THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut Supérieur de l'Aéronautique et de l'Espace

Présentée et soutenue par :
Jonathan DETCHART

le mercredi 5 décembre 2018

Titre :

Optimisation de codes correcteurs d'effacements par application de transformées polynomiales

---

École doctorale et discipline ou spécialité :
ED MITT : Réseaux, télécom, système et architecture

Unité de recherche :
Équipe d'accueil ISAE-ONERA MOIS

Directeur(s) de Thèse :

M. Jérôme LACAN (directeur de thèse)
M. Emmanuel LOCHIN (co-directeur de thèse)

Jury :

M. Daniel AUGOT Directeur de Recherche INRIA Saclay - Président, Rapporteur
M. Nicolas NORMAND Maître de Conférences IRRCyN Polytech'Nantes - Rapporteur
M. Vincent ROCA Chargé de Recherche INRIA Grenoble
M. Jérôme LACAN Professeur ISAE-SUPAERO - Directeur de thèse
M. Emmanuel LOCHIN Professeur ISAE-SUPAERO - Co-directeur de thèse

Remerciements

Mes premiers remerciements vont à mes directeurs de thèses, Jérôme Lacan et Emmanuel Lochin qui m'ont guidé et soutenu, toujours dans la bonne humeur, pendant ces trois années de thèse. Merci à eux pour leur patience, leur disponibilité et surtout leur confiance.

Je remercie également les rapporteurs de ma thèse, Daniel Augot et Nicolas Normand, pour leurs nombreuses remarques et suggestions qui m'ont permis d'améliorer la qualité de ce manuscrit.

Et bien entendu, je remercie Vincent Roca qui m'a permis de découvrir le monde des codes à effacements à l'INRIA avant cette thèse.

Un grand merci à toute l'équipe RESCOM du département DISC de l'ISAE-SUPAERO : Yann, Fabrice, Frédéric, Ahlem, Tanguy, Bastien, Doriane, Ludovic, et les nombreuses personnes du département DISC avec qui j'ai pu échanger : Eyal, Henrick, Clément, Ludovic et tous les autres. Les discussions quotidiennes avec vous, scientifiques ou non, ont rendu ces trois années innoubliables.

Je remercie également Odile, la secrétaire du département DISC, toujours présente pour résoudre les nombreux problèmes administratifs.

Je remercie tous mes proches pour leur écoute et leurs encouragements. Je remercie également mes parents qui m'ont toujours soutenu et encouragé à poursuivre dans ce qui me passionne depuis plusieurs années.

Enfin, je remercie ma Caroline d'être à mes cotés chaque jour et de m'avoir encouragé et soutenu jusqu'au bout.

Merci à vous tous !

Table des matières

Table des figures

Chapitre 1

Introduction

Sommaire

1.1 Contexte

En 1948, Claude Shannon a introduit ce qu'il a appelé la «théorie mathématique de l'information ». De cette dernière est née la théorie des codes. L'objectif de ces codes est de pouvoir corriger toute perturbation pouvant apparaitre sur un canal de transmission, au sens large du terme. De manière générale, les codes à effacement fonctionnent par ajout de redondance. Cette redondance est générée par l'émetteur en fonction des données à protéger. Elle est ensuite utilisée par le récepteur pour récupérer l'information perturbée.
On distingue plusieurs types de perturbations. D'abord, l'information peut être corrompue. Ces perturbations, appelées erreurs, entrainent une modification des bits d'information envoyés sur le canal. Une autre perturbation possible est la perte complète de l'information, résultant par exemple d'erreurs irrécupérables ou d'une panne matérielle et/ou logicelle.

De nombreux codes ont été développés, permettant de fiabiliser différents types de canaux de transmission. Parmi ces codes, on trouve la famille des codes correcteurs d'effacement. Ils permettent de corriger des pertes d'information. Un cas d'usage est par exemple un système de stockage distribué où lorsqu'un disque tombe en panne, il faut pouvoir reconstruire l'information perdue. Un autre cas d'usage concerne les transmissions sans-fil. Lorsque les erreurs dans un paquet d'information n'ont pas pu être corrigées par les couches basses de la pile protocolaire, ce paquet est considéré comme perdu et peut donc être reconstruit à l'aide de ces codes.

1.2 Motivations et objectifs

Les besoins industriels, toujours plus exigeants, nécessitent des codes ayant des vitesses d'exécution de plus en plus élevées. Par exemple, avec l'arrivée des réseaux de communications 3G/4G, et bientôt 5G, les services de diffusion de contenus multimédias se sont démocratisés sur les terminaux mobiles. En revanche, ces derniers possédent des ressources limitées, comme un processeur basse consommation, avec une puissance de calcul réduite par rapport aux machines de type serveur. De plus, leur mobilité faisant varier la qualité des communications, les codes à effacement font partie des techniques permettant l'amélioration de la qualité de ces services. L'intérêt d'un code dans les applications dépend de ses propriétés mais aussi de l'implémentation.
Par conséquent, les implémentations de ces codes doivent être les plus performantes possibles, permettant un traitement des données rapide et efficace.

La plupart des codes à effacement sont basés sur les corps finis, définissant les opérations d'addition et de multiplication sur un ensemble fini d'éléments tel que chaque élément non nul possède un unique inverse pour la multiplication.
Plusieurs implémentations permettant d'effectuer des opérations dans un corps fini ont été étudiées dans le contexte des codes à effacement. Afin de simplier les calculs, ces codes utilisent des corps finis de caractéristique 2, définissant l'addition comme un simple *xor*. En revanche, l'opération de multiplication est plus complexe à réaliser. L'une des solutions pour effectuer ces opérations est alors de garder en mémoire, dans une table de correspondance le résultat de la multiplication de tous les couples d'éléments du corps, nécessitant beaucoup d'accès mémoire, pouvant ralentir dans certains cas les calculs. Une autre solution est de réaliser la multiplication par un ensemble prédéfini de *xor*. Cette méthode, dite *xor-based*, a été particulièrement étudiée pour les petits corps finis. En effet, le nombre de *xor* a effectuer dépend directement de la taille du corps et du polynome irréductible définissant le corps fini.

L'objectif principal de cette thèse est de réduire la complexité des codes à effacement basés sur les corps finis et de proposer des implémentations efficaces. Pour cela, nous introduisons une méthode permettant de remplacer les opérations dans un corps fini par des opérations dans un anneau. De cette manière, il est possible de travailler avec une structure algébrique dans laquelle les opérations telles que la multiplication peuvent être beaucoup plus simples. Nous définissons plusieurs transformées entre le corps fini et l'anneau ayant des propriétés différentes. Puis, pour chacune d'entre elles, nous évaluons la complexité en nombre d'opérations. Ce résultat nous permet de comparer la complexité globale du codage et du décodage avec celles des méthodes travaillant directement dans le corps fini. Nous montrons que le passage dans un anneau nous permet de réduire drastiquement le nombre d'opérations. Nous étudions également la possibilité de réduire la complexité des codes à effacement en appliquant un réordonnancement des opérations dans l'anneau. Cette technique est très employée dans les codes basés sur les corps finis.

En utilisant certaines propriétés des anneaux, nous proposons plusieurs techniques permettant de réduire le nombre d'opérations nécessaires au codage. Nous analysons les gains obtenus par rapport à un réordonnancement des opérations appliqué à des éléments d'un corps fini.

En plus de cette étude théorique, nous proposons une implémentation efficace d'un code MDS basé sur certains corps finis en utilisant des transformées rapides. Cette implémentation met en avant une technique permettant d'appliquer les transformées sur les éléments du corps fini à la volée par le processeur, travaillant directement avec les registres de ce dernier et évitant toute écriture mémoire inutile susceptible de réduire les performances. Nous évaluons les performances sur plusieurs architectures matérielles, et pour plusieurs paramètres de code. Nous comparons notre méthode avec les meilleures implémentations à notre connaissance dans le contexte des codes MDS.
Les processeurs possédant de plus en plus de cœur de calculs, y compris sur des terminaux mobiles, nous avons également vérifié le passage à l'échelle d'une telle méthode en analysant les performances d'exécution sur plusieurs cœur de calculs en parallèle pour les opérations de codage et de décodage.

1.3 Plan du manuscrit

Cette thèse se découpe en trois parties.
Tout d'abord, dans le Chapitre 2, nous présentons sous la forme d'un état de l'art les principaux concepts de la théorie de l'information, des codes à effacement et de leurs implémentations en utilisant l'arithmétique des corps finis. Nous passons en revue différents types de codes à effacement qui utilisent les corps finis, puis nous détaillons les différentes méthodes permettant d'effectuer des opérations dans un corps fini.

Le Chapitre 3 regroupe l'ensemble des résultats théoriques. D'abord, nous présentons le contexte algébrique dans lequel nous nous situons. Puis nous détaillons les transformées permettant de passer d'éléments de certains corps finis vers des éléments d'un anneau polynomial avant d'évaluer la complexité de cette méthode. Enfin, nous présentons certaines techniques de réordonnancement des opérations permettant de réduire d'avantage la complexité au codage.

Dans le Chapitre 4, nous rassemblons toutes les analyses de performance que nous avons effectuées. D'abord, nous présentons une implémentation complète d'un code à effacement MDS basé sur les corps finis. Puis nous présentons les résultats de cette implémentation pour plusieurs paramètres de codes. L'analyse de performances sera faite sur plusieurs architectures matérielles largement utilisées dans les applications pratiques actuelles.

Enfin, le Chapitre 5 donnera les conclusions de ces travaux ainsi que les perspectives futures.

Chapitre 2

Etat de l'art

Sommaire

2.1 Théorie des codes

Les problématiques étudiées dans ce manuscrit ont pour origine la théorie de l'information. Nous allons donc commencer par rappeler dans ce chapitre les principes fondamentaux de cette théorie et ses principales applications.

2.1.1 La théorie de l'information

En 1948, Claude Shannon introduit la théorie de l'information [50] dans laquelle il formalise en particulier le problème de la transmission d'information en assurant un certain niveau de fiabilité pour un coût de redondance minimum. Plus spécifiquement, Shannon démontre que pour n'importe quel type de canal de communication, il est possible de transmettre sur celui-ci avec un taux d'erreur arbitrairement proche de zéro tant que le taux de codage reste inférieur à un seuil appelé la capacité du canal. Toutefois, cette preuve n'est pas constructive car Shannon ne donne pas de construction de codes correcteurs atteignant la limite prévue par son théorème du codage de canal. Ce théorème motive toujours la construction de méthodes générant de façon efficace la redondance afin que le destinataire puisse soit détecter que le message a été altéré, soit corriger lui-même les éventuelles erreurs. l'ensemble des travaux relatifs à la construction des codes correcteurs d'erreurs, aussi appelé codage canal, constitue la théorie de codes.

