, elle est donc forcément composée de w colonnes distinctes et ces colonnes n'ont jamais de 1 sur une même ligne.

Si une solution existe, l'ensemble de w colonnes forme donc un sous-ensemble $V \subseteq U$ qui est solution de l'instance initiale de 3DM. De même, si une solution de 3DM existe, elle sera une solution valable pour l'instance de RSD associée. Les deux instances de 3DM et RSD associées sont donc strictement équivalentes.

Si on sait résoudre toutes les instances de RSD, on pourra donc résoudre n'importe quelle instance de 3DM. Le problème RSD est donc lui aussi NP-complet.

b) Réduction de 3DM à 2-RNSD

Comme précédemment, on voudrait construire une matrice pour laquelle résoudre une instance de 2-RNSD est équivalent à résoudre une instance de 3DM. La principale difficulté est qu'on ne peut plus prendre $\mathcal{S} = (1, \ldots, 1)$ puisque le problème ne concerne que les instances où le syndrome $\mathcal{S}$ est nul.

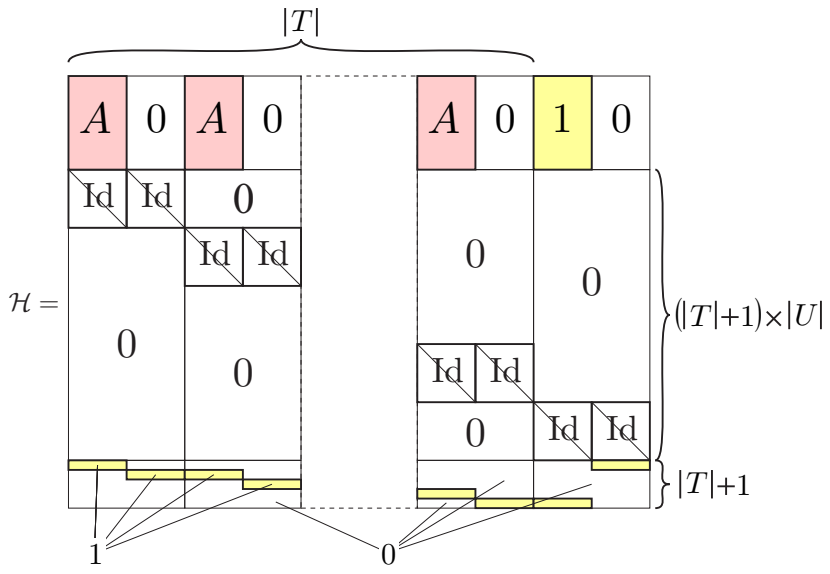


FIG. 8-4 – *Matrice de parité utilisée pour la réduction de 3DM à 2-RNSD.*

Pour cette raison la matrice $\mathcal{H}$ que nous devons construire est un peu plus compliquée. On peut utiliser une matrice de la forme de la figure 8-4.

Cette matrice est composée de trois parties : le haut avec la répétition de matrices A, le milieu avec des paires de matrices identité de taille $|U| \times |U|$, et le bas avec des petites lignes de 1.

Le but de cette construction est de s'assurer qu'une solution de 2-RNSD sur $\mathcal{H}$ (avec $w = |T| + 1$) existe si et seulement si $|T|$ colonnes de A ajoutées à une colonne de 1 donnent bien 0. Ceci est alors équivalent à avoir une solution pour l'instance du problème 3DM sous-jacente.

La partie du haut est celle qui contient le lien à l'instance de 3DM : dans 2-RNSD on prend 2 colonnes dans certains des blocs ; notre but est de prendre 2 colonnes dans chaque bloc, et à chaque fois, une dans le sous-bloc A et une dans le sous-bloc nul.

La partie du milieu fait que quand une solution de 2-RNSD contient une certaine colonne de $\mathcal{H}$ elle doit aussi contenir la seule autre colonne de $\mathcal{H}$ ayant un 1 sur la même ligne, afin que la somme finale soit nulle. De ce fait, à chaque fois qu'une colonne est choisie dans un sous-bloc A, la « même » colonne est choisie dans le sous-bloc nul associé. Ainsi, dans les $2w'$ colonnes formant une solution, w' devront être prises dans les sous-blocs A (ou dans le sous-bloc tout à 1), et w' dans les sous-blocs nuls. Les parties du haut et du milieu suffisent donc pour garantir qu'une solution de 2-RNSD va donner un ensemble de w' colonnes de A ou 1 (pas nécessairement distinctes) de somme nulle.

Enfin, la partie du bas de la matrice va garantir que si $w' > 0$ (comme cela est précisé dans l'énoncé du problème 2-RNSD) alors $w' = w$. En pratique, à chaque fois qu'une colonne est choisie dans le bloc i la partie centrale de la matrice fait qu'il faut en choisir une deuxième dans l'autre moitié de ce bloc, insérant ainsi deux 1 dans la somme finale. Pour les éliminer il est nécessaire de prendre aussi une colonne dans les blocs $i - 1$ et $i + 1$, et ainsi de suite, jusqu'à

avoir pris des colonnes dans chacun des w blocs.

Le résultat est que résoudre une instance de 2-RNSD sur $\mathcal{H}$ est équivalent à choisir $|T|$ colonnes dans A (pas nécessairement différentes) qui ont une somme toute à 1. Comme dans la preuve précédente, cela suffit pour conclure la réduction de 3DM à 2-RNSD, et 2-RNSD est donc NP-complet.

On pourra noter qu'au lieu d'utiliser une réduction de 3DM on aurait pu utiliser RSD pour cette réduction. Il suffit pour cela de remplacer les sous-blocs A pas des w sous-blocs de l'instance de RSD et de remplacer le bloc 1 par un bloc de colonnes toutes égales au $\mathcal{S}$ de RSD. La réduction est alors aussi possible.

8-5 Sécurité pratique : coût des meilleures attaques

Comme pour la fonction construite sur le codage en mots de poids constant bijectif, l'inversion et la recherche de collisions pour le codage en mots réguliers se ramènent à des problèmes NP-complets. Cependant, le coût des attaques sur ce système peut être très différent de celui d'une attaque sur le système avec un codage bijectif. En effet, en restreignant les solutions aux mots réguliers on diminue beaucoup le nombre de solutions, mais on diminue aussi la taille de l'espace de recherche. Il est donc difficile de prévoir comment va évoluer la sécurité du système.

8-5.1 Adaptation des attaques « classiques »

Les meilleures attaques publiées sur le problèmes SD standard passent toutes par la recherche d'ensembles d'informations (voir section 6-3 page 82). Pour avoir une idée de la complexité d'une attaque sur un jeu de paramètres il suffit donc d'évaluer la probabilité qu'un ensemble d'information tiré au hasard soit valide pour l'une des solutions au problème. Il faut pour cela connaître le nombre moyen $\mathcal{N}_w$ de solutions de poids w à une instance du problème, et la probabilité $\mathcal{P}_{w,1}$ de tirer un ensemble d'information valide vis-à-vis d'une certaine solution. Si on considère (de manière abusive, mais cela ne doit pas être significatif) que les probabilités sont indépendantes pour chaque solution on aura alors une probabilité de succès totale de :

$$\mathcal{P}_w = 1 - (1 - \mathcal{P}_{w,1})^{\mathcal{N}_w}.$$

Le coût d'une attaque par ensemble d'information est alors approximativement de $\frac{Q}{\mathcal{P}_w}$, où Q désigne une composante polynomiale en les paramètres du système. Pour la clarté des calculs l'approximation $\mathcal{P}_w \simeq \mathcal{P}_{w,1} \times \mathcal{N}_w$ sera utilisée.

Dans le cas d'une attaque sur le problème SD on va chercher un mot de poids w de syndrome $\mathcal{S}$ donné. La matrice binaire utilisée est de dimension $r \times n$. On a :

$$\mathcal{N}_w = \frac{\binom{n}{w}}{2^r} \quad \text{et} \quad \mathcal{P}_{w,1} = \frac{\binom{n-w}{k}}{\binom{n}{k}} = \frac{\binom{n-k}{w}}{\binom{n}{w}} = \frac{\binom{r}{w}}{\binom{n}{w}},$$

et on trouve donc :

$$\mathcal{P}_w \simeq \frac{\binom{r}{w}}{2^r}.$$

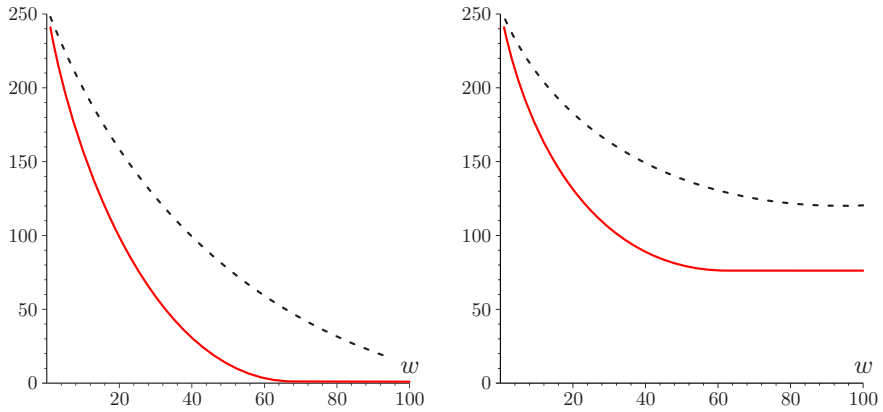


FIG. 8-5 – *Sécurité (logarithme de l'inverse de la probabilité de trouver un ensemble d'information valide) de la fonction de compression contre l'inversion (courbe pointillée) ou la recherche de collisions (courbe continue) en fonction de w, pour une sortie de taille $r = 256$. À gauche la version à codage bijectif, à droite avec le codage en mots réguliers.*

a) Application au schéma utilisant un codage bijectif

Si on veut appliquer cette attaque à la fonction de compression utilisant un codage en mots de poids constant bijectif, il faut alors considérer deux attaques : l'attaque d'inversion et la recherche de collisions. Pour l'inversion on est exactement dans le cas d'une attaque sur le problème SD et la complexité de l'attaque est exactement celle évaluée juste avant :

$$\mathcal{P}_{\text{inv}} = \frac{\binom{r}{w}}{2^r}.$$

Dans le cas de l'inversion on va chercher un mot de poids $2w'$ avec $1 \leq w' \leq w$. Si on considère encore une fois que les probabilités qu'un ensemble d'information donné soit valide sont indépendantes pour toutes les solutions on trouve :

$$\mathcal{P}_{\text{col}} = \sum_{i=1}^{w} \frac{\binom{r}{2i}}{2^r}.$$

La figure 8-5 nous donne une idée de la sécurité d'une fonction de compression utilisant le codage bijectif. On constate qu'il est absolument nécessaire de choisir un w très petit si on veut pouvoir obtenir une sécurité suffisante. Malheureusement, comme on le voit sur la figure 8-3 page 112, cela oblige à lire peu de bits en entrée de la fonction. Si on veut donc pouvoir allier sécurité et efficacité il va donc être nécessaire de prendre un n très grand.

b) Application au schéma utilisant un codage en mots réguliers

Dans le cas du codage en mots réguliers on peut toujours très facilement évaluer le nombre moyen de solutions à une instance du problème. En revanche, la probabilité qu'un ensemble d'information soit valide pour l'une de ces solutions est plus difficile à évaluer. En effet, l'attaquant est libre de choisir les ensembles d'information qu'il veut essayer dans l'ordre qu'il veut, et même si dans ce schéma toutes les positions ont la même probabilité de faire partie d'une

solution, tous les ensembles d'information n'ont pas la même chance d'être valides.

On sait qu'une solution a exactement une position dans chaque bloc de $\frac{n}{w}$ positions. Si l'attaquant choisit donc k_i positions dans le $i^{\text{ème}}$ bloc la probabilité que son ensemble d'information soit valide est :

$$\mathcal{P} = \prod_{i=1}^{w} \frac{\frac{n}{w} - k_i}{\frac{n}{w}} \quad \text{avec} \quad \sum_{i=1}^{w} k_i = k = n - r.$$

Pour maximiser cette probabilité, il suffit de choisir $k_i = \frac{k}{w}$ pour tout i. L'attaquant a donc intérêt à ne choisir que des ensembles d'information « réguliers » qui ont tous la probabilité maximale d'être valides. Dans ce cas on peut évaluer la probabilité de choisir un bon ensemble d'information :

$$\mathcal{P}_{w,1} = \left(\frac{\frac{n}{w} - \frac{k}{w}}{\frac{n}{w}} \right)^w = \frac{\binom{r}{w}^w}{\binom{n}{w}^w}.$$

Le nombre de solutions est en moyenne :

$$\mathcal{N}_w = \frac{\binom{n}{w}^w}{2^r},$$

on trouve donc pour une tentative d'inversion de la fonction de compression :

$$\mathcal{P}_{\text{inv}} = \mathcal{P}_{w,1} \times \mathcal{N}_w = \frac{\binom{r}{w}^w}{2^r}.$$

Encore une fois on constate que la sécurité ne dépend pas de n. De plus, cette probabilité est beaucoup plus petite que celle obtenue pour la fonction à codage bijectif puisqu'il y a, à peu près, un facteur $w!/w^w$ entre les deux. L'utilisation d'un codage non bijectif semble donc avoir renforcé le système contre les attaques d'inversion.

Pour la recherche de collisions le problème est un peu le même. Le nombre moyen de solutions reste toujours facile à calculer et on évalue le nombre de solutions de poids $2i$ à en moyenne :

$$\mathcal{N}_i = \frac{\binom{w}{i}\binom{n/w}{2}^i}{2^r}.$$

En revanche, l'attaquant a maintenant plusieurs stratégies possibles : soit il peut, comme pour l'inversion, tirer des ensembles réguliers pour maximiser le nombre de solutions pour lesquelles il peut être valide ; soit il peut aussi s'attaquer à moins de solutions mais avec une meilleure probabilité de succès.

Avec la première stratégie l'attaquant choisit encore k/w positions par blocs pour son ensemble d'information. La probabilité de succès pour une solution de poids $2i$ est alors :

$$\mathcal{P}_{i,1} = \frac{\binom{n/w-2}{k/w}^i \binom{n/w}{k/w}^{w-i}}{\binom{n/w}{k/w}^w} = \frac{\binom{n/w-k/w}{2}^i}{\binom{n/w}{2}^i} = \frac{\binom{r/w}{2}^i}{\binom{n/w}{2}^i}.$$

Avec cette stratégie la probabilité de trouver une collision est donc de :

$$\mathcal{P}_{\text{col totale}} = \sum_{i=1}^{w} \frac{\binom{w}{i}\binom{r/w}{2}^i}{2^r} = \frac{1}{2^r}\left(\left[\binom{\frac{r}{w}}{2}+1\right]^w - 1\right) \simeq \frac{1}{2^r}\left[\binom{\frac{r}{w}}{2}+1\right]^w.$$

La deuxième stratégie consiste à dire que si l'attaquant ne cherche que des solutions n'ayant, par exemple, aucune position dans un certain bloc, alors il peut avoir une meilleure probabilité de trouver chacune de ces solutions et, par la même occasion, une probabilité de succès totale plus grande. Pour cela nous allons évaluer le coût d'une attaque pour un attaquant utilisant la stratégie suivante :

- choisir w_0 blocs parmi les w disponibles,
- prendre systématiquement toutes les positions des $w - w_0$ blocs restants,
- choisir les k_0 positions restantes de manière équitable dans les w_0 blocs.

Ainsi l'attaque ne porte que sur une sous instance du problème, avec des paramètres qui peuvent être plus intéressants. La probabilité de tirer un ensemble d'information valide s'exprime alors comme précédemment :

$$\mathcal{P}_{\text{col } w_0} = \frac{1}{2^r}\left[\binom{\frac{n}{w} - \frac{k_0}{w_0}}{2}+1\right]^{w_0} \quad \text{avec} \quad k_0 = k - (w-w_0)\times\frac{n}{w} = \frac{nw_0}{w} - r.$$

On peut alors simplifier l'expression en :

$$\mathcal{P}_{\text{col } w_0} = \frac{1}{2^r}\left[\binom{\frac{r}{w_0}}{2}+1\right]^{w_0}.$$

Étonnamment, cette expression ne dépend plus ni de n, ni de w et comme le w_0 peut être choisi librement par l'adversaire il pourra lui donner la valeur qui donne la meilleure probabilité en fonction de r. La seule contrainte est de garder $w_0 \leq w$, ce qui fait que l'adversaire ne pourra peut-être pas toujours opter pour le w_0 correspondant au maximum absolu. Le mieux qu'il pourra faire sera toujours :

$$\mathcal{P}_{\text{col optimal}} = \frac{1}{2^r}\max_{w_0\in[1;w]}\left[\binom{\frac{r}{w_0}}{2}+1\right]^{w_0}.$$

Si on remplace w_0 par αr dans l'expression $\left[\binom{\frac{r}{w_0}}{2}+1\right]^{w_0}$ et que l'on dérive, on obtient une expression qui ne dépend plus de r et qui a pour seule racine positive $\alpha \approx 0.24231\ldots$ Le maximum $\mathcal{P}_{\text{col optimal}}$ est donc toujours atteint pour $w_0 \simeq 0.24r$. Si $w > 0.24r$ on aura alors :

$$\mathcal{P}_{\text{col optimal}} = \frac{1}{2^r}\left[\binom{\frac{1}{\alpha}}{2}+1\right]^{\alpha r} \simeq 0.81^r \simeq 2^{-\frac{r}{3.3}}$$

La figure 8-5 page 116 nous permet de bien observer ce phénomène de seuil : on a $r = 256$ et donc pour $w > 61$ on a bien atteint un palier et le coût d'une attaque qui devrait normalement réaugmenter reste égal à son minimum. On remarque aussi que le coût d'une attaque sur la version utilisant des mots réguliers est effectivement bien plus grand que sur la version à codage bijectif. Dans tous les cas on peut borner le coût minimal d'une attaque par ensembles d'information sur un tour de la fonction de compression par $2^{\frac{r}{3.3}}$ opérations.

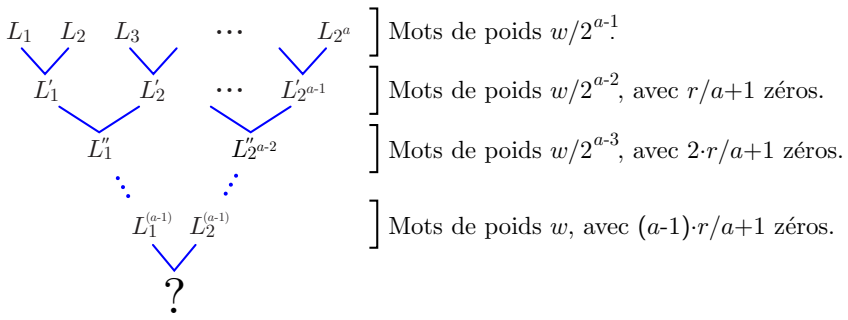


FIG. 8-6 – *Application du paradoxe des anniversaires généralisé de Wagner à la recherche de collisions. Les listes sont toutes de taille* $\ell = 2^{\frac{r}{a+1}}$. *Il reste en moyenne une seule solution à la fin.*

Il suffit alors de choisir un r assez grand pour avoir la sécurité désirée et de fixer les paramètres n et w de façon à avoir la meilleure efficacité possible. Malheureusement les choses ne sont pas si simples que ça : si l'attaque par ensembles d'information est la meilleure pour résoudre le problème SD, dans notre version réduite une autre attaque donne aussi de bons résultats.

8-5.2 Le paradoxe des anniversaires généralisé de Wagner

Soit le problème suivant : on se donne deux ensembles aléatoires L_1 et L_2 de taille ℓ constitués d'éléments codés sur r bits, et on cherche à trouver une paire d'éléments $x_1 \in L_1$ et $x_2 \in L_2$ tels que $x_1 \oplus x_2 = 0$. Dans ce cas le paradoxe des anniversaires permet de dire que si $\ell = 2^{\frac{r}{2}}$ alors il y aura en moyenne une solution au problème. Ce paradoxe peut s'utiliser pour rechercher des collisions dans une fonction de hachage et donne une attaque dont le coût est $\mathcal{O}\left(2^{\frac{r}{2}} \log_2 r\right)$.