Dans de nombreuses applications soumises à des perturbations, comme la communication sur un canal bruité ou le stockage de données distribuées sur plusieurs serveurs, le codage canal peut servir à détecter et corriger tout ou partie des erreurs ou également à reconstruire des effacements. Considérons un système de communication avec lequel on souhaite envoyer un certain nombre de symboles d'informations à travers un canal bruité :

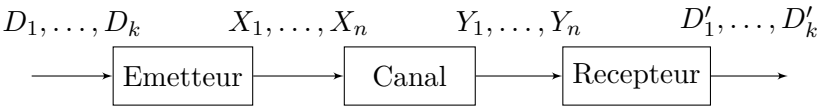


FIGURE 2.1 – Modèle simplifié d'un système de communication

L'objectif est de transmettre k *symboles d'informations* supposés aléatoires $D = (D_1, \ldots, D_k)$ représentant des données. A l'aide d'une fonction de codage, l'émetteur associe au mot de k bits d'information D un mot de longeur n, noté $X = (X_1, \ldots, X_n)$, qu'il envoie sur le canal. Le récepteur essaie de retrouver à partir du mot reçu $Y = (Y_1, \ldots, Y_n)$ les valeurs des bits d'informations D transmis. Par une fonction de décodage, le récepteur va produire le mot $D' = (D'_1, \ldots, D'_k)$. Si $D = D'$, alors, il n'y a pas d'erreur.

Le codage s'intéresse à la conception des fonctions de codage et de décodage. Ces dernières vont entre autres dépendre du type de canal sur lequel l'information sera envoyée.

Source d'information : Une source d'information discrète est représentée par une variable aléatoire discrète X qui prend ses valeurs dans un alphabet $S = \{s_0, \ldots, s_n\}$ avec une distribution de probabilités $P = \{p_0, \ldots, p_n\}$ où p_i représente ainsi la probabilité d'occurence du symbole s_i.

Canal de communication : Toute communication s'opère sur un canal de communication entre une source d'information et un destinataire. Pour chaque séquence de n symboles d'entrée $x_1, \ldots, x_n$, un canal produit en sortie une séquence de n symboles $Y_1, \ldots, Y_n$. Les symboles d'entrée appartiennent à un alphabet $\mathcal{X}$, dit alphabet d'entrée. Les symboles de sortie appartiennent à l'alphabet de sortie $\mathcal{Y}$. Pour chaque $t \in \{1, \ldots, n\}$, le symbole de sortie Y_t est obtenu à partir du symbole d'entrée x_t selon une loi de transition qui détermine la probabilité d'obtenir $Y_t \in \mathcal{Y}$ à la sortie du canal. Cette probabilité est notée $P(Y|X)$. La connaissance du canal nous donne ainsi la loi de transition pour chaque couple (x_t, Y_t).

Un canal est donc caractérisé par le triplet $(X, Y, P(Y|X))$ où :

— $\mathcal{X}$ est un alphabet fini contenant toutes les valeurs possibles à l'entrée du canal

— $\mathcal{Y}$ est un alphabet fini contenant toutes les valeurs possibles à la sortie du canal
— $P(Y|X)$ représente la loi de transition déterminant comment le symbole Y_t est obtenu à partir de x_t

FIGURE 2.2 – Modèle simplifié d'un canal de communication

L'entropie : Soit X une variable aléatoire sur l'alphabet $\mathcal{X}$ avec une loi de probabilité P_X, l'entropie permet de déterminer le degré d'incertitude contenue dans X, l'entropie de cette variable est définie de la manière suivante :

$$H(X) = -\sum_{x \in \mathcal{X}} P_X(x) \log_2(P_X(x))$$

Dans le cas d'une variable aléatoire binaire de probabilité p, la fonction d'entropie binaire est définie de cette manière :

$$H_b(p) = -p \log_2(p) - (1-p) \log_2(1-p)$$

Ainsi, si X est une variable aléatoire binaire prenant la valeur 1 avec une probabilité p et la valeur 0 avec une probabilité $(1-p)$, nous avons :

$$H_b(p) = H(X)$$

L'entropie conjointe : Soient X et Y deux variables aléatoires sur des alphabets finis $\mathcal{X}$ et $\mathcal{Y}$ avec une loi de probabilité conjointe $P_{XY}(x, y)$. L'entropie conjointe mesurant la quantité d'information contenue dans X et Y est définie par :

$$H(X, Y) = -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p_{x,y} \log_2(p_{x,y})$$

L'entropie conditionnelle : L'entropie conditionnelle, mesurant l'entropie restante provenant d'une variable aléatoire X, connaissant parfaitement une variable aléatoire Y est définie par :

$$H(X|Y) = H(X, Y) - H(Y)$$

L'information mutuelle : On note $I(X, Y)$ l'information que l'on peut obtenir sur X à partir de Y, appellée information mutuelle. Elle s'exprime de la manière

suivante :

$$I(X,Y) = H(X) - H(X|Y)$$

Dans le cas d'un canal parfait (sans erreurs), $H(X|Y) = 0$. Ainsi, la connaissance de Y permet de déterminer X : l'information mutuelle est maximale. A l'inverse, lorsque le canal est très mauvais, $H(X|Y) = H(X)$, l'information mutuelle est nulle et la connaissance de Y n'apporte aucune information sur X. La communication est impossible.

Capacité d'un canal : La capacité C d'un canal de communication se définit comme le maximum de l'information mutuelle sur toutes les lois de probabilités possibles pour X. Elle s'exprime de la manière suivante :

$$C = \max_X \{I(X,Y)\}$$

Ainsi, dans le cas d'un canal optimal, $C = \max_X \{H(X)\}$. A l'inverse, dans le pire cas où $H(X|Y) = H(X)$, la capacité est nulle : $C = \max_X \{H(X) - H(X|Y)\} = 0$.

2.1.1.1 Le canal binaire symétrique

Afin d'illustrer ces concepts, prenons l'exemple du canal binaire symétrique ou « Binary Symmetric Channel »(BSC). Il s'agit du canal dont les entrées et sorties ont valeur dans $\{0,1\}$ et où la probabilité d'erreur dans la transmission d'un symbole est de p.

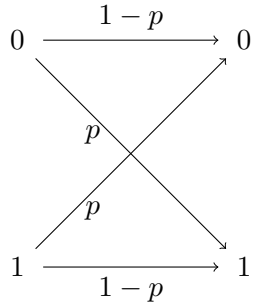


FIGURE 2.3 – Le canal binaire symétrique

Ainsi, la capacité du canal vaut :

$$C = \max_X \{I(X,Y)\} = 1 - H_b(p)$$

La capacité du canal symétrique binaire vaut dont 1 si la probabilité d'erreur p est nulle. A l'inverse, lorsque p vaut $\frac{1}{2}$, $H_b(p) = 1$ c'est à dire son maximum. La capacité d'un canal binaire symétrique de paramètre 0.5 est donc nulle. C'est le plus mauvais canal binaire.

2.1.1.2 Le canal binaire à effacement

A titre de comparaison avec le précédent canal, nous allons introduire un autre type de canal dont les erreurs se représentent par des effacements. C'est en 1955, que Peter Elias publia le modèle du canal binaire à effacement ou « Binary Erasure Channel »(BEC) [19]. Il s'agit du canal dont les entrées ont valeur dans $\{0, 1\}$, mais les sorties ont valeur dans $\{0, 1, \Delta\}$, le symbole Δ représentant un effacement (une perte). La probabilité d'effacement d'un symbole est notée p. L'information transmise sur le canal n'est donc pas altérée, elle est soit reçue, soit perdue :

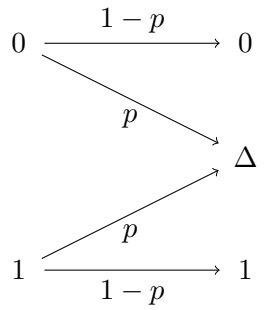


FIGURE 2.4 – Le canal binaire à effacement

La capacité de ce canal vaut :

$$\begin{aligned} C_{BEC} &= \max_X\{(1-p)H(X)\} \\ &= (1-p) \end{aligned}$$

Ainsi, la capacité d'un canal binaire à effacement de paramètre p est égale à $1 - p$.
En comparant le canal binaire symétrique et le canal binaire à effacement, on constate que pour une même probabilité p de perte (ou effacement), la capacité du canal binaire à effacement est supérieure à celle du canal binaire symétrique. En effet, sur le BEC, l'emplacement des effacements est connu. Une étape supplémentaire est nécessaire sur le BSC pour détecter quel symbole a été altéré. Cette étape peut se révéler couteuse en termes de redondance supplémentaire sur le canal et également en complexité.

2.1.1.3 Quelques exemples de canaux à effacements

En pratique, les effacements ne concernent pas des bits isolés. Ils sont regroupés en un ensemble de données, comme un fichier, un paquet émis sur le réseau ou un disque dur. Par exemple, sur l'Internet, l'information est transmise en paquets

sous la forme de datagrammes IP. Que ce soit à cause d'une erreur de routage, de congestion, de connexion intermitente ou de transmission sans fil de mauvaise qualité, certains paquets peuvent être perdus. Autre exemple, un système de stockage distribué contenant des fichiers : lorsqu'un noeud de stockage disparait, certains fichiers sont détruits ou effacés.

2.1.2 Les codes correcteurs d'erreurs

Grâce aux travaux de Shannon, on sait que pour fiabiliser une transmission sur un canal comportant des erreurs, une fonction de codage de l'information est nécessaire pour atteindre un taux d'erreur final négligeable. Dans la suite, nous entendrons par *erreurs* des effacements et nous présenterons plusieurs constructions de codes après avoir rappelé quelques notions d'algèbre utilisées.

2.1.2.1 Les structures algébriques utilisées par les codes

Nous allons ici présenter quelques concepts et outils mathématiques nécessaires à la construction de ces codes. En effet, nous aurons besoin de certaines structures algébriques permettant d'effectuer des opérations sur un ensemble d'éléments. Par exemple, les fonctions de codage doivent être inversibles et cela nécessitera des structures permettant d'inverser tous les éléments de l'ensemble.

Groupe

Un ensemble G muni d'une loi de composition, notée $*$, est appelé un groupe si et seulement si, pour tout a, b et c dans G, cette loi satisfait les trois propriétés suivantes :

— $*$ est associative : pour tout $a, b, c \in G$, $a * (b * c) = (a * b) * c$.

— Il existe un élément neutre $e \in G$, tel que $a * e = e * a = a$.

— Pour tout $a \in G$, $\exists a' \in G$ tel que $a * a' = a' * a = e$.

Un tel groupe est noté $(G, *)$ ou G.

Si G contient un nombre fini d'éléments, on dit que c'est un groupe fini et on note $|G|$ le nombre d'éléments du groupe appelé cardinal.

Si la loi de composition est commutative, alors G est dit abélien.