Le paradoxe des anniversaires généralisé de Wagner [98] est une extension de la méthode précédente au cas où l'on a 2^a ensembles au lieu de 2. Soient L_1, $L_2 \ldots L_{2^a}$ des ensembles de ℓ éléments. Si on cherche $x_1 \oplus x_2 \oplus \ldots \oplus x_{2^a} = 0$, alors si on a $\ell = 2^{\frac{r}{a+1}}$ on peut trouver en moyenne une solution en temps $\mathcal{O}\left(2^a 2^{\frac{r}{a+1}} \log_2 r\right)$. Bien sûr $\ell = 2^{\frac{r}{2^a}}$ suffit pour avoir en moyenne une solution, mais dans ce cas le coût pour la trouver est toujours de $\mathcal{O}\left(2^{\frac{r}{2}} \log_2 r\right)$. Cette généralisation permet donc de savoir que si on a plus d'éléments qu'il ne faut, le coût de la recherche peut être grandement diminué.

Pour obtenir cette complexité la recherche fonctionne comme sur la figure 8-6. À la première étape on a 2^a listes de taille $\ell = 2^{\frac{r}{a+1}}$ que l'on va transformer paire par paire en 2^{a-1} listes toujours de taille ℓ (en moyenne) mais qui ne contiennent que des élément formés par la somme de deux éléments des premières listes et ayant des 0 sur les $\frac{r}{a+1}$ premiers bits. À chacune des étapes suivantes on refait la même opération jusqu'à n'avoir plus que 2 listes de mots n'ayant que $\frac{2r}{a+1}$ bits non nuls. Alors une application du paradoxe des anniversaires classique permet de trouver en moyenne une collision dans ces listes. La complexité de la fusion de deux listes étant de $\mathcal{O}\left(2^{\frac{r}{a+1}} \log_2 r\right)$ et l'algorithme nécessitant la fusion de 2^{a+1} listes, le coût total est comme annoncé de $\mathcal{O}\left(2^a 2^{\frac{r}{a+1}} \log_2 r\right)$.

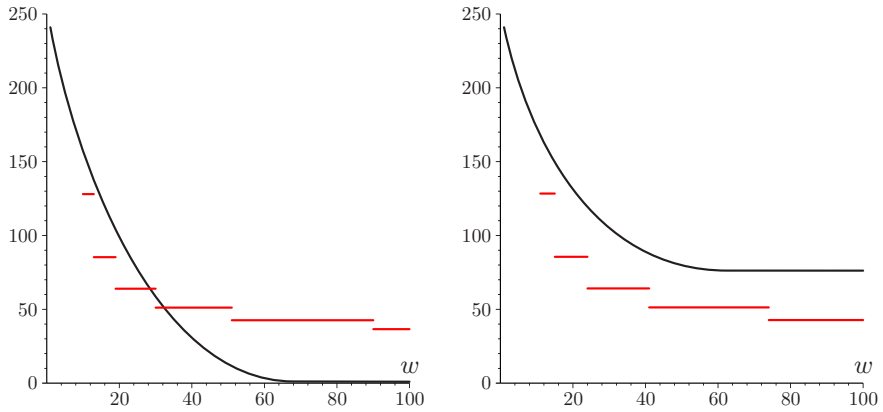


FIG. 8-7 – *Comparaison du coût de l'attaque par paradoxe des anniversaires de Wagner par rapport à la recherche d'ensembles d'information pour la recherche de collisions sur la fonction de compression. À gauche sur la version à codage bijectif, à droite sur la version à codage en mots réguliers. Les paramètres sont toujours* $r = 256$ *et* $n = 2^{16}$.

Si on veut appliquer cette méthode à la recherche de collisions dans notre fonction de compression (avec codage bijectif ou non), on va chercher des mots de poids $2w$ ayant un syndrome nul. Pour utiliser le paradoxe des anniversaires classique il faudrait construire deux listes de $2^{\frac{r}{2}}$ éléments de poids w. Ici on construit 2^a listes de $2^{\frac{r}{a+1}}$ éléments de poids $\frac{w}{2^{a-1}}$. L'attaquant va donc choisir le plus grand a possible pour avoir la complexité la plus faible. La seule contrainte est la taille des listes : il faut être capable de construire des listes ayant assez d'éléments.

Pour la fonction de compression utilisant le codage bijectif il faut pouvoir construire $2^a 2^{\frac{r}{a+1}}$ mots de poids $\frac{w}{2^{a-1}}$. Il faut donc :

$$a + \frac{r}{a+1} \leq \log_2 \binom{n}{\frac{w}{2^{a-1}}}$$

Le terme de droite décroissant beaucoup plus vite que celui de gauche, on ne pourra jamais prendre un a très grand. Comme il est possible de le voir sur la figure 8-7 cela ne permet que rarement d'être plus efficace que l'attaque par ensemble d'information.

Dans le cas du codage en mots réguliers chaque liste va correspondre à un groupe de $\frac{w}{2^a}$ blocs. Puisque l'on cherche des mots 2-réguliers, dans un bloc il est possible de construire $\binom{\frac{n}{w}}{2} + 1$ éléments différents (élément nul compris) et donc on peut construire 2^a listes de taille jusqu'à $\left(\binom{\frac{n}{w}}{2} + 1\right)^{\frac{w}{2^a}}$. La contrainte sur a est donc de la forme :

$$\frac{r}{a+1} \leq \frac{w}{2^a} \log_2 \left[\binom{\frac{n}{w}}{2} + 1 \right],$$

soit

$$\frac{2^a}{a+1} \leq \frac{w}{r} \log_2 \left[\binom{\frac{n}{w}}{2} + 1 \right].$$

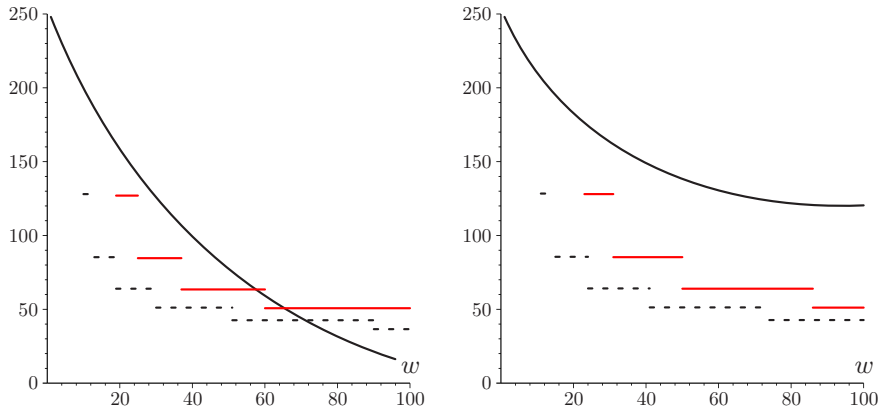


FIG. 8-8 – *Comparaison des deux attaques pour l'inversion de la fonction de compression. À gauche le codage bijectif, à droite les mots réguliers. Pour comparaison le coût de la recherche de collisions est dessiné en pointillés. Les paramètres sont toujours* $r = 256$ *et* $n = 2^{16}$.

On voit, cette fois-ci, sur la figure 8-7 page précédente que cette attaque est toujours meilleure que l'attaque par ensembles d'information. Il va donc falloir fixer les paramètres du système en fonction de cette attaque en priorité. Cela veut aussi dire que l'on ne pourra pas avoir une sécurité de $2^{\frac{r}{3.3}}$, sauf en prenant w très petit.

Pour l'inversion on peut aussi évaluer la meilleure valeur de a possible, mais le poids étant plus petit, le nombre de mots disponibles est plus faible et la taille des listes plus petite. On ne pourra donc pas en général prendre un a aussi grand que pour les collisions. C'est ce que l'on voit sur la figure 8-8.

Pour la version à codage bijectif l'inégalité à vérifier est :

$$a + \frac{r}{a+1} \leq \log_2 \binom{n}{\frac{w}{2^a}}.$$

Pour la version avec des mots réguliers c'est :

$$\frac{2^a}{a+1} \leq \frac{w}{r} \log_2 \left(\frac{n}{w}\right).$$

8-5.3 *Attaque de Wagner extrapolée*

L'attaque de Wagner comme décrite précédemment donne des coûts d'attaque en escalier. Cela n'a pas de sens, car même s'il y a un phénomène de seuil, au-delà de ce seuil le coût de l'attaque augmente exponentiellement, mais doit rester continu.

L'attaque de Wagner a un coût égal exactement à $\frac{r}{a+1}$ quand les inégalités décrites précédemment sont des égalités. Si on a effectivement une inégalité on doit pouvoir y gagner un peu. De même, si l'inégalité n'est pas vérifiée il restera moins d'une solution en moyenne à la fin, mais la probabilité qu'une solution subsiste n'est pas nulle.

Il faut donc choisir un modèle d'attaque correspondant au cas où l'inégalité n'est pas vérifiée et un autre pour le cas ou elle est vérifiée strictement. On

pourra ainsi évaluer le coût d'une attaque pour un a fixé sur tout l'ensemble des paramètres possibles. La courbe de sécurité que l'on cherche sera le minimum de toutes les courbes obtenues pour $a \geq 2$ et sera de ce fait continue.

a) Cas où l'inégalité n'est pas vérifiée

Dans ce cas, au lieu d'avoir des listes de $2^{\frac{r}{a+1}}$ éléments, elles n'en contiennent que 2^ℓ avec $\ell < \frac{r}{a+1}$. On va donc pouvoir annuler seulement ℓ bits du syndrome à chaque niveau de la méthode et les deux listes finales auront toujours 2^ℓ éléments, mais seulement $(a-1)\ell$ bits annulés. La probabilité de trouver une collision est donc de $\frac{2^{(a+1)\ell}}{2^r}$. Si on n'a pas de collision il suffit de recommencer la construction en annulant les bits dans un ordre différent, ou en formant des groupes différemment. Le coût de construction des deux listes finales étant de $\mathcal{O}\left(2^a 2^\ell \log_2 \ell\right)$ le coût total de l'attaque est $\mathcal{O}\left(2^{r+a-a\ell} \log_2 \ell\right)$. La taille ℓ est la taille maximale des listes que l'on peut construire, soit, dans le cas de collisions, pour la construction utilisant des mots réguliers $\ell = \frac{w}{2^a} \log_2 \left[\binom{n}{\frac{w}{2}} + 1\right]$.

b) Cas où l'inégalité est vérifiée strictement

Dans ce cas, on supposera aussi que l'on a des listes de taille 2^ℓ, mais maintenant $\ell > \frac{r}{a+1}$. Si on reprend la méthode précédente on peut donc, en conservant des listes de la même taille à chaque étape de l'algorithme, annuler ℓ bits à chaque fois pour se retrouver, à la fin, avec plus de mots et donc en moyenne plus d'une solution. Cependant traiter des listes plus grosses va augmenter le coût de l'attaque et le fait d'avoir plusieurs solutions valables ne fera rien gagner.

On peut en revanche travailler dans l'autre sens : on commence à annuler des bits à l'intérieur des listes avant la première étape. Pour chaque bit annulé la taille de la liste est divisée par deux : si on en annule α on se retrouve donc avec des listes de taille $2^{\ell-\alpha}$ pour annuler les $r' = r - \alpha$ bits restants. L'idéal est donc de choisir α tel que $\ell - \alpha = \frac{r'}{a+1}$ soit :

$$\alpha = \frac{\ell(a+1) - r}{a} \quad \text{et} \quad r' = \frac{a+1}{a}(r - \ell).$$

Le coût total de l'attaque est alors de $\mathcal{O}(2^a 2^{\frac{r-\ell}{a}} \log_2(\frac{r-\ell}{a}))$, sauf si la première étape d'annulation des α bits devient plus coûteuse que cela. Cette première étape ayant un coût de $\mathcal{O}(2^{\frac{\ell}{2}})$ (on suppose que l'on n'utilise pas la méthode de Wagner pour cette étape-ci) elle ne devient en gros plus coûteuse que si les listes ont le carré de la taille nécessaire, ce qui correspond exactement au moment où on peut commencer à utiliser un a plus grand. Les courbes de la figure 8-9 page suivante montrent le coût d'une attaque pour un a donné quand on fait varier les paramètres, et la façon dont ces courbes plus ou moins affines par morceaux se joignent entre elles pour donner le coût d'une attaque optimale sur le système.

Notons que la technique utilisée dans le cas ou l'inégalité est vérifiée strictement semble pouvoir aussi s'appliquer dans le cas où elle n'est pas vérifiée du tout : on peut ajouter des bits au début pour allonger le syndrome et faire grossir les listes jusqu'à atteindre une taille suffisante. On obtiendrait alors la même formule de complexité que pour le cas où l'inégalité est stricte. Cela améliorerait donc grandement l'attaque, cependant cette méthode ne marche pas. En augmentant de cette façon la taille des listes, les listes L'_i issues de la première fusion contiendront chaque mot en 2^α exemplaires. Leur taille réelle ne sera donc que

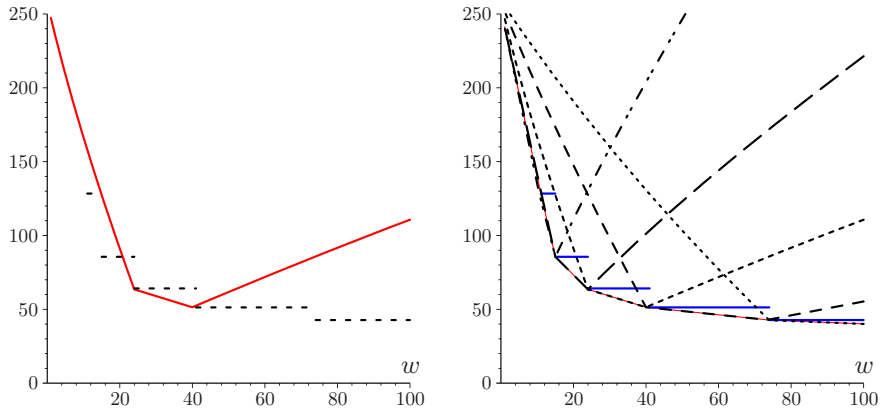


FIG. 8-9 – *Coût d'une recherche de collisions pour la fonction de compression utilisant le codage en mots réguliers avec l'attaque de Wagner extrapolée. À gauche pour* $a = 3$*, à droite pour* $a \in [1; 5]$*. La courbe en escalier est le coût de l'attaque dans sa version de base pour les mêmes paramètres* $r = 256$ *et* $n = 2^{16}$*.*

de 2^ℓ mots et l'on se retrouvera exactement comme après la première étape sans avoir augmenté le nombre de mots.

8-6 Choix des paramètres de la fonction de compression

C'est l'étape la plus importante de la conception de la fonction de compression. Il faut trouver des paramètres qui soient à la fois suffisamment sûrs et qui donnent en même temps une fonction la plus rapide possible. Pour cela nous avons la possibilité de fixer r, n et w. La taille de sortie r va être le paramètre prépondérant pour la sécurité. Si on ne regarde que les attaques par ensembles d'information c'est même le seul paramètre qui intervient puisque la sécurité est de $2^{\frac{r}{3.3}}$. Pour l'attaque de Wagner n et w vont aussi intervenir puisqu'ils vont définir la taille maximale des listes et donc la valeur de a.

Dans le cas de fonctions de hachage classiques, la sécurité attendue est en $2^{\frac{r}{2}}$ et toute fonction ayant une sécurité plus faible sera considérée comme moins bonne. Dans notre cas on ne pourra jamais atteindre une telle sécurité puisque l'attaque par ensembles d'information donne déjà de meilleurs résultats que cela. Les paramètres que l'on cherche seront donc ceux qui donneront une sécurité de 2^{80} (borne standard utilisée en cryptographie à clef publique) et la meilleure vitesse de hachage possible.

8-6.1 Efficacité de la fonction de compression

L'efficacité de la fonction de compression est ce qui va déterminer la vitesse du hachage. Les seuls calculs que nous ayons à effectuer pendant le hachage étant des XORs de colonnes de la matrice, la meilleure mesure d'efficacité sera le nombre de XORs binaires à effectuer par bit lu dans le document (bits d'entrée moins les bits de chaînage). À chaque tour de la fonction de compression on va devoir effectuer w XORs de colonnes de r bits. Le nombre de XORs par bit

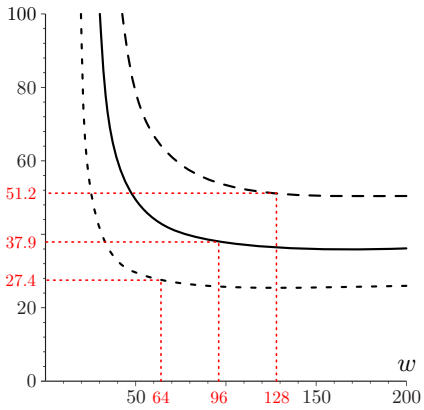


FIG. 8-10 – *Nombre moyen de XORs par bit d'entrée de la fonction de hachage en fonction de w pour les paramètres : $r = 192$ $n = 2^{16}$ (pointillés), $r = 256$ $n = 2^{16}$ (trait continu) et $r = 256$ $n = 2^{14}$ (tirets).*

d'entrée sera donc :

$$\mathcal{N}_{XOR} = \frac{rw}{w \log_2 \frac{n}{w} - r}.$$

La figure 8-10 donne une idée de la forme des courbes d'efficacité. La très forte pente à gauche correspond à une tangente verticale au point où w devient trop petit pour lire des bits du document : c'est le moment ou la fonction de compression commence à prendre moins de bits en entrée qu'en sortie. En revanche, dès que l'on s'éloigne de cette borne on arrive vers une sorte de palier où changer w n'affecte que peu la quantité de travail à effectuer (cela changera uniquement la taille des blocs).

Le point important est que l'on est donc assez libre pour le choix de w : à partir du moment où il n'est pas trop petit, sa valeur ne changera qu'assez peu la rapidité de la fonction de hachage. Reste donc à trouver des valeurs de n et r valables du point de vue de la sécurité.

8-6.2 Des paramètres pour une bonne sécurité

Pour la plupart des fonctions de hachage couramment utilisées, la meilleure attaque est l'attaque utilisant le paradoxe des anniversaires classique qui donne une sécurité en $2^{\frac{r}{2}}$. Dans notre cas, une telle sécurité correspond au cas où $a = 1$ et donc :

$$\frac{r}{2} = \frac{w}{2} \log_2 \left[\binom{\frac{n}{w}}{2} + 1 \right] \quad \Longleftrightarrow \quad r \simeq 2w \log_2 \left(\frac{n}{w} \right).$$

Le nombre de bits lus dans le document à chaque tour de fonction de compression est lui toujours de :

$$w \log_2 \left(\frac{n}{w} \right) - r.$$

On constate facilement que l'on ne peut donc pas se placer à une sécurité de $2^{\frac{r}{2}}$, sinon le nombre de bits lus devient négatif. Cela signifierait que notre fonction de compression a une sortie plus grande que son entrée. Si on veut donc

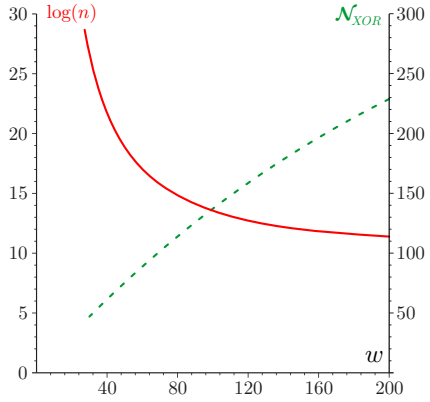


FIG. 8-11 – *Évolution de* $\log_2(n)$ *(courbe pleine) et* $\mathcal{N}_{XOR}$ *(courbe pointillée) en fonction de* w *quand on fixe* $r = 400$ *et* $a = 4$*, soit une sécurité de* 2^{80} *opérations.*

pouvoir réellement avoir une fonction de compression il va falloir permettre des attaques pour des valeurs de a plus grandes. Si on se place à $a = 3$ on aura cette fois :

$$r = \frac{w}{2}\log_2\left[\binom{\frac{n}{w}}{2} + 1\right] \simeq \frac{w}{2}\log_2\left[\left(\frac{n}{w}\right)^2 \times \frac{1}{2}\right] \simeq w\log_2\left(\frac{n}{w}\right) - \frac{w}{2}.$$

La fonction de compression aura donc bien une entrée plus grande que sa sortie, mais la différence est si petite que le nombre de XORs par bits d'entrée sera très grand :

$$\mathcal{N}_{XOR} = \frac{rw}{\frac{w}{2}} = 2r.$$

On vise un sécurité de 2^{80} opérations. Donc pour $a = 3$ cela donne $r = 320$ et donc $\mathcal{N}_{XOR} \approx 640$. Nous sommes donc très loin des valeurs que l'on peut lire sur la figure 8-10 page précédente. Si on veut pouvoir aller plus vite il va donc falloir accepter qu'un attaquant puisse utiliser des valeurs de a plus grandes.