Pour $a \in G$, on note a^n la répétition de la loi $*$, $a * \cdots * a$, portant sur n termes égaux à a pour tout $n \in \mathbb{N}$.

Si un élément $g \in G$ est tel que pour tout $a \in G$, il existe $i \in \mathbb{Z}$, tel que $a = g^i$, alors g est un générateur du groupe $(G, *)$ ou une racine primitive. Si un groupe possède un générateur, il est dit cyclique.

De plus, soit H un sous-ensemble de G. H est un sous-groupe si et seulement si :

— $e \in H$

— $\forall x, y \in H$, $x * y \in H$

— $\forall x \in H$, si $x * y = e$, $y \in H$

Anneaux

Un ensemble A muni de deux lois de composition internes, notées $*$ (appelé multiplication) et $+$ (appelé addition), est un anneau si et seulement si, pour tout a, b et c dans A, ces deux lois satisfont les propriétés suivantes :

— $(A, +)$ est un groupe abélien avec l'élement neutre $e = 0$.

— $*$ est associative : $\forall a, b, c \in A,\ a * (b * c) = (a * b) * c$.

— $*$ est distributive sur $+$: $\forall a, b, c \in A,\ a * (b + c) = (a * b) + (a * c)$ et $(b + c) * a = (b * a) + (c * a)$.

Si $*$ possède un élément neutre dans A, A est dit unitaire. S'il existe, cet élément est noté 1. Si de plus $*$ est commutative, A est dit commutatif. Tous les éléments d'un anneau ont un opposé qui est l'inverse par la loi $+$. En revanche, ils n'ont pas forcément d'inverse par la loi $*$.

Deux anneaux $(A, +_A, *_A)$ et $(B, +_B, *_B)$ sont isomorphes lorsqu'il existe une bijection $f : A \to B$ vérifiant pour tout x et y dans A :

$$f(x +_A y) = f(x) +_B f(y), f(x *_A y) = f(x) *_B f(y)$$

Un idéal I est un sous-groupe d'un anneau A pour la loi $+$ qui est absorbant pour la loi $*$: pour $g \in I$, le produit $a * g$ reste dans I pour n'importe quel élément a de l'anneau A. Pour tout $x \in A$, la partie $Ax = \{ax; a \in A\}$ est un idéal de A appelé idéal engendré par x. Un idéal I de A est dit principal s'il existe un générateur x (tel que $I = Ax$). Un anneau est principal si et seulement si tout idéal est principal.

Corps

Un anneau $(K, +, *)$ est un corps si et seulement si c'est un anneau unitaire tel que tout $a \neq 0$ dans $(K, +, *)$ possède un élément symétrique a^{-1} tel que $a * a^{-1} = a^{-1} * a = 1$. De plus, un corps doit contenir au moins deux éléments distincts : 0 et 1. De même, un corps est dit fini si il possède un nombre fini d'éléments.

Il existe un corps fini $\mathbb{F}_q$ à q éléments si et seulement si $q = p^m$ où p est premier et $m \geq 1$. Par conséquent, on peut construire un corps fini $\mathbb{F}_p$ à p éléments appelé corps premier. Ce dernier contient des éléments qui peuvent être associés aux différents entiers inférieurs à p et les opérations $+$ et $*$ peuvent être associées à l'addition et la multiplication des entiers modulo p. L'exemple le plus simple étant $\mathbb{F}_2$, contenant deux éléments : 0 et 1. La construction d'un corps fini $\mathbb{F}_{p^m}$ est possible par extension de $\mathbb{F}_p$, à condition de trouver un polynome $\pi(x)$ de degré m dont les coefficients sont dans $\mathbb{F}_p$ et qui est irreductible sur $\mathbb{F}_p$.

Soit K' un sous-ensemble non-vide d'un corps K. K' est un sous-corps si et seulement si :

— $1_K \in K'$

— K' est stable par $+$ et $*$

— $\forall a \in K'$, $-a \in K'$ et $\forall a \in K' \setminus \{0\}, a^{-1} \in K'$

Enfin, deux corps sont isomorphes si ils sont isomorphes en tant qu'anneau.

Définition 1. *Soit $\pi(x)$ un polynome de degré m et irreductible sur $\mathbb{F}_p$. L'ensemble des polynomes en x de degré $\leq m-1$ et à coefficients dans $\mathbb{F}_p$ calculés modulo $\pi(x)$ forment un corps de cardinalité p^m et de caractéristique p. Tous les corps finis de cardinalité p^m sont isomorphes. Cela garantit l'unicité du corps $\mathbb{F}_{p^m}$.*

Définition 2. *Dans un corps fini $\mathbb{F}_q$ avec $q = p^m$, l'ordre d'un élément est la plus petite puissance strictement positive de cet élément égale à l'élement neutre par la multiplication, noté 1 : si $\alpha^{q-1} = 1$ et $\forall r, 0 < r < q-1$ avec $\alpha^r \neq 1$, l'ordre de l'élément α est $q-1$. Lorsque cet ordre est égal à $q-1$, l'élément est appelé racine primitive du corps. Un tel élément est appelé générateur car chaque élément du corps pourra être défini comme une puissance de cette racine primitive.*

Notons que les membres de $\mathbb{F}_{p^m}$ peuvent également être définis comme les classes résiduelles des polynomes en x à coefficient dans $\mathbb{F}_p$ modulo $\pi(x)$:

$$\mathbb{F}_{p^m} = \mathbb{F}_p[X]/(\pi(x))$$

Soit α la classe résiduelle de x. Ainsi, $\pi(\alpha) = 0$. Autrement dit, $\pi(x) = 0$ admet une racine α dans $\mathbb{F}_{q^m}$.

Espace vectoriel

Un ensemble $\mathbb{E}$ est un espace vectoriel sur un corps K, appelé alors K-espace vectoriel, s'il est muni d'une loi de composition interne $+$ et d'une loi externe à gauche «$\cdot$», telles que :

— $(\mathbb{E}, +)$ est un groupe commutatif
— Pour tout $a \in E$, $1_K \cdot a = a$
— Pour tous $\lambda, \mu \in K$ et $a \in \mathbb{E}$, $\lambda \cdot (\mu \cdot a) = (\lambda * \mu) \cdot a$
— Pour tous $\lambda, \mu \in K$ et $a \in \mathbb{E}$, $(\lambda + \mu) \cdot a = \lambda \cdot a + \mu \cdot a$
— Pour tous $\lambda \in K$ et $a, b \in \mathbb{E}$, $\lambda \cdot (a + b) = \lambda \cdot a + \lambda \cdot b$

Les éléments de $\mathbb{E}$ sont appelés vecteurs et les *éléments* de K sont appelés *scalaires*. L'ensemble $\{0, 1\}$ muni des opérations d'addition et de multiplication est un corps commutatif, noté $\mathbb{F}_2$. L'ensemble des tableaux de bits de taille n peut donc être muni d'une structure d'espace vectoriel. L'ensemble des mots de code est alors $\mathbb{F}_2^n$.

2.1.2.2 Terminologie relative aux codes correcteurs d'erreurs

Définition 3. *Un code en bloc linéaire $\mathscr{C}$ de longeur n et de dimension k, noté (n, k), est un sous-espace vectoriel de dimension k de l'espace vectoriel $\mathbb{F}_q^n$. Les éléments de $\mathscr{C}$ sont appelés les mots de code.*

A partir de sa longeur et de sa dimension, on peut définir le rendement d'un code, également appelé taux de codage. Il correspond à la quantité de redondance ajoutée par le code.

Définition 4. *Le rendement d'un code en bloc $\mathscr{C}$ de longeur n est :*

$$R = \frac{\log_2(|\mathscr{C}|)}{n} = \frac{k}{n}$$

Définition 5. *Une matrice $G \in \mathcal{M}(\mathbb{F}_q)_{k,n}$ est dite génératrice du code si ses k lignes forment une base du code, avec $\mathcal{M}(\mathbb{F}_q)_{k,n}$ l'ensemble des matrices (k, n) à coefficients dans $\mathbb{F}_q$.*

Soit $D \in \mathbb{F}_q^k$ et G la matrice génératrice d'un code (n, k). D est encodé en un mot de code $X \in \mathbb{F}_q^n$ par la multiplication de la matrice génératrice :

$$X = DG$$

Lorsque le vecteur source se retrouve dans le vecteur encodé, on parle d'un code systématique. A une permutation de colonnes près, la matrice génératrice contient alors la matrice identité :

$$G_{sys} = [Id_k|C]$$

Définition 6. *Soit $\mathcal{C}$ un code (n, k). On dit que $H \in \mathcal{M}(\mathbb{F}_q)_{(n-k),n}$ est une matrice de parité du code si et seulement si $\forall X \in \mathcal{C}$, $HX^t = 0$ et H est de rang $(n-k)$.*

La construction d'une matrice de parité H à partir de la matrice génératrice sous sa forme systématique peut se faire de la manière suivante :

$$H = [-C^T|Id_k]$$

Le décodage d'un code correcteur a pour objectif de retrouver la séquence ayant été la plus probablement envoyée. Pour comparer plusieurs mots de code à une séquence reçue, il est nécessaire d'introduire une notion de distance. La distance utilisée est généralement la distance de Hamming qui découle de la fonction de poids de Hamming.

Définition 7. *Le poids de Hamming d'une chaine de symboles sur un alphabet donné est le nombre de symboles qui diffèrent du symbole nul.*

Définition 8. *La distance de Hamming entre deux vecteurs de symboles a et b est le poids de Hamming du vecteur $a - b$.*

Définition 9. *La distance minimale d'un code linéaire, notée d_{min}, est la plus petite distance de Hamming qui existe entre deux mots de code distincts. A cause de la structure d'espace vectoriel des codes linéaires, c'est aussi le minimum des poids de Hamming de tous les mots de code.*

Ainsi, un code (n, k) de distance minimale d sera noté (n, k, d). La distance minimale d'un code nous donne une borne sur le nombre de corrections garanties par un code donné. En effet, un code (n, k, d) sera toujours capable de corriger t

erreurs et e effacements avec $d \geq 2t + e + 1$. Enfin, cette distance minimale possède une borne supérieure. Cette borne dépend de la longueur, de la dimension du code, ainsi que du corps sur lequel est défini le code.

Définition 10. *Un code linéraire de longeur n et de dimension k est dit MDS si : $n - k = d_{min} - 1$. Le code atteint ce qu'on appelle la borne de Singleton [52].*
Dans le cadre des codes à effacement, un code est dit MDS si et seulement si tout sous-ensemble de k symboles reçus parmi les n générés permet de décoder les k symboles sources.

Le code de Hamming est une classe de codes linéraires binaires. Ces codes font partie des tout premiers codes correcteurs d'erreurs. Ils ont été publiés en 1950 par Richard Wesley Hamming [23].