Si on fixe maintenant $a = 4$ on trouve :

$$\mathcal{N}_{XOR} = \frac{rw}{\left(1 - \frac{5}{8}\right)\log_2\left(\frac{n}{w}\right) + \frac{5}{16}w}.$$

La principale différence par rapport au cas précédent est que si maintenant on veut faire diminuer le nombre de XORs, il suffit de faire diminuer w en faisant augmenter n (tout en vérifiant quand même l'équation liée à $a = 4$). La figure 8-11 montre comment vont évoluer ces valeurs en fonction de w. Il faut donc choisir un compromis entre la vitesse et la taille de la matrice utilisée (proportionnelle à n). Si on prend en compte le fait que, pour simplifier l'implémentation, $\frac{n}{w}$ doit être une puissance de 2 (pour qu'un nombre entier de bits code chaque position), il ne reste qu'assez peu de choix raisonnables. Le tableau 8-12 page suivante donne la liste des paramètres possibles pour une sécurité de 2^{80} (ou un peu plus pour avoir des paramètres entiers) avec $r = 400$.

En augmentant la valeur de r on pourra alors au choix : augmenter la sécurité, diminuer la taille de la matrice ou diminuer le nombre de XORs par bits

$\log_2\left(\frac{n}{w}\right)$	w	$\mathcal{N}_{XOR}$	taille de $\mathcal{H}$
16	41	64.0	~ 1 Gbit
15	44	67.7	550 Mbits
14	47	72.9	293 Mbits
13	51	77.6	159 Mbits
12	55	84.6	86 Mbits
11	60	92.3	47 Mbits
10	67	99.3	26 Mbits
9	75	109.1	15 Mbits
8	85	121.4	8.3 Mbits
7	98	137.1	4.8 Mbits
6	116	156.8	2.8 Mbits
5	142	183.2	1.7 Mbits
4	185	217.6	1.1 Mbits

TAB. 8-12 – *Valeurs possibles des paramètres pour $r = 400$ et $a \leq 4$ en fonction du choix de $\frac{n}{w}$. La sécurité correspondante est toujours de 2^{80} opérations ou un peu plus.*

d'entrée (ou n'importe quelle combinaison de ces trois possibilités). En revanche, si on veut augmenter la sécurité en gardant la même valeur de r, comme nous l'avons vu précédemment, on arrive très vite dans des domaines de paramètres impraticables ou totalement inefficaces.

8-6.3 Étude asymptotique

Nous avons vu que pour une sécurité donnée il était assez facile de trouver des paramètres satisfaisants pour la fonction de hachage. En revanche, rien ne dit que ce sera pareil si on vise une sécurité plus élevée. Pour faire l'étude asymptotique du système nous considérerons que la sécurité est exactement $2^{\frac{r}{a+1}}$. La seule condition que nous ayons sur a est :

$$\frac{2^a}{a+1} \leq \frac{w}{r} \log_2\left(\frac{n}{w}\right).$$

Si on choisit r comme seul paramètre du système, en supposant que w et n sont donc polynomiaux en r, on a :

$$\frac{2^a}{a+1} \leq \mathcal{P}oly\,(r) \log_2\left(\mathcal{P}oly\,(r)\right)$$

Le coefficient a est donc nécessairement de la forme $\mathcal{P}oly\,(\log_2(r))$. On peut donc en déduire que tant que n et w sont polynomiaux en r, la sécurité est bien exponentielle en r. Il devrait donc être facile de trouver des paramètres qui passent facilement à l'échelle quand on veut faire augmenter la sécurité.

a) La méthode « linéaire »

La solution la plus simple pour cela est de travailler avec des paramètres qui augmentent en ne permettant pas à l'adversaire d'augmenter la valeur de a. Si on choisit par exemple de garder w et n proportionnels à r, le membre droit de

l'inégalité que doit vérifier a va rester constant et donc, quel que soit r, a reste toujours borné par une constante.

Si on prend $w = \omega \times r$ et $n = \nu \times r$ on aura :

$$\frac{2^a}{a+1} \leq \omega \log_2 \frac{\nu}{\omega} \quad \text{donc} \quad a \simeq \log_2 \omega \log_2 \log_2 \frac{\nu}{\omega} = \kappa \quad \text{une constante.}$$

On aura donc asymptotiquement :
- Sécurité exponentielle : $\mathcal{O}\left(2^{\frac{r}{\kappa+1}}\right)$.
- Taille des blocs de matrice constants : $\log_2 \frac{n}{w} = \log_2 \frac{\nu}{\omega} = \mathcal{O}(1)$ (reste une puissance de 2 tout le temps).
- Coût du hachage linéaire : $\mathcal{N}_{XOR} = \frac{r^2\omega}{r(\omega \log_2 \frac{\nu}{\omega} - 1)} = \mathcal{O}(r)$.
- Taille de matrice quadratique : $r \times n = r^2\nu = \mathcal{O}\left(r^2\right)$.

On peut donc faire évoluer asymptotiquement n'importe quel jeu de paramètres valable pour une sécurité donnée. Par exemple pour les paramètres tirés de la ligne $\log_2 \frac{n}{w} = 8$ du tableau, correspondant au cas bien pratique ou on lit les données du document octet par octet, on a :

$$\omega = \frac{85}{400} = 0.2125 \quad \text{et} \quad \nu = 256\ \omega = 54.4$$

Si on veut une sécurité de 2^{160} il suffira de prendre $r = 800$ et donc $n = 43\,520$ et $w = 170$. De même, n'importe quel jeu de paramètres correspondant à une sécurité de $\frac{r}{4}$ ou $\frac{r}{6}$ pourra passer à l'échelle en gardant ce même niveau de sécurité tout le long, seules les constantes ω et ν seront changées.

b) En gardant aussi w constant

On peut, tout en gardant encore a constant, aussi garder w constant en ne faisant grandir que n. Il faut alors que $\frac{w}{r} \log_2 \left(\frac{n}{w}\right)$ reste quand même égal à une constante κ. La valeur de n va donc devoir grandir exponentiellement en r et cela nous fait sortir du contexte où tous les paramètres sont polynomiaux en r. Puisque a est borné tout devrait bien se passer quand même. On aura donc $n = w\, 2^{\kappa \frac{r}{w}}$, ce qui veut dire que la taille de la matrice augmentera aussi vite que la sécurité. Ce modèle asymptotique ne semble pas très praticable, sauf si on envisage d'utiliser la solution expliquée dans la section 8-7.

On aura alors :
- Sécurité exponentielle : $\mathcal{O}\left(2^{\frac{r}{\kappa+1}}\right)$.
- Taille des blocs de matrice linéaire (en nombre de bits) : $\log_2 \frac{n}{w} = \kappa \frac{r}{w} = \mathcal{O}(r)$ (reste une puissance de 2 quand on garde r multiple de $\frac{w}{\kappa}$).
- Coût du hachage constant : $\mathcal{N}_{XOR} = \frac{w}{\kappa - w} = \mathcal{O}(1)$.
- Taille de matrice exponentielle : $r \times n = rw\, 2^{\kappa \frac{r}{w}} = \mathcal{O}\left(r\, 2^{\frac{\kappa r}{w}}\right)$.

On garde donc ici un coût de hachage constant quelle que soit la sécurité demandée, mais cela nécessite de pouvoir se passer de la matrice...

8-7 Utilisation d'une matrice générée

Cette variante de la fonction de hachage a pour but de pouvoir se dispenser du stockage de la matrice de la fonction de hachage. L'idée est de générer cette matrice à partir d'un générateur pseudo-aléatoire, au fur et à mesure que l'on en a besoin. On pourrait grâce à cela utiliser des matrices de longueur beaucoup

plus longues. Comme nous l'avons vu, un grand n ne va pas affecter la sécurité mais va par contre donner des blocs plus longs et donc une même quantité de calculs pour traiter plus de données à la fois. Une telle construction ne sera donc pas adaptée au hachage de petits documents. En revanche, pour le hachage de grandes quantités de données, elle permet de diminuer le coût $\mathcal{N}_{XOR}$ par bit d'entrée aussi bas que souhaité.

En revanche, il faudra quand même ajouter au coût du hachage le coût de la génération des w colonnes. Cela implique, entre autres, d'utiliser un générateur qui peut efficacement générer directement la $i$ème colonne sans avoir à générer les $i-1$ précédentes.

8-7.1 Conditions sur le générateur

Le générateur doit, comme nous venons de le dire, pouvoir générer directement la $i$ème colonne à partir de son indice et sans calculer toutes les colonnes précédentes. Il sera donc défini par une fonction $g : [0; n] \longmapsto \mathbb{F}_2^r$ qui à un indice i associe la colonne $g(i)$. Il faut dans ce cas que la fonction g utilisée ait de bonnes propriétés afin de ne pas permettre de nouvelles attaques sur la fonction de hachage construite avec.

Il faut tout d'abord que g soit sans collisions pour ne pas permettre une construction plus simple de collisions sur la fonction de compression : il suffit, pour que cette contrainte soit simple à satisfaire, que l'espace d'arrivée soit plus grand que l'espace de départ et donc que $n \leq 2^r$. Cette contrainte n'est pas très forte mais va quand même borner $\mathcal{N}_{XOR}$ en fonction de r.

Il faut ensuite que l'inversion de la fonction de compression ne soit pas simplifiée et donc que, pour un $\mathcal{S}$ donné, chercher $(s_i)_{i\in[1;w]}$ tels que $\bigoplus_{i=1}^{w} g(s_i) = \mathcal{S}$ ne soit pas simple. Si on accepte que les s_i prennent n'importe quelles valeurs, cela implique que g soit calculatoirement non inversible. Si on leur implique de former un mot régulier, il suffit alors que l'inverse de g ait de bonnes propriétés de diffusion. Ce sera aussi le cas dès que n est suffisamment petit par rapport à 2^r. En effet, la probabilité qu'un inverse $g^{-1}(\mathcal{S})$ corresponde alors à un indice valable (plus petit que n) peut alors devenir négligeable.