Définition 11. *Un code binaire de Hamming $\mathscr{H}$ est un code linéaire binaire permettant de corriger une erreur. La longeur du code est $n = 2^r - 1$, la dimension est $k = 2^r - 1 - r$ et la distance minimale est toujours 3 (dans sa forme non généralisée), avec r le nombre de bits de parité. Sa matrice de parité H possède n colonnes représentant les $2^r - 1$ vecteurs de longeur r. Il est possible de réordonner H pour obtenir un code systématique.*

La limite de cette classe de code est qu'il n'est possible de corriger qu'une seule erreur. Une autre famille de codes est capable de corriger plusieurs erreurs : les codes BCH [15], [24] (de leurs inventeurs : Bose, Ray-Chaudhuri et Hocquenghem).

Définition 12. *Les codes BCH sont des codes cycliques sur $\mathbb{F}_q$ qui permettent de corriger t erreurs. En prenant n et q premiers, α un élément de $\mathbb{F}_{q^m}$ d'ordre n, nous avons la factorisation $X^n - 1 = P_1(X) \dots P_k(X)$ en polynomes irreductibles de $\mathbb{F}_q[X]$. Comme n est premier avec q, toutes les racines de $X^n - 1$ sont distinctes et sont donc les racines d'un et un seul des facteurs irreductibles P_i. Soit i_k l'indice du polynome irreductible dont α^k est racine, le polynome générateur d'un code BCH est :*

$$g(x) = ppcm\{P_{i_{k+1}}(X), \dots, P_{i_{k+2t}}(X)\}$$

Ces codes ont permis de définir les codes de Reed-Solomon. Ces derniers peuvent être vus comme un cas particulier des codes BCH. Ils ont été introduits par Irvine Reed et Gustave Solomon [47] en 1960 et ont été rendus célèbres par leur utilisation sur les « Compact Disk », pour les transmissions avec la sonde Voyager ou encore dans des standards comme DVB-S (Digital Video Broadcast - Satellite) ou CCSDS (Consultative Committee for Space Data Systems). Ils vérifient $n = q - 1$. Dans ce cas, il exite toujours des racines primitives n-ième de l'unité. Ainsi, le polynome générateur d'un code de Reed-Solomon ($n = q - 1, k = n - r$) avec $r < n$ est de la forme :

$$g(X) = \prod_{i=t}^{t+r-1} (X - \alpha^i)$$

Les codes de Reed-Solomon étant des codes BCH, la distance minimale $\delta \geq r + 1$. De plus, en tant que code linéaire, le code atteint la borne Singleton, soit,

$\delta = n-k+1$. Ainsi, un code Reed-Solomon est un code BCH optimal avec $\delta = r+1$. Il suffit donc de trouver un corps fini avec q éléments pour construire un code de Reed-Solomon de longeur $n_{max} = q - 1$ pour le canal à erreur.

2.2 Les codes correcteurs d'erreurs pour le canal à effacement

Dans cette partie, nous allons présenter les différents types de codes qui peuvent être utilisés sur un canal à effacement. Supposons un canal à effacement avec une probabilité d'effacement p. La capacité du canal est donc de $1 - p$. Afin de transmettre de manière fiable k symboles sur celui-ci, on choisit un code MDS dont le rendement R est inférieur à la capacité du canal.
Soit $m = (m_0, \ldots, m_{k-1})$ le message source à encoder, constitué de k symboles de $\mathbb{F}_q$. Soit G une matrice génératrice de dimension (n, k). L'opération d'encodage consiste à multiplier le message m par la matrice génératrice.

2.2.1 Les codes Maximum Distance Separable (MDS)

Un code à effacement MDS est construit sur un corps fini. Il utilise une matrice génératrice dite MDS, notée G dont les coefficients sont des éléments de ce corps fini. L'opération de codage consiste à générer n combinaisons linéaires de k symboles sources. Au décodage, à partir de k symboles codés, une sous-matrice de G correspondant aux symboles codés reçus et aux symboles sources manquants est calculée puis inversée afin de déterminer les opérations inverses pour reconstruire les k symboles sources originaux. Ils ont l'avantage d'être optimaux mais ont une complexité quadratique pour la plupart.

Les codes de Reed-Solomon

Les codes Reed-Solomon sont les codes MDS les plus classiques. L'opération de codage consiste en une évaluation en n points du polynome représenté par les données sources. Pour cela, on utilise une matrice de Vandermonde à coefficients dans $\mathbb{F}_q$. Par construction, une telle matrice est MDS.

Une matrice de Vandermonde de taille $m*m$, notée $V(x_1, \ldots, x_m)$ est construite à partir de m termes distincts $x_1, \ldots, x_m$:

$$V(x_1, \ldots, x_m) = \begin{pmatrix} x_1^0 & x_1^1 & x_1^2 & \cdots & x_1^{m-1} \\ x_2^0 & x_2^1 & x_2^2 & \cdots & x_2^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_m^0 & x_m^1 & x_m^2 & \cdots & x_m^{(m-1)} \end{pmatrix}$$

Le déterminant est égal à :

$$det(V) = \prod_{1 \leq i < j \leq m} (x_j - x_i)$$

Pour un code de Reed-Solomon, la matrice génératrice est construite à partir d'un élément primitif du corps fini, noté α. En effet, ce dernier génère tous les éléments du corps, excepté l'élément neutre par la loi de composition interne $+$, et peut donc générer une séquence de m termes distincts à condition que le corps fini possède au moins $m+1$ éléments. Cette matrice est notée $V(\alpha)$:

$$V(\alpha) = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{m-1} & \alpha^{2(m-1)} & \cdots & \alpha^{(m-1)(m-1)} \end{pmatrix}$$

Pour construire une matrice génératrice $k \times n$ d'un code MDS à partir d'une matrice de Vandermonde, il suffit de considérer une matrice de Vandermonde carrée $r \times r$ telle que $r \geq n$ et d'en prendre la sous-matrice $k \times n$ formée des k premières lignes et des n premières colonnes. Sur cette matrice génératrice, on peut voir que toute sous-matrice carrée $k \times k$ est une matrice de Vandermonde et donc qu'elle est inversible. C'est cette propriété fondamentale qui est utilisée par le décodeur pour reconstruire les paquets d'informations à partir de k paquets quelconques reçus. En effet, à tout ensemble de k paquets reçus, il peut associer une sous-matrice $k \times k$ de la matrice génératrice, l'inverser puis multiplier cette matrice inverse par les paquets reçus.
Le principal problème des codes de Reed-Solomon est que lorsqu'ils sont construits directement avec une matrice de Vandermonde, ils ne sont pas systématiques, c'est-à-dire que leur matrice génératrice ne contient pas la matrice identité. Il faut noter que si l'on construit une matrice génératrice qui contient l'identité sur les k premières colonnes et une sous-matrice d'une matrice de Vandermonde sur la partie « redondance », ce code n'est pas forcément MDS. Ceci s'explique par le fait que si l'on veut que le code soit MDS, il faut que toute sous-matrice $k \times k$ soit inversible. Comme une partie de cette sous-matrice peut contenir des colonnes appartenant à l'identité, il est nécessaire que toute sous-matrice $i \times i$ (où $i \leq k$) de la matrice de Vandermonde soit inversible. Les matrices de Vandermondeà coefficients dans un corps fini ne possèdent pas cette propriété. Pour remédier à cela, on peut « systématiser »le code en extrayant une sous-matrice carrée $k \times k$ de la matrice génératrice, en calculant son inverse, puis en multipliant cet inverse par la matrice génératrice [29].

Codes MDS contruits à partir d'une matrice de Cauchy

Une autre construction possible d'un code MDS consiste à utiliser une matrice de Cauchy. Une matrice de Cauchy $m * n$ est une matrice dont les coefficients $a_{i,j}$ sont de la forme :

$$a_{i,j} = \frac{1}{x_i - y_j}; x_i - y_j \neq 0, 1 \leq i \leq m, 1 \leq j \leq n$$

où x_i et y_j contiennent des éléments distincts. Cette construction, à la différence de

celle utilisée avec une matrice de Vandermonde, permet d'obtenir directement une matrice MDS systématique. En effet, toute sous-matrice de Cauchy est également une matrice de Cauchy et est donc inversible. De plus, ces matrices peuvent être utilisées pour construire directement la partie redondance d'une matrice génératrice MDS à laquelle il suffit de concaténer une matrice identité pour obtenir un code systématique.

Les premiers codes utilisant des matrices de Cauchy ont été introduits par McWilliams et Sloane [35]. Elles ont été utilisées pour un code à effacement pour la première fois par Rabin [44] dans le contexte du stockage distribué et pour des transmissions temps-réel fiables [14].

Enfin, il est possible d'utiliser une matrice de Cauchy dans sa forme généralisée, où tous les éléments de la première ligne et de la première colonne de la partie non-systématique sont égaux à 1 :

$$a_{i,j} = \frac{r_i * s_j}{x_i - y_j}; x_i - y_j \neq 0, 1 \leq i \leq m, 1 \leq j \leq n$$

En effet, en choisissant le coefficient s_j égal à $x_1 + y_j$, pour $j = 1, \ldots, m$ et r_i égal à $(x_i + y_1)/s_1$ pour $i = 1, \ldots, k$, alors, tous les éléments de la première colonne et la première ligne de la partie non-systématique sont égaux à 1.

Nous verrons dans la partie suivante que cette construction permet de réduire la complexité de l'encodage d'un code MDS en utilisant une représentation des données spécifique qui permet de réaliser l'opération d'encodage uniquement avec des opérations *xor* lorsqu'on utilise un corps fini de caractéristique 2.

2.2.2 Les codes à matrice creuse

A la différence des codes de Reed-Solomon, les codes à matrice creuse ont une matrice génératrice ou de parité à faible densité. En d'autres termes, ils contiennent en majorité des termes nuls. Cela permet de réduire la complexité de codage en diminuant le nombre d'opérations nécessaires. A la différence des codes de Reed-Solomon qui ont une complexité quadratique, nous verrons que ces codes permettent également d'utiliser des algorithmes qui possèdent une complexité sous-quadratique sans trop réduire les capacités de correction. Malgré ce compromis, ces codes ne sont pas MDS, bien que très proches pour certains. En revanche, la dimension de ces codes peut être très grande avec un alphabet petit.

2.2.2.1 Les codes LDPC

Les codes LDPC ("Low Density Parity Check") ont été inventés au début des années 1960 par Gallager [20]. A l'origine, ces codes consistent à générer des combinaisons linéaires d'un sous-ensemble des symboles sur le corps fini à deux éléments $\mathbb{F}_2$. Les opérations de codage se résument à additionner ou soustraire ($\oplus$ dans $\mathbb{F}_2$) certains symboles entre eux. Plusieurs versions ont été proposées pour améliorer les capacités de correction ou pour permettre une complexité constante pour l'enco-

dage, avec par exemple les LDPC-staircase qui sont disponibles dans la bibliothèque OpenFEC[6].

On peut représenter des codes LDPC par deux manières : un graphe biparti ou une matrice de parité qui est en fait la matrice d'adjacence du graphe.

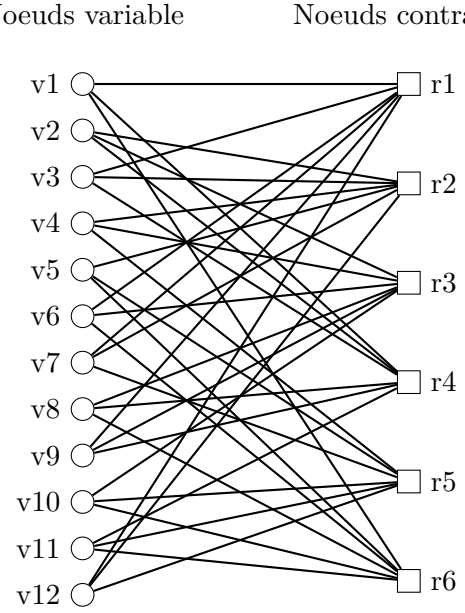


FIGURE 2.5 – Graphe de Tanner d'un code LDPC regulier

Ce graphe représente les équations utilisées pour générer les symboles de redondance, appelés noeuds contrainte à partir des symboles de redondance, appelés noeuds variable. La matrice de parité de ce graphe est la suivante :

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

FIGURE 2.6 – Matrice de parité d'un code LDPC régulier

Plusieurs variantes des codes LDPC existent. Par exemple, on peut citer les codes LDPC-staircase, dont la matrice de parité H se compose de deux parties, H1

et H2. La partie gauche, H1, de taille $(n-k) \times k$, indique dans quelles équations chaque symbole source intervient. Elle est créée de façon aléatoire, avec comme seules contraintes qu'il y ait (au moins) trois éléments différents de 0 par colonne, et au moins deux par ligne. La partie droite, H2, est spécifique au code considéré. Dans le cas de LDPC-staircase, il s'agit d'une matrice « escalier », avec une double diagonale. Les codes LDPC-staircase ont la particularité de permettre un encodage avec une complexité constante, c'est à dire que chaque symbole de redondance nécessitera le même nombre d'opérations pour être généré, quelle que soit la dimension du code, cela grâce à la structure en double diagonale de la partie droite de la matrice de parité. Pour ce qui est du décodage, si le récepteur reçoit suffisament de symboles, les symboles perdus peuvent être reconstruits itérativement, ne nécessitant pas d'inversion de matrice. Cela n'empêche pas, après avoir reconstruit quelques symboles itérativement, de continuer par un décodage plus complexe mais plus efficace comme une élimination par pivot de Gauss afin d'améliorer les capacités de correction du code.

De la même manière que les codes LDPC binaires, il est tout à fait possible d'utiliser des coefficients sur un corps fini contenant plus d'éléments afin de gagner en capacité de correction au prix d'une augmentation de la complexité.

2.2.2.2 Les codes Tornado

Les codes Tornados ont été publiés par M. Luby en 2001 [33]. Ils consistent en une succession de plusieurs couches appliquant un code LDPC sur les symboles avant d'appliquer un code Reed-Solomon. En générant ainsi des symboles intermédiaires, la complexité de codage est linéaire pour une capacité de correction proche de l'idéal.

2.2.2.3 Les codes sans rendement (rateless)

A la différence des codes précédemment, les codes sans rendement ou codes fontaine peuvent théoriquement générer une quantité infinie de symboles de redondances à partir d'un ensemble fini de symboles sources. Le principe reste le même que les codes précédents : un symbole de redondance est une combinaison linéaire des symboles sources. Etant donné que le nombre de symboles sources est fini, le nombre de combinaisons linéaires de ces sources est lui aussi fini. Par exemple, avec k symboles sources, le nombre de combinaisons linéaires différentes sur $\mathbb{F}_2$ est 2^k.

2.2.2.4 Les codes LT

Les codes LT (« Luby Transform ») ont été proposés par Luby en 1998 et publiés en 2002 [34]. Il s'agit de codes sans rendement avec lesquels il est possible de générer à la volée des symboles de redondance. Un symbole de redondance est une combinaison linéaire aléatoire de d symboles sources sur $\mathbb{F}_2$. La distribution des degrés d des symboles de redondance influe sur la capacité de correction de ces codes. Dans [34], Luby propose d'utiliser la distribution robuste de Soliton afin d'obtenir des codes asymptotiquement optimaux. Avec cette distribution, le degré moyen est

de $log(k)$, soit une complexité d'encodage de $\mathcal{O}(\log k)$ opérations par symbole. Les opérations de codage sont donc très rapides. Mais le principal inconvénient de ces codes est qu'avec une telle distribution, ces derniers ne sont pas systématiques.

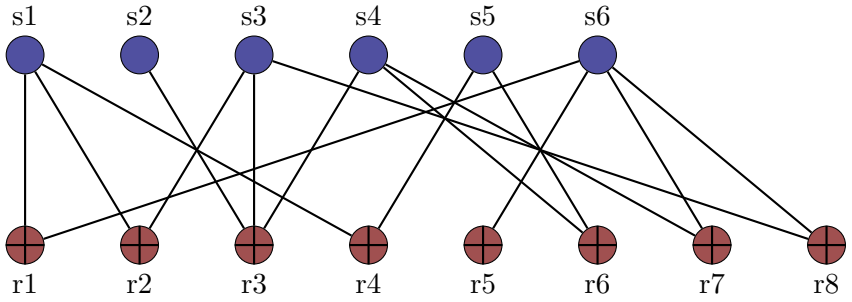


FIGURE 2.7 – Codage avec un code LT $(8, 6)$

2.2.2.5 Les codes Raptor

Les codes Raptor[51] ont été inventés en 2006 comme une amélioration des codes Tornado. Ils corrigent le principal défaut des codes LT, à savoir l'impossibilité d'obtenir un code systématique. Ils permettent d'éviter que la totalité des symboles sources ne soient codés. De plus ils disposent d'une complexité de codage faible ($\mathcal{O}(\log k)$ par symboles).

Pour cela, les k symboles d'information sont d'abord codés en n symboles en utilisant un code $C(n, k)$, appelé précode, capable de récupérer les k symboles à partir des n symboles codés. Les n symboles codés sont ensuite codés en utilisant un code LT capable de récupérer de ces symboles codés les n symboles intermédiaires.

Les codes Raptor utilisent un précode binaire alors que les codes RaptorQ utilisent un précode pouvant travailler sur $\mathbb{F}_q$, améliorant ainsi les capacités de correction.

2.2.2.6 L'approche par géométrie discrète

Une autre approche permettant de construire un code (n, k) est d'utiliser la transformée Mojette [22], version discrète et exacte de la transformée de Radon continue [45] (rééditée et traduite dans [46]). L'opération de codage consiste à générer des mots de codes appelés projections depuis une grille rectangulaire de dimension $P * Q$ représentant les données. Une projection est un ensemble d'éléments appelés *bins*, qui est définie par une direction de projection (p, q), avec $p, q \in \mathbb{Z}$ premiers entre eux. L'ensemble des projections est défini par :

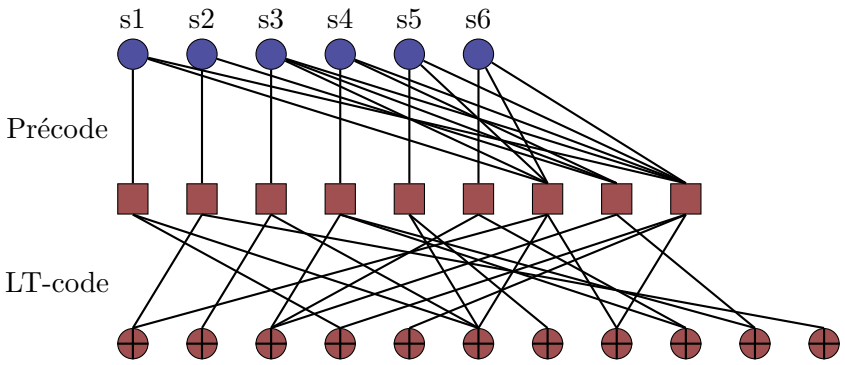


FIGURE 2.8 – Codage avec un code Raptor

$$M_{\{b,p_i,q_i\}}[f] = \sum_{k=0}^{Q-1} \sum_{l=0}^{P-1} f(k,l)\Delta(b - lp_i + kq_i)$$

avec $\Delta(.) = 1$ si $b = lp_i + kq_i$, sinon 0. Le paramètre b correspond à l'index du bin de la projection d'angle (p_i, q_i). Plus précisément, cela signifie que la valeur des bins de la projection suivant la direction (p_i, q_i) résulte de la somme des pixels situés sur la droite discrète d'équation $b = lp_i + kp_i$.

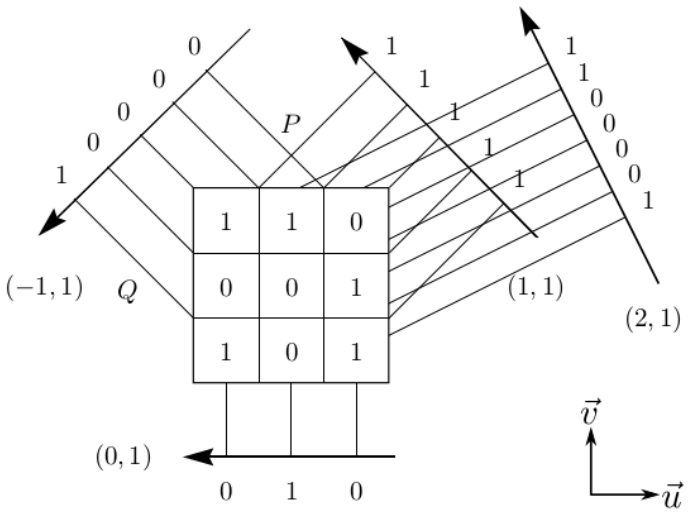


FIGURE 2.9 – Représentation de la transformée Mojette dans $\mathbb{F}_2$ d'une grille d'image $P \times Q = 3 \times 3$ sur laquelle sont calculées 4 projections discrètes $S_4 = \{(0,1),(1,1),(2,1),(-1,1)\}$ avec comme base utilisée les vecteurs unitaires $\vec{u}$ et $\vec{v}$

Cette construction particulière permet d'obtenir un code sous-optimal permettant d'utiliser un algorithme de décodage itératif [38]. Ceci est rendu possible par

le fait que les symboles de redondance représentés par les projections ont une taille légèrement supérieure à celle des symboles source et variable selon l'angle de projection. De plus, dans certaines projections, un bin peut contenir des données sources. Ainsi, de la même manière que les codes LDPC systématiques, il est possible de reconstruire itérativement les données sources. Par conséquent, ce code est considéré comme « quasi-MDS », et non MDS. Enfin, par défaut, les codes utilisant la transformée Mojette sont des codes non-systématiques, mais il est possible d'obtenir un code systématique [18]. Ce code à effacement est utilisé dans un projet open-source appelé RozoFS [8]. Il est également valorisé par la société *Rozo Systems* qui propose une solution de stockage haute performance grâce à ce code [7].