Il suffit donc de choisir une fonction g injective, ayant de bonnes propriétés de diffusion et de fixer $n \ll 2^r$ pour ne pas affecter la sécurité du système. De plus, il faudra que g soit calculable rapidement afin de ne pas trop ralentir le hachage.

8-7.2 Utilisation du générateur explicite de Eichenauer-Herrmann et Niederreiter

Ce générateur [32, 71] qui utilise une fonction de la forme $g(i) = ai^{-1} + b$ sur un corps fini a de très bonnes propriétés de diffusion quand a et b sont bien choisis. Il ne devrait pas poser de problèmes de sécurité, surtout en utilisant des mots réguliers. On considère l'indice i de la colonne comme un élément du corps $\mathbb{F}_{2^r}$ et la colonne est obtenue en calculant $g(i) = ai^{-1} + b$ et en prenant l'écriture binaire de l'élément obtenu. Le coût de la génération d'une colonne sera donc juste une inversion, un produit et une somme dans le corps à 2^r éléments.

En utilisant une méthode standard l'inversion aura un coût de $r-2$ multiplications dans $\mathbb{F}_{2^r}$ ce qui sera de loin l'étape la plus coûteuse de la génération de

colonne. En utilisant une base normale et l'algorithme de Itoh-Tsujii [45, 50] on peut cependant ramener ce coût à $2\lfloor \log_2(r-1) \rfloor$. Le coût d'une multiplication dans $\mathbb{F}_{2^r}$ est $\mathcal{O}\left(r^2\right)$ avec une méthode standard et peut descendre, en utilisant une transformée de Fourier rapide, à $\mathcal{O}\left(r \log_2(r)^3\right)$ avec une forte constante (qui risque de rendre cette méthode moins avantageuse). Le coût total de la génération d'une colonne est donc de l'ordre de $\mathcal{O}\left(r^2 \log_2 r\right)$. Cela fait que le coût total d'un tour de la fonction de compression de la fonction de hachage devient $\mathcal{O}\left(wr^2 \log_2 r\right)$.

8-7.3 Avec plusieurs générateurs en parallèle

Pour améliorer la vitesse de génération on peut essayer de travailler sur un corps plus petit. Pour cela on va couper chaque colonne en μ tranches et travailler dans $\mathbb{F}_{2^{\frac{r}{\mu}}}$. On utilise alors le fait que la fonction g définie précédemment n'est que très peu corrélée avec d'autres fonctions choisies avec des a et b différents. Si on utilise donc une famille de telles fonction $(g_j)_{1 \leq j \leq \mu}$ avec $g_j(i) = a_j i^{-1} + b_j$ et une fonction g_j différente pour chaque tranche de la matrice on aura encore un générateur de bonne qualité.

Avec cette modification le coût de génération d'une colonne devient alors $\mathcal{O}\left(\frac{r^2}{\mu} \log_2 \frac{r}{\mu}\right)$ et a donc en gros été divisé par μ. Cependant on ne peut pas choisir un μ quelconque : le nombre de colonnes distinctes que peut générer ce nouveau générateur est $2^{\frac{r}{\mu}}$ et il faut donc maintenant que ce soit cette valeur qui soit très grande devant n pour pouvoir profiter des bonnes propriétés de diffusion et ne pas souffrir de l'utilisation d'un générateur pseudo-aléatoire. Cela va donc borner μ en fonction de n et il ne sera jamais très grand pour les valeurs intéressantes de n.

8-7.4 Discussion

L'utilisation d'un générateur pseudo-aléatoire pour recalculer les colonnes à chaque fois que nécessaire permet donc de se dispenser du stockage de la matrice et donc d'utiliser des longueurs n qui ne seraient autrement pas raisonnables. Cela va donc permettre de diminuer le nombre de tours de hachage, cependant le coût de la génération d'une colonne (environ 1000 fois supérieur au coût du hachage pour le générateur de Eichenauer-Herrmann et Niederreiter) avec un tel algorithme risque en général d'être trop élevé pour que cette méthode soit réellement avantageuse. Il est donc nécessaire de trouver un algorithme beaucoup plus efficace avant d'utiliser une méthode de ce genre.

8-8 Conclusion

Nous avons vu dans ce chapitre la construction d'une famille de fonctions de hachage, construites sur le schéma standard de Damgård et Merkle [26, 67], dont la sécurité repose sur un problème NP-complet. En étudiant les meilleures attaques sur cette construction il a été possible de déterminer des paramètres offrant une sécurité suffisante et conduisant en même temps à un débit de hachage suffisamment élevé pour concurrencer les meilleures fonctions actuellement utilisées. Ces fonctions présentent aussi l'avantage d'être facilement réglables, aussi

bien en ce qui concerne la taille de sortie (avec malheureusement quand même une taille minimale autour de 400 bits) que la taille de blocs.

De plus, pour un même jeu de paramètres, on peut construire une infinité de fonctions de hachage indépendantes les unes des autres. Ainsi trouver des collisions sur une fonction ne permet aucunement d'en trouver sur une autre. On peut ainsi simplement changer de matrice si des collisions apparaissent pour une matrice donnée. L'inconvénient est qu'en revanche, même si on sait qu'en général il sera difficile de trouver des collisions pour une fonction de cette famille, rien ne garantit que quand on en a choisi une elle est dans ce cas-là : le seul moyen de vérifier l'absence de collisions est d'en chercher, comme le ferait un attaquant. Cependant, il faut garder à l'esprit que la probabilité de tomber sur une fonction ayant des collisions faciles à trouver est certainement négligeable.

Ajoutons enfin que si l'on n'a pas besoin d'une bonne sécurité contre les collisions on peut simplement construire la fonction pour résister aux tentatives d'inversion et ainsi obtenir des fonctions à sens unique un peu plus rapides qu'une fonction de hachage ayant la même taille de sortie. Cela permet aussi de construire des fonctions à sens unique ayant une sortie un peu plus courte (320 bits).

Conclusions et perspectives

OUS avons exploré dans cette thèses quelques nouvelles applications de la théorie des codes correcteurs d'erreurs en cryptographie. Ces applications sont bien concrètes, mais elles nécessitent certainement encore quelques petits ajustements avant d'être réellement utilisables en pratique. En effet, la sécurité d'un système s'évalue grâce au coût des meilleures attaques, mais l'effort fourni pour essayer d'en trouver de meilleures est aussi une part importante de cette sécurité. De ce point de vue là, un système ancien et bien connu de tous est souvent préférable à un système plus jeune, même s'il offre de bonnes propriétés.

Dans ce dernier chapitre, je tâcherai donc de présenter un bref inventaire des travaux restant à effectuer pour chacun des quatre systèmes présentés.

Le schéma de signature McEliece

C'est certainement la partie de cette thèse qui, du point de vue de la sécurité, est la mieux achevée. En effet, la sécurité de ce schéma repose directement sur les mêmes problèmes que dans le système de McEliece. Or, ce système ayant été étudié en détail depuis plus de 25 ans, il peut être considéré comme suffisamment robuste. Il suffit donc de vérifier que les quelques modifications qui ont été apportées, dues au fait qu'on utilise le système dans un contexte de signature ou encore aux changements de paramètres, n'affaiblissent pas le système original.

Mis à part les problèmes de taille de clef ou autres, présents aussi dans le système de McEliece et ne semblant pas faciles à résoudre, les principaux inconvénients de cette construction sont les temps de signature et de vérification (dans le cas des signatures les plus courtes). Ces temps ne sont pas rédhibitoires, mais ils sont néanmoins trop grands pour la plupart des utilisations « classiques » des signatures. La meilleure façon que l'on peut voir d'améliorer cela est de faire appel à une implémentation matérielle de ce schéma. En effet, la vérification comme la signature sont la répétition d'un même test un grand nombre de fois, et sur des entrées indépendantes à chaque fois. Elles semblent donc toutes deux bien adaptées à des implantations en parallèle, utilisant aussi les avantages du pipe-line par exemple.

Une étude des performances possibles avec une telle implantation est en cours dans le cadre de l'ACI OCAM. Pour l'instant, les premiers essais sur des architectures reprogrammables (FPGA) semblent donner des résultats intéressants avec des premières estimations du temps de signature à moins d'une seconde,

soit un gain d'un facteur 100 par rapport à une implantation logicielle. Il doit cependant être possible de faire encore beaucoup mieux et, en faisant appel à un coprocesseur spécialisé, ce schéma de signature devrait pouvoir s'adapter à de nouvelles applications.

Mots de poids minimum d'un Goppa

Dans cette partie nous avons vu comment évaluer le nombre de mots d'un poids donné dans un code de Goppa corrigeant un petit nombre d'erreurs. Ces travaux n'ont pas en soi d'intérêt cryptographique, mis à part en confortant l'idée que les codes de Goppa ressemblent beaucoup à des codes aléatoires. En revanche, l'algorithme utilisé pour trouver des mots de poids minimum est certainement réutilisable dans un contexte de cryptographie à clef publique.

En effet, seule la connaissance de la structure du code de Goppa permet d'avoir un algorithme plus efficace que la recherche exhaustive pour exhiber des mots de poids minimum. Ainsi, on peut imaginer s'identifier auprès de quelqu'un en lui prouvant que l'on est capable de fournir un mot de poids minimum d'un code de Goppa. On a donc à notre disposition un nouvel outil, présentant toutefois les même inconvénients de coût que le schéma de signature, pour des applications cryptographiques. Celles-ci restent encore à trouver.

Le cryptosystème Augot-Finiasz

Ce schéma était le premier à utiliser pour sa sécurité la difficulté de certaines instances du problème de *Polynomial Reconstruction*. Il a été cassé par une attaque qui arrivait à contourner cette difficulté en s'appuyant sur une faille de conception du système. Même s'il semble difficile à l'heure actuelle de réparer ce système sans y apporter de changements majeurs, les idées sur lesquelles il reposait ne semblent pas toutes mauvaises.