2.2.3 Les codes à fenêtre glissante

Les codes en bloc MDS permettent d'obtenir une capacité de correction optimale. Mais cela est possible au prix d'une latence corrélée en partie à la taille du bloc. En effet, il est nécessaire d'avoir k symboles parmi les n symboles du bloc pour reconstruire les k symboles d'information. Dans le contexte des transmissions temps-réels, utiliser des codes en bloc peut s'avérer problématique en termes de latence. En effet, le fait d'attendre la réception d'un nombre suffisant de symboles avant de pouvoir décoder peut poser problème. Il est donc parfois judicieux d'encoder les données à la volée, amenant possiblement plus de redondance qu'avec un code [55], mais permettant de décoder beaucoup plus rapidement les symboles reçus.

Plusieurs constructions sont possibles, mais le principe est toujours le même : générer des combinaisons linéaires sur un corps fini. Plusieurs paramètres peuvent influer sur la capacité de correction : un grand corps fini permettra d'éviter des dépendances linéaires, des combinaisons incluant un grand nombre de symboles peuvent également entrainer des dépendances linéaires.

2.2.3.1 Les codes convolutifs sur $\mathbb{F}_{2^n}$

Les premiers codes correcteurs d'effacement à fenêtre glissante ont été introduits par Martinian en 2002 [36]. L'objectif était de proposer une solution permettant de corriger des pertes en rafale tout en limitant la latence introduite. Dans cette construction, il utilise un code convolutif, c'est à dire que tous les symboles de redondance sont générés à partir de symboles sources différents mais utilisent les mêmes coefficients sur $\mathbb{F}_{2^n}$.

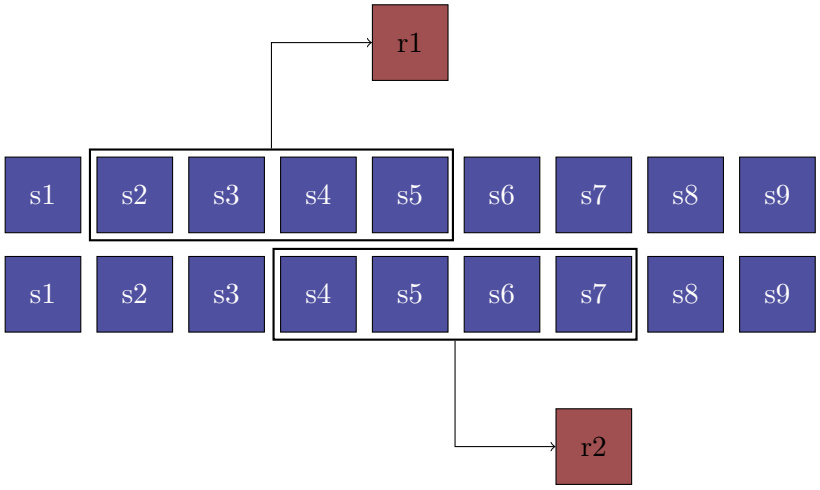


FIGURE 2.10 – Codage avec une fenêtre glissante

$$\begin{pmatrix} \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & 0 & 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & \alpha_{i,j} & 0 & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \end{pmatrix}$$

FIGURE 2.11 – Matrice génératrice d'un code à fenêtre glissante

2.2.3.2 Les codes à combinaisons linéaires aléatoires (RLC)

Il est également possible d'utiliser un générateur pseudo-alétoire pour générer les coefficients [28]. Par exemple, pour chaque symbole de redondance, son identifiant sert de graine au générateur et pour chaque symbole source, le générateur pseudo-aléatoire génère un coefficient. Ainsi, en connaissant uniquement les identifiants des symboles et l'algorithme du générateur pseudo-aléatoire, il est possible de calculer tous les coefficients utilisés pour générer un symbole de redondance donné.

2.2.3.3 Les codes Structured-RLC

Il est également possible d'utiliser des algorithmes qui permettent de générer des combinaisons mélangeant des coefficients binaires et non binaires [37] afin de rendre creuses les matrices génératrices et ainsi travailler sur $\mathbb{F}_2$ ou encore de faire apparaitre des structures dans la matrice qui permettent de décoder par inactivation [43].

2.2.3.4 Les codes à coefficients déterministes

Une autre approche consiste à utiliser la construction des matrices MDS, telles que Cauchy ou Vandermonde. Pour cela, on utilisera les identifiants des symboles

pour identifier les colonnes et lignes correspondantes de la matrice génératrice à utiliser. Cette méthode a l'avantage de pouvoir calculer les coefficients des symboles indépendamment, sans connaitre les précédents. Cette méthode est proposée dans le protocole Tetrys présenté à l'IETF [27].

2.3 Implémentations efficaces des codes à effacement

Dans cette section, nous verrons les différentes manières d'effectuer efficacement des opérations dans un corps fini, puis nous détaillerons quelques algorithmes permettant d'améliorer sensiblement les performances des codes basés sur les corps finis, puis nous finirons par présenter différentes méthodes permettant la réduction de la complexité des codes.

2.3.1 Algorithmes et arithmétique pour l'implémentation des corps finis

Comme nous l'avons rappelé, beaucoup de codes correcteurs d'erreurs, effectuent des opérations dans un corps fini. Ces opérations sont définies par une matrice génératrice, MDS ou non. Dans cette section, nous allons détailler différentes méthodes permettant d'implémenter efficacement des opérations dans un corps fini.

2.3.1.1 La correspondance symboles/données

Nous avons vu différentes méthodes de codage utilisées par différents codes. En pratique, ces codes nécessitent une certaine adaptation pour être utilisés de manière la plus efficace possible. Les opérations de codage ou de décodage ne doivent pas être trop contraignantes en termes de calculs. Par exemple, dans les couches hautes de la pile protocolaire, les codes correcteurs d'effacement utilisés doivent être capables de récupérer les effacements de paquets entiers, unités de données envoyées sur le réseau. Il est bien évident que l'ensemble de ces paquets mis bout-à-bout ne représente pas un seul mot de code à encoder. Au lieu de cela, chaque paquet représente Λ symboles, et les i^{eme} symboles de chaque paquet sont utilisés pour former un mot, comme illustré dans la Figure 2.12.

En prenant un code MDS, systématique ou non, dont la dimension k est égale au nombre de paquets à protéger, un mot de code est généré en multipliant le mot initial par une matrice génératrice. On obtient alors l'ensemble des symboles $s_{i_{(1,j)}}, \ldots, s_{i_{(k,j)}}$. Cette opération est répétée pour chaque ensemble des symboles contenu dans dans les paquets. On obtient alors n paquets codés.

La même approche est utilisée lorsque les codes correcteurs d'effacement sont utilisés pour fiabiliser un système de stockage distribué. Dans ce cas, le codage peut se faire à plusieurs niveaux. On peut par exemple coder un fichier, ou un disque entier. De la même manière, lorsqu'une perte apparait, traduisant par exemple une

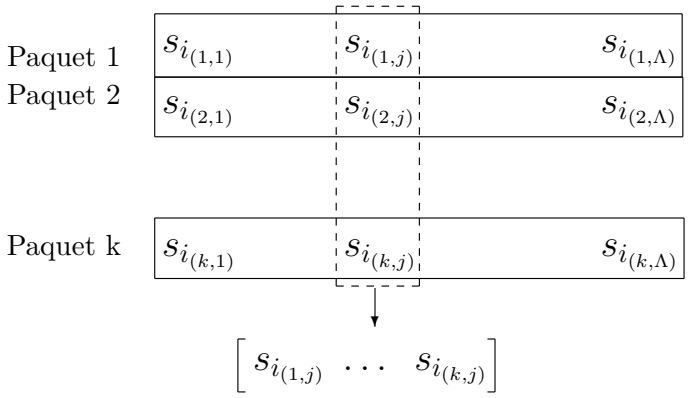


FIGURE 2.12 – Représentation des mots à encoder

panne matérielle d'un disque de stockage, ce sont plusieurs téraoctets de données qui vont être perdus et devront être reconstruits.

$$\left[s_{i_{(1,j)}} \ \ldots \ s_{i_{(k,j)}} \right] \times \left[M_G \right] = \left[s_{c_{(1,j)}} \ \ldots \ s_{c_{(n,j)}} \right]$$

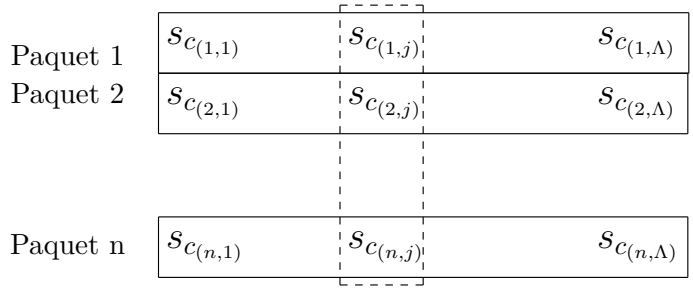


FIGURE 2.13 – Codage par paquets

Ce principe de codage permet d'obtenir n paquets, mais il est possible de tronquer le code et de ne générer qu'une sous-partie des paquets. Chaque paquet s'obtient en multipliant le vecteur $s_{i_{(1,j)}}, \ldots, s_{i_{(k,j)}}$ par une colonne de la matrice génératrice. Cette technique est très utilisée dans les codes au niveau transport car elle permet d'adapter la quantité de redondance envoyée sur le réseau [56].

Enfin, lorsque les unités de données sont de taille variable, il est possible d'utiliser du padding pour obtenir des unités de taille constante. Les données à coder étant des vecteurs de bits, on utilisera un corps fini à 2^n éléments afin de permettre un meilleur alignement entre les éléments du corps fini et les données.

2.3.1.2 Implémentation de l'arithmétique des corps finis

On distingue plusieurs classes de corps finis. Les corps premiers, peuvent être représentés par $\mathbb{F}_p$ avec p premier. L'addition et la multiplication se font modulo p. Cependant, il est préférable d'utiliser un corps fini avec un nombre d'éléments étant une puissance de 2. En effet, les données en machine étant manipulées par bloc de n bits, il est préférable d'utiliser les corps binaires de caractéristique 2, notés $\mathbb{F}_{2^n}$. Dans ce cas, chaque bloc mémoire de n bits représentera un symbole.