L'utilisation du problème de *Polynomial Reconstruction* en cryptographie a déjà été l'objet de nombreuses recherches peu fructueuses et même s'il présente d'intéressantes propriétés en matière de sécurité il semble difficile d'y cacher une trappe pour une utilisation en clef publique. En revanche, l'idée de cacher une instance facile à l'intérieur d'une instance difficile, comme cela est fait dans ce cryptosystème, est certainement réutilisable. Il suffit pour cela d'avoir un problème possédant un effet de seuil : un problème pour lequel certaines instances sont faciles et d'autres, relativement voisines, sont très difficiles. Cette voie reste donc à explorer, même si l'expérience infructueuse de ce système peut sembler décourageante.

La fonction de hachage

C'est le travail pour lequel l'étude de la sécurité semble la moins achevée. En effet, la sécurité du système repose directement, et de façon relativement claire, sur un problème bien précis, voisin du célèbre problème de *syndrome decoding* (SD). Cependant, même si ce problème proche est lui aussi NP-complet, l'étude de la complexité des meilleures attaques en est encore à son début. Nous avons

vu par exemple que les meilleures attaques sur SD s'appliquent mal à cet autre problème, et une simple attaque utilisant le paradoxe des anniversaires généralisé de Wagner arrive à être plus efficace. Il parait donc fort probable que d'autres attaques, plus spécifiques soient elles aussi possibles.

L'idée d'utilisation d'une matrice générée sur demande est aussi à explorer plus en détail : même si les premiers résultats ne semblent pas très probants, une analyse plus fine des contraintes à fixer sur le générateur utilisé pourrait permettre d'alléger le coût de la génération.

Enfin, un travail est en cours pour étudier la possibilité de faire intervenir une clef dans cette fonction pour en faire un MAC. On peut par exemple tirer profit du grand nombre de fonctions différentes que l'on peut construire dans cette famille et par exemple utiliser une matrice qui dépendrait de la clef choisie.

Autres perspectives

Bien sûr, rien ne sert d'essayer de refaire toute la cryptographie moderne avec des codes correcteurs d'erreurs. Cependant, en faisant cela on risque parfois de faire des découvertes intéressantes. Ça a par exemple été le cas pour la signature : en plus de construire un schéma solide utilisant des codes, on a pu obtenir les signatures les plus courtes connues à ce jour. Il reste donc un énorme travail d'investigation à effectuer pour essayer d'utiliser de nouveaux problèmes difficiles de théorie des codes, ou pour tenter d'adapter les problèmes bien connus à de nouvelles primitives : signatures basées sur l'identité, signatures de groupes, générateurs d'aléa, boîtes S...

De ce point de vue là, les codes ont l'avantage de présenter peu de contraintes, avec des paramètres faciles à faire varier, tout en regorgeant de problèmes difficiles, et même souvent difficiles pour une instance aléatoire moyenne. Les cryptologues n'auront donc certainement pas fini d'entendre parler de codes avant quelques générations...

Bibliographie

[1] C. Adams and H. Meijer. Security-related comments regarding McEliece's public-key cryptosystem. In C. Pomerance, editor, *Advances in Cryptology - CRYPTO'87*, volume 293 of *Lecture Notes in Computer Science*, pages 224–228. Springer-Verlag, 1987.

[2] C. M. Adams. *Examination and Analysis of McEliece's Public-Key Cryptosystem.* PhD thesis, Department of Computing and Information Science, Queen's University, Ontario, Canada, 1985.

[3] M. Alabbadi and S. B. Wicker. Cryptoanalysis of the Harn and Wang modification of the Xinmei digital signature scheme. *Electronic Letters*, 28(18):1756–1758, 1992.

[4] M. Alabbadi and S. B. Wicker. Digital signature scheme based on error-correcting codes. In *Proceedings of IEEE International Symposium on Information Theory*, page 199, San Antonio, USA, 1993.

[5] M. Alabbadi and S. B. Wicker. A digital signature scheme based on linear error-correcting block codes. In J. Pieprzyk and R. Safavi-Naini, editors, *Advances in Cryptology – ASIACRYPT'94*, volume 917 of *Lecture Notes in Computer Science*, pages 238–248. Springer-Verlag, 1995.

[6] D. Augot and M. Finiasz. A public key encryption scheme based on the polynomial reconstruction problem. In E. Biham, editor, *Eurocrypt 2003*, volume 2656 of *Lecture Notes in Computer Science*, pages 229–240, Warsaw, Poland, May 2003. Springer-Verlag.

[7] D. Augot, M. Finiasz, and N. Sendrier. A fast provably secure cryptographic hash function. Cryptology ePrint Archive, 2003. Available at: http://eprint.iacr.org/2003/230/.

[8] A. Barg. Complexity issues in coding theory. In V. S. Pless and W. C. Huffman, editors, *Handbook of Coding theory*, volume I, chapter 7, pages 649–754. North-Holland, 1998.

[9] M. Bellare and P. Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *CCS'93 – 1st Conference on Computer and Communications Security*, pages 62–73. ACM, 1993.

[10] E. R. Berlekamp. *Algebraic Coding Theory.* Aegen Park Press, 1968.

[11] E. R. Berlekamp. Goppa codes. *IEEE Transactions on Information Theory*, 19(5):590–592, September 1973.

[12] E. R. Berlekamp, R. J. McEliece, and H. C. van Tilborg. On the inherent intractability of certain coding problems. *IEEE Transactions on Information Theory*, 24(3), May 1978.

[13] E. R. Berlekamp and L. R. Welch. Error correction for algebraic block codes. US Patent 4 633 470, 1986.

[14] E. F. Brickell. Breaking iterated knapsacks. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 342–358. Springer-Verlag, 1985.

[15] K. A. Bush. Orthogonal arrays of index unity. *Annals of Mathematical Statistics*, 23:426–434, 1952.

[16] A. Canteaut. *Attaques de cryptosystèmes à mots de poids faible et construction de fonction t-résilientes*. Thèse de doctorat, Université Paris 6, October 1996.

[17] A. Canteaut and F. Chabaud. A new algorithm for finding minimum-weight words in a linear code: Application to McEliece's cryptosystem and to narrow-sense BCH codes of length 511. *IEEE Transactions on Information Theory*, 44(1):367–378, January 1998.

[18] F. Chabaud and A. Joux. Differential collisions in SHA-0. In H. Krawczyk, editor, *Advances in Cryptology – CRYPTO'98*, volume 1462 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, 1998.

[19] B. Chor and R. Rivest. A knapsack-type public key cryptosystem based on arithmetic in finite fields. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 54–65. Springer-Verlag, 1985.

[20] J.-S. Coron. Cryptanalysis of a public-key encryption scheme based on the polynomial reconstruction problem. Cryptology ePrint Archive, 2003. http://eprint.iacr.org/2003/036/.

[21] J.-S. Coron. Cryptanalysis of the repaired public-key encryption scheme based on the polynomial reconstruction problem. Cryptology ePrint Archive, 2003. http://eprint.iacr.org/2003/219/.

[22] N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a McEliece-based digital signature scheme. In C. Boyd, editor, *Asiacrypt 2001*, volume 2248 of *LNCS*, pages 157–174. Springer-Verlag, 2001.

[23] N. Courtois, M. Finiasz, and N. Sendrier. How to achieve a McEliece-based digital signature scheme. Rapport de recherche 4118, INRIA, February 2001.

[24] N. Courtois, M. Finiasz, and N. Sendrier. Short McEliece-based digital signatures. In *Proceedings 2002 IEEE International Symposium on Information Theory*, Lausanne, Suisse, July 2002.

[25] T. Cover. Enumerative source encoding. *IEEE Transactions on Information Theory*, 19(1):73–77, January 1973.

[26] I.B. Damgard. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology - Crypto' 89*, Lecture Notes in Computer Science, pages 416–426. Springer-Verlag, 1989.

[27] B. den Boer and A. Bosselaers. Collisions for the compression function of MD5. In T. Helleseth, editor, *Advances in Cryptology – EUROCRYPT'93*, volume 765 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag, 1994.

[28] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[29] H. Dobbertin. The status of MD5 after a recent attack. *Cryptobytes*, 2(2):1–6, 1996.

[30] H. Dobbertin. Cryptanalysis of MD4. *Journal of Cryptology*, 11(4):253–271, 1998.

[31] H. Dobbertin, A. Bosselaers, and B. Preneel. RIPEMD-160: a strengthened version of RIPEMD. In D. Gollmann, editor, *Advances in Cryptology – FSE 96*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer-Verlag, 1996.

[32] J. Eichenauer-Herrmann and H. Niederreiter. Digital inversive pseudorandom numbers. *ACM Transactions on Modeling and Computer Simulation*, 4(4):339–349, October 1994.

[33] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and D. Chaum, editors, *Advances in Cryptology – CRYPTO'84*, volume 196 of *Lecture Notes in Computer Science*, pages 10–18. Springer-Verlag, 1985.

[34] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[35] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer-Verlag, 1987.

[36] M. Finiasz. Words of minimal weight and weight distribution of binary Goppa codes. In *Proceedings of IEEE International Symposium on Information Theory*, Yokohama, Japan, July 2003. IEEE.

[37] J. K. Gibson. Discrete logarithm hash function that is collision free and one way. *Computers and Digital Techniques, IEE Proceedings E*, 138(6):407–410, November 1991.

[38] J. K. Gibson. Equivalent Goppa codes and trapdoors to McEliece's public key cryptosystem. In D. W. Davies, editor, *Advances in Cryptology - EUROCRYPT'91*, volume 547 of *Lecture Notes in Computer Science*, pages 517–521. Springer-Verlag, 1991.

[39] M. Girault. A (non-practical) three-pass identification protocol using coding theory. In J. Seberry and J. Pieprzyk, editors, *Advances in Cryptology - AUSCRYPT'90*, volume 453 of *Lecture Notes in Computer Science*, pages 265–272. Springer-Verlag, 1990.

[40] O. Goldreich, R. Rubinfeld, and M. Sudan. Learning polynomials with queries: The highly noisy case. In *36th Annual Symposium on Foundations of Computer Science*, pages 294–303, Milwaukee, Wisconsin, October 1995. IEEE.

[41] O. Goldreich, R. Rubinfeld, and M. Sudan. Learning polynomials with queries: The highly noisy case. *SIAM Journal on Discrete Mathematics*, 13(4):535–570, 2000.

[42] S. Golomb. Run-length encoding. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[43] V. D. Goppa. A new class of linear error-correcting codes. *Probl. Inform. Transm.*, 6:207–212, 1970.