La représentation polynomiale

En utilisant une base polynomiale binaire, un élément de $\mathbb{F}_{2^n}$ peut être vu comme un polynome $a(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_1x + a_0$ de degré $n-1$ dont les coefficients sont dans $\mathbb{F}_2$. On représentera ce même élément comme un vecteur de n bits : $(a_{n-1}, a_n, \ldots, a_1, a_0)$. Dans cette base, les opérations d'addition peuvent être implémentées par un simple *xor*. En revanche, l'opération de multiplication n'est pas directe. Soit $a(x)$ et $b(x)$, 2 polynomes représentant 2 éléments de $F = \mathbb{F}_{2^n} = \mathbb{F}_2[X]/(p(x))$, avec $p(x)$ un polynome irreductible de degré n.

$$a(x) *_F b(x) = a(x) * b(x) \mod p(x)$$

Plusieurs implémentations existent afin de réaliser cette opération de multiplication et il sera préférable de privilégier un polynome irreductible de poids faible afin de réduire le nombre d'opérations dans le calcul du modulo. Néammoins, il est également possible de réaliser cette opération en utilisant des tables précalculées. Par exemple, pour le corps fini $\mathbb{F}_{2^8}$, un élément du corps sera représenté par un octet de donnée, ce qui s'avère fort pratique. Ainsi, la table complète stockant toutes les multiplication possibles peut être stockée en mémoire dans 2^{16} octets, soit $64Ko$. Luigi Rizzo propose une implémentation basée cette méthode [48]. La bibliothèque Openfec [6] implémente également un codec utilisant cette solution. Avec une telle méthode, le nombre d'opérations est constant pour tout élément et pour tout polynome irréductible : la multiplication de deux éléments se traduira par un accès à une table de correspondance. Il en sera de même pour inverser un élément en utilisant la même technique pour précalculer la table des inverses des 2^n éléments du corps.

La représentation exponentielle

Lorsque le corps fini devient grand, les tables précalculées pour l'opération de la multiplication peut s'avérer problématique. Prenons le corps $F = \mathbb{F}_{2^{16}}$. Dans ce cas, la table des multiplications sera de $2^{16} * 2^{16} = 2^{32}$ éléments de 16 bits. Il convient alors d'utiliser un polynome irréductible *primitif*. Dans ce cas, l'élément x est un générateur du groupe multiplicatif F^*. On peut donc exprimer chaque élément du corps comme une puissance de α. La multiplication sera alors une simple addition des exposants et une lecture dans la table précalculée des logarithmes discrets de

chaque élément. Soient a et b deux éléments de $\mathbb{F}_2[X]/(p(x))$:

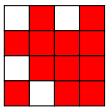
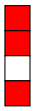
$$a *_F b = \alpha^i * \alpha^j = \alpha^{i+j}$$

Il suffira de précalculer une table des logarithmes, une seconde pour les exposants et troisième pour les inverses. Chacune des tables ayant 2^n entrées, soit $128Ko$ par table pour $\mathbb{F}_{2^{16}}$.

La représentation xor-based

Une autre manière de représenter un élément $a = a(x) = \sum_{i=0}^{w-1} a_i x^i$ d'un corps fini à 2^w éléments est d'utiliser une matrice $w * w$ à coefficients dans $\mathbb{F}_2$ dont la colonne i contient la représentation binaire de ax_i [14]. Ainsi, la multiplication peut s'exprimer en une série de *xor*.

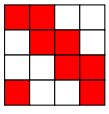
Soit le corps $F = \mathbb{F}_{2^4} = \mathbb{F}_2[X]/(x^4 + x + 1)$ et deux éléments de celui-ci : $c(x) = x + x^3$ et $a(x) = 1 + x^2 + x^3$. La multiplication $a(x) * c(x)$ dans F, donnant comme résultat $b(x) = 1 + x + x^3$ peut se représenter de la manière suivante :

$c(x) * a(x) =$  $*$  $=$  $= b(x)$

Nous avons ici la colonne 0 qui contient donc la représentation binaire de $a(x) \cdot x^0$, soit $a(x)$. La seconde colonne contiendra $a(x).x$ et ainsi de suite. Toutes les opérations sont faites modulo le polynome irréductible définissant le corps fini utilisé.

Notons que la représentation en matrice binaire des éléments du corps dépend directement du polynome irreductible utilisé. Deux corps isomorphes définis par un polynome générateur différent représenteront les éléments par des matrices binaires différences. Cette méthode, à la différence de la précédente qui fonctionne par accès tableau, donne un nombre d'opérations variables en fonction de l'élément représenté, même si en moyenne, ce nombre d'opérations est constant.

Enfin, cette représentation est permise quel que soit le polynome par lequel on calcule le modulo. C'est notamment le cas lorsque le polynome n'est pas irréductible. Dans ce cas, on travaille dans un anneau et non pas un corps. Elle s'applique à tout anneau polynomial à coefficient dans $\mathbb{F}_2$. Ainsi, en prenant un anneau polynomial $R = \mathbb{R}_{2^{16}} = \mathbb{F}_2[X]/(x^4 + 1)$, on peut représenter les éléments de celui-ci de la même manière.

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

Un anneau polynomial à 2^n éléments défini par un polynome de la forme $x^n + 1$ donnera des matrices binaires où on peut voir apparaitre des diagonales qui pourront être implémentées de manière directe en machine. Cependant, dans cet anneau, certains éléments n'ont pas d'inverse. On ne peut malheureusement pas travailler directement avec cette structure pour les codes correcteurs.

La représentation polynomiale par la methode *split-table*

C'est l'implémentation qui fait référence aujourd'hui. Elle a été présentée pour la première fois dans [30], puis dans [9] et [41]. Cette méthode exploite indirectement la propriété distributive de la multiplication dans un corps fini : $(a+b)*c = a*c+b*c$ avec a, b, c des éléments d'un corps fini. Soit $a_i^g(x)$ un polynome de degré inférieur à g, alors, de manière générale, on peut représenter un élément d'un corps fini à 2^n éléments $a(x)$ comme il suit :

$$a(x) = \sum_{i=0}^{\lceil \frac{n}{g} \rceil - 1} a_i(x) x^{gi}$$

Pour multiplier deux polynomes $a(x)$ et $b(x)$ représentant deux éléments d'un corps fini, on utilisera cette propriété pour avoir :

$$\begin{aligned} a(x) * b(x) &= \left(\sum_{i=0}^{\lceil \frac{n}{g_a} \rceil - 1} a_i(x) x^{g_a i} \right) \left(\sum_{j=0}^{\lceil \frac{n}{g_b} \rceil - 1} b_j(x) x^{g_b j} \right) \\ &= \sum_{i=0}^{\lceil \frac{n}{g_a} \rceil - 1} \sum_{j=0}^{\lceil \frac{n}{g_b} \rceil - 1} a_i(x) b_j(x) x^{g_a i + g_b j} \end{aligned}$$

En choisissant correctement g_a et g_b, il est possible de tirer parti de certaines instructions SIMD [1] pour effectuer plusieurs multiplications en parallèle à l'aide de tables de correspondance préalablement calculées. Afin de bien comprendre les différences entre les travaux présentés et les implémentations utilisées actuellement, nous allons détailler l'implémentation Split-table avec deux corps finis : $\mathbb{F}_{2^4}$ et $\mathbb{F}_{2^8}$. Il s'agit à ce jour de la méthode la plus performante pour effectuer des opérations dans un corps fini sur des processeurs modernes. Elle est détaillée dans [40].

Le cas $\mathbb{F}_{2^4}$

Dans le corps $\mathbb{F}_{2^4}$, les 16 éléments sont représentés par des mots de 4 bits. Chaque octet contient 2 éléments, et nécessite donc deux accès à une table de look-up pour la multiplication. Cela représente donc plusieurs milliers d'accès mémoires pour encoder un vecteur de quelques centaines d'octets. Pour éviter cela, il est possible d'utiliser des instructions SIMD permettant de manipuler des registres de grande taille (128 bits, 256 ou 512 bits). Par exemple, avec SSE, l'instruction *pshufb*, abréviation de *Packed Shuffle Bytes* ([1]) permettra de paralléliser la multiplication d'un vecteur de 16 octets, soit 32 éléments du corps, par un élément constant en quelques instructions : soient deux registres de 128 bits ou 16 octets, dont chaque octet qui les composent contiennent des mots de 4bits représentant un élément du corps fini : $A = [a_0, a_1, \ldots, a_{15}]$ et $B = [b_0, b_1, \ldots, b_{15}]$. En appliquant l'instruction *pshufb* sur ces deux registres, le premier sera mis à jour de la manière suivante : $A = [a_{b_0}, a_{b_1}, \ldots, a_{b_{15}}]$.

La Figure 2.14 montre les différentes instructions SSE, présentes sur les processeurs Intel depuis 2006 pour celles qui nous intéressent, à utiliser pour multiplier en parallèle 128 bits de données par un élément sur $\mathbb{F}_{2^4}$. On constate un éclatement en deux du vecteur 128 bits b à l'aide d'un masque, puis on effectue la multiplication à l'aide de tables de correspondances et de l'instruction *pshufb* (mm_shuffle_epi8) sur les deux vecteurs *low* et *high* avant de les sommer entre eux dans *product*.

byte	f	e	d	c	b	a	9	8	7	6	5	4	3	2	1	0
b:	39	1d	9f	5a	aa	ab	15	c3	63	e0	7c	43	fb	83	16	23
table1:	0b	0c	05	02	04	03	0a	0d	06	01	08	0f	09	0e	07	00
table2:	b0	c0	50	20	40	30	a0	d0	60	10	80	f0	90	e0	70	00
mask1:	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f	0f
mask2:	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0	f0
low = **mm_and_si128***(b, mask1)*:	09	0d	0f	0a	0a	0b	05	03	03	00	0c	03	0b	03	06	03
low = **mm_shuffle_epi8***(table1, low)*:	0a	05	0b	03	03	04	08	09	09	00	02	09	04	09	01	09
high = **mm_and_si128***(b, mask2)*:	30	10	90	50	a0	a0	10	c0	60	e0	70	40	f0	80	10	20
high = **mm_srli_epi64***(high, 4)*:	03	01	09	05	0a	0a	01	0c	06	0e	07	04	0f	08	01	02
high = **mm_shuffle_epi8***(table2, high)*:	90	70	a0	80	30	30	70	20	10	c0	60	f0	b0	d0	70	e0
product = **mm_xor_si128***(high, low)*:	9a	75	ab	83	33	34	78	29	19	c0	62	f9	b4	d9	71	e9

FIGURE 2.14 – Instructions SSE utilisées pour effectuer 32 multiplications simultanées dans $\mathbb{F}_{2^4}$ (extrait de [40]).