[44] V. D. Goppa. Codes on algebraic curves. *Soviet Math. Dokl.*, 24:170–172, 1981.

[45] J. Guajardo and C. Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Designs, Codes and Cryptography*, 25(2):207–216, February 2002.

[46] Y. Gurevich. Average case completeness. *Journal of Computer and System Sciences*, 42(3):346–398, 1991.

[47] V. Guruswami and M. Sudan. Improved decoding of Reed-Solomon and Algebraic-Geometric codes. *IEEE Transactions on Information Theory*, 45(6):1757–1767, September 1999.

[48] V. Guruswami and A. Vardy. Maximum-likelihood decoding of Reed-Solomon codes is NP-hard. arXiv.org e-Print archive, May 2004. http://arxiv.org/abs/cs/0405005.

[49] L. Harn and D. C. Wang. Cryptoanalysis and modification of digital signature scheme based on error-correcting codes. *Electronic Letters*, 28(2):157–159, 1992.

[50] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $gf(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[51] A. Joux J.-C. Faugère. Algebraic cryptanalysis of hidden field equation (HFE) cryptosystems using gröbner bases. In D. Boneh, editor, *Advance in Cryptology – CRYPTO'03*, volume 2729 of *Lecture Notes in Computer Science*, pages 44–60. Springer-Verlag, 2003.

[52] T. Johansson and F. Jönsson. Fast correlation attacks based on turbo code techniques. In M. J. Wiener, editor, *Advances in Cryptology – CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 181–197. Springer-Verlag, 1999.

[53] T. Johansson and F. Jönsson. Improved fast correlation attacks on stream ciphers via convolutional codes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT'99*, volume 1592 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1999.

[54] D. Johnson, A. Menezes, and S. Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, August 2001.

[55] G. Kabatianskii, E. Krouk, and B. Smeets. A digital signature scheme based on random error-correcting codes. In *the 6th IMA International Conference*, pages 161–177, Cirencester, UK, December 1997.

[56] G. L. Katsman, M. A. Tsfasman, and S. G. Vlădut. Modular curves and codes with a polynomial construction. *IEEE Transactions on Information Theory*, 30(2):353–355, March 1984.

[57] A. Kiayias and M. Yung. Cryptanalyzing the Polynomial-Reconstruction based public-key system under optimal parameter choice. unpublished, 2004.

[58] P. J. Lee and E. F. Brickell. An observation on the security of McEliece's public-key cryptosystem. In C. G. Günther, editor, *Advances in Cryptology – EUROCRYPT'88*, volume 330 of *Lecture Notes in Computer Science*, pages 275–280. Springer-Verlag, 1988.

[59] J. S. Leon. A probabilistic algorithm for computing minimum weights of large error-correcting codes. *IEEE Transactions on Information Theory*, 34(5):1354–1359, September 1988.

[60] L. Levin. Average case complete problems. *SIAM Journal on Computing*, 15(1):285–286, 1986.

[61] F. Levy-dit-Vehel and S. Litsyn. On the covering radius of long Goppa codes. In G.Cohen, M.Giusti, and T.Mora, editors, *AAECC-11*, volume 948 of *Lecture Notes in Computer Science*, pages 341–346. Springer-Verlag, June 1995.

[62] F. Levy-dit-Vehel and S. Litsyn. Parameters of Goppa codes revisited. *IEEE Transactions on Information Theory*, 43(6):1811–1819, November 1997.

[63] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977.

[64] J. L. Massey. Shift-register synthesis and BCH decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, January 1969.

[65] J. L. Massey. SAFER K-64: A byte-oriented block-ciphering algorithm. In R. J. Anderson, editor, *Advances in Cryptology – FSE 93*, volume 809 of *Lecture Notes in Computer Science*, pages 1–17. Springer-Verlag, 1994.

[66] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. *DSN Prog. Rep.,* Jet Prop. Lab., California Inst. Technol., Pasadena, CA, pages 114–116, January 1978.

[67] R. C. Merkle. One way hash functions and DES. In Gilles Brassard, editor, *Advances in Cryptology - Crypto' 89*, Lecture Notes in Computer Science. Springer-Verlag, 1989.

[68] R. C. Merkle and M. E. Hellman. Hiding information and signatures in trap-door knapsacks. *IEEE Transactions on Information Theory*, 24:525–530, 1978.

[69] National Insitute of Standards and Technology. *FIPS Publication 180: Secure Hash Standard*, 1993.

[70] H. Niederreiter. Knapsack-type crytosystems and algebraic coding theory. *Prob. Contr. Inform. Theory*, 15(2):157–166, 1986.

[71] H. Niederreiter. Pseudorandom vector generation by the inversive method. *ACM Transactions on Modeling and Computer Simulation*, 4(2):191–212, April 1994.

[72] A. M. Odlyzko. Cryptanalytic attacks on the multiplicative knapsack cryptosystem and on Shamir's fast signature system. *IEEE Transactions on Information Theory*, 30:594–601, 1984.

[73] H. Ong, C. Schnorr, and A. Shamir. An efficient signature scheme based on quadratic forms. In *Proceedings of 16th ACM Symposium on Theoretical Computer Science*, pages 208–216, 1984.

[74] J. Patarin. Hidden fields equations (hfe) and isomorphisms of polynomials (ip): two new families of asymmetric algorithms. In *Eurocrypt'96*, Lecture Notes in Computer Science, pages 33–48, 1996.

[75] J. Patarin, L. Goubin, and N. Courtois. 128-bit long digital signatures. In *Cryptographers' Track Rsa Conference 2001*, San Francisco, April 2001. Springer-Verlag.

[76] N. J. Patterson. The algebraic decoding of Goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207, March 1975.

[77] E. Petrank and R. M. Roth. Is code equivalence easy to decide? *IEEE Transactions on Information Theory*, 43(5):1602–1604, September 1997.

[78] M. Rabin. Digital signatures and public-key functions as intractable as factorization. Technical Report MIT/LCS/TR-212, MIT Laboratory of Computer Science, January 1979.

[79] T. R. N. Rao. Cryptosystems using algebraic codes. In *International conference on Computer Systems & Signal Processing*, December 1984.

[80] T. R. N. Rao and K. Nam. Private-key algebraic-coded cryptosystems. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 35–48. Springer-Verlag, 1986.

[81] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the SIAM*, 8(2):300–304, June 1960.

[82] R. L. Rivest. The MD4 message digest algorithm. In A.J. Menezes and S.A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90*, Lecture Notes in Computer Science, pages 303–311. Springer-Verlag, 1991.

[83] R. L. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1321, April 1992. Internet Activities Board, Internet Privacy Task Force.

[84] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[85] N. Sendrier. Finding the permutation between equivalent codes: the support splitting algorithm. *IEEE Transactions on Information Theory*, 46(4):1193–1203, July 2000.

[86] N. Sendrier. *Cryptosystèmes à clé publique basés sur les codes correcteurs d'erreurs.* Mémoire d'habilitation à diriger des recherches, Université Paris 6, 2002. `http://www-rocq.inria.fr/codes/Nicolas.Sendrier/`.

[87] A. Shamir. A polynomial time algorithm for breaking the basic merkle-hellman cryptossytem. *IEEE Transactions on Information Theory*, 30:699–704, 1984.

[88] J. Stern. A method for finding codewords of small weight. In G. Cohen and J. Wolfmann, editors, *Coding theory and applications*, volume 388 of *Lecture Notes in Computer Science*, pages 106–113. Springer-Verlag, 1989.

[89] J. Stern. An alternative to the Fiat-Shamir protocol. In J.-J. Quisquater and J. Vandewalle, editors, *Advances in Cryptology – EUROCRYPT'89*, volume 434 of *Lecture Notes in Computer Science*, pages 173–180. Springer-Verlag, 1990.

[90] J. Stern. A new identification scheme based on syndrome decoding. In D. R. Stinson, editor, *Advances in Cryptology - CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 13–21. Springer-Verlag, 1993.

[91] J. Stern. Can one design a signature scheme based on error-correcting codes? In *Asiacrypt 1994*, volume 917 of *Lecture Notes in Computer Science*, pages 424–426. Springer-Verlag, 1994. Rump session.

[92] M. Sudan. Decoding of Reed-Solomon codes beyond the error-correction bound. *Journal of Complexity*, 13(1):180–193, March 1997.

[93] M. A. Tsfasman, S. G. Vlădut, and T. Zink. Modular curves, Shimura curves, and Goppa codes, better than the Varshamov-Gilbert bound. *Math. Nachrichten*, 109:21–28, 1982.

[94] J. van Tilburg. Cryptanalysis of Xinmei digital signature scheme. *Electronic Letters*, 28(20):1935–1936, 1992.

[95] J. van Tilburg. Cryptanalysis of the Alabbadi-Wicker digital signature scheme. In *Proceedings of 14th Symposium on Information Theory in the Benelux*, pages 114–119, Veldhoven, Netherlands, May 1993.

[96] A. Vardy. The intractability of computing the minimum distance of a code. *IEEE Transactions on Information Theory*, 43(6):1757–1766, November 1997.

[97] D. Vertigan. Bicycle dimension and special points of the Tutte polynomial. *Journal of Combinatorial Theory, Series B*, 74(2):378–396, November 1998.

[98] D. Wagner. A generalized birthday problem. In M. Yung, editor, *CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–304. Springer-Verlag, 2002.

[99] X. M. Wang. Digital signature scheme based on error-correcting codes. *Electronic Letters*, 26(13):898–899, June 1990.

[100] S. B. Xu and J. M. Doumen. An attack against the Alabbadi-Wicker scheme. In *Proceedings of 20th Symposium on Information Theory in the Benelux*, Haasrode, Belgium, May 1999.

[101] S. B. Xu, J. M. Doumen, and H. C. A. van Tilborg. On the security of digital signature schemes based on error-correcting codes. *Designs, Codes and Cryptography*, 28(2):187–199, March 2003.

[102] Y. Zheng, J. Pieprzyk, and J. Seberry. HAVAL - a one-way hashing algorithm with variable length of output. In J. Seberry and Y. Zheng, editors, *Advances in Cryptology – ASIACRYPT'92*, volume 718 of *Lecture Notes in Computer Science*, pages 83–104. Springer-Verlag, 1993.