Le cas $\mathbb{F}_{2^8}$

Dans le cas d'un corps fini à 2^8 éléments, il est possible de remplacer la multiplication à l'aide d'une table de correspondance de $256 * 256$ par 2 multiplications à l'aide de deux tables de correspondances de $256 * 16$ éléments et une somme (dans $\mathbb{F}_2$). La méthode est exactement la même que pour $\mathbb{F}_{2^4}$, mais en utilisant des tables contenant un plus grand nombre d'éléments. A ce jour, l'implémentation la plus efficace utilisant cette méthode est le code d'Intel : ISA-L [3]. Il s'agit d'un code entièrement écrit en langage machine, spécifiquement pour les processeurs Intel. Ce code servira de référence dans la suite de ce manuscrit.

2.3.2 Réduction de la complexité de codage

Afin de réduire la complexité des codes correcteurs basés sur les corps finis, en plus d'utiliser des implémentations efficaces, de nombreux travaux ont permis de diminuer le nombre d'opérations définies par la matrice génératrice ou de les réordonner. L'utilisation de la représentation *xor-based* permet de réaliser cela en travaillant directement sur des matrices binaires.

2.3.2.1 Réduction du nombre d'opérations

En 1999, Blaum et Roth donnent une limite théorique du nombre minimal d'éléments différents de l'élément nul que peut contenir une matrice génératrice pour

conserver sa propritété MDS et donne quelques constructions pour certains paramètres [12]. Pour un code (n, k), en travaillant sur $\mathbb{F}_2^w$, la matrice génératrice, pour être MDS, doit contenir au minimum $k * r * w$ éléments non nuls, soit $(k-1) * r * w$ opérations.

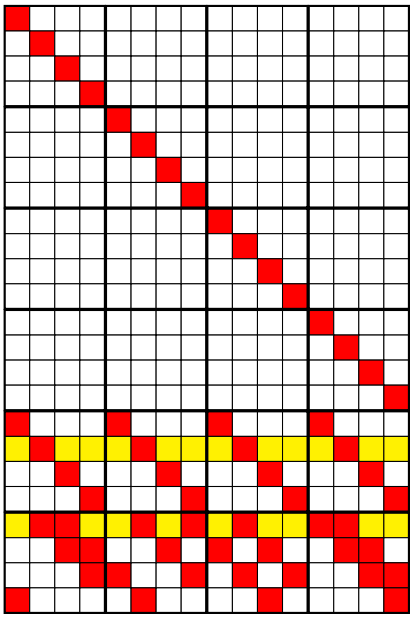


FIGURE 2.15 – Matrice génératrice d'un code RDP $(6, 4)$

Même si les codes de Reed-Solomon n'atteignent pas cette limite, d'autres codes spécifiques à certains paramètres comme le nombre de symboles de redondance maximum que l'on peut générer ont été présentés. Par exemple, les codes EVENODD [13] ou les codes RDP (Row Diagonal Parity) [17] permettent de corriger seulement deux effacements. Ils ont été pensés pour être utilisés dans les systèmes de type RAID-6 (redundant array of independent disks), mais peuvent être utilisés dans n'importe quel autre cas d'usage, à condition qu'il n'y ait pas plus de deux pertes par bloc. Les codes RDP doivent avoir comme contrainte $w + 1$ premier et $k \leq w$ et atteignent des performances de codage optimales, soit $k - 1$ *xor* par symbole de redondance, lorsque $k = w$ ou $k + 1 = w$. Ils réduisent la complexité de calcul mais ne sont pas aussi génériques que les code Cauchy Reed-Solomon. Par exemple, la Figure 3.2 représente la matrice génératrice d'un code RDP $(6, 4)$. Dans ce cas, chaque symbole est représenté par 4 bits, et chaque bit de redondance est une combinaison d'en moyenne 5.25 bits sources, soit 4.25 *xor*. Cependant, certaines équations peuvent être factorisées, par exemple celles en jaune dans la Figure 3.2 :

$$c_{0,1} = d_{0,1} + d_{1,1} + d_{2,1} + d_{3,1}$$
$$c_{1,0} = d_{0,1} + d_{0,2} + d_{1,1} + d_{1,3} + d_{2,1} + d_{3,0} + d_{3,1}$$

On peut constater que $c_{1,0}$ est calculé à partir de $c_{0,1}$, ainsi, nous avons :

$$c_{0,1} = d_{0,1} + d_{1,1} + d_{2,1} + d_{3,1}$$
$$c_{1,0} = c_{0,1} + d_{0,2} + d_{1,3} + d_{3,0}$$

Ainsi, pour le code RDP $(6, 4)$, après étape de factorisation, le nombre total de *xor* nécessaire par bit de redondance passe de 33 à 24, soit $(k-1) * r$, ce qui est moins élevé que la limite que Blaum et Roth ont définie.

En 2011, Plank a présenté une méthode qui permet à un codeur de descendre en dessous des $k-1$ *xor* par symbole de redondance [39]. En effet, en testant des algorithmes de factorisation sur plusieurs matrices, il apparait que certaines d'entre elles permettent d'obtenir moins de $k-1$ *xor* par symbole de redondance.

Cependant, tous ces travaux permettent d'optimiser les opérations de codage. En effet, étant donné que les opérations à réaliser lors du décodage dépendent du modèle de perte, il est difficile de prévoir tous les cas possibles et d'optimiser toutes les matrices de décodage [25].

2.3.2.2 Réduction de la complexité à l'aide de transformées

Afin de réduire la complexité dans les codes à effacement basés sur les corps finis, il est également possible de modifier la manière dont les combinaisons sont faites. En effet, les codes correcteurs d'effacement ont par défaut une complexité quadratique en $\mathcal{O}((n-k)k)$: on génère $n-k$ combinaisons linéaires de k symboles. Mais il est possible d'utiliser des algorithmes dont la complexité est sous-quadratique. L'un des premiers travaux à utiliser ce genre d'algorithme utilise la transformée de Walsh et permet de coder et décoder avec une complexité en $\mathcal{O}(q \log^2 q)$, avec $q = 2^r$ la taille du corps fini utilisé. Dans [54], les auteurs proposent un code de Reed-Solomon avec comme prérequis l'utilisation d'un corps fini $\mathbb{F}_p$ où p est un nombre de Fermat premier (de la forme $p = 2^{2^k} + 1$). Outre le fait que les opérations d'addition et de multiplication dans $\mathbb{Z}/p\mathbb{Z}$ s'exécutent différemment d'un corps fini de cardinalité 2 en raison de l'arithmétique des entiers utilisée, l'intérêt ici est de pouvoir utiliser la transformée de Fourier rapide (FFT) dans un corps fini dont la complexité est $\mathcal{O}(n \log n)$. Dans ces corps finis, cette transformée est appelée Transformée en Nombre de Fermat (FNT) et n'est possible que si le corps fini est un corps premier. En effet, pour être implémentée de manière efficace, la tranformée FNT nécessite de travailler avec un corps fini possédant un élément d'ordre pair, soit $\mathbb{F}_p$ où $p = 2^{2^k} + 1$. Dans [32], les auteurs améliorent cette méthode pour les cas où $n/2 \leq k < n$ et celle-ci semble être une bonne solution pour les processeurs

ne possédant d'instructions SIMD. Concernant les processeurs modernes équipés d'instructions SIMD, il sera préférable de travailler sur $\mathbb{F}_{2^n}$ avec $n = 4, 8, 16, \ldots$ pour avoir de meilleures performances.

Récemment, des travaux ont présenté un algorithme de transformée de Fourrier s'utilisant sur un corps fini de caractéristique 2 [31] où les polynomes sont réprésentés dans une base non-standard [16]. Ainsi, il est possible d'obtenir une implémentation utilisant des instructions SIMD avec une complexité sous quadratique en $\mathcal{O}(n \log n)$. A ce jour, la seule implémentation publique connue est Leopard-RS [5].

Chapitre 3

Transformées entre un corps fini et un anneau

Sommaire

3.1 Introduction

Dans ce chapitre, nous présentons une méthode permettant de réduire la complexité de la multiplication dans un corps fini. Comme nous l'avons vu, la plupart des codes à effacement utilisent les corps finis et l'opération de multiplication matrice vecteur est présente dans les processus de codage et de décodage. Afin d'être le plus performant possible, les codes à effacement utilisent très généralement des corps finis de caractéristique 2, de manière à représenter l'addition par un *xor*.

Concernant la multiplication, deux approches sont possibles pour effectuer cette opération. La première est de décomposer cette opération en un certain nombre de *xor* [12]. Cela n'est faisable que si la caractéristique du corps fini utilisé est 2. La seconde approche consiste à utiliser des tables précalculées. Le résultat de la multiplication de deux éléments sera directement lu depuis la mémoire. Cette méthode a l'avantage de fonctionner sur tous les corps finis, quel que soit leur caractéristique mais présente l'inconvénient de devoir calculer, stocker et accéder à une grande quantité de données. Grâce aux instructions SIMD [1] des processeurs, l'utilisation des tables précalculées pour effectuer de manière parallèle plusieurs multiplications dans un corps fini est aujourd'hui la méthode la plus rapide [41].
Indépendamment, dans le contexte des applications cryptographiques et pour implémenter en matériel efficacement la multiplication dans un grand corps fini, Itoh et Tsujii [26] proposent une méthode pour effectuer ces multiplications dans un anneau polynomial, facilitant ainsi l'opération du modulo. En effet, dans cet anneau, les opérations sur les polynomes sont faites modulo $x^n + 1$, contrairement aux corps finis où les opérations sont faites modulo un polynome irréductible. La multiplication par un monome est donc bien plus facile car l'opération modulo peut s'effectuer par un simple décalage cyclique. Il s'agit donc d'une implémentation matérielle de type *xor-based*.

Nous proposons d'étendre et d'appliquer cette approche aux codes correcteurs en définissant les éléments nécessaires à une implémentation complète pour les codes à effacement et nous montrerons pourquoi il est possible d'effectuer la multiplication dans un anneau. Nous détaillerons plusieurs transformées qui nous permettent de passer d'un élément du corps fini à un élément de l'anneau, puis de revenir. Nous montrerons que ces transformées possèdent des propriétés différentes qui peuvent être exploitées afin de réduire grandement la complexité des codes à effacement. Enfin, nous analyserons la complexité des processus de codage et de décodage pour les codes MDS et nous donnerons quelques constructions permettant dans certains cas d'obtenir une complexité très proche de la limite donnée dans [12].

3.2 Contexte algébrique

Dans un code à effacement, l'opération de multiplication matrice vecteur est présente au codage et au décodage entre les données représentées sous la forme de vecteurs d'éléments du corps et les coefficients d'une matrice, également éléments du corps. Afin de pouvoir réaliser cette opération dans un anneau, il nous faut donc transformer les données à coder et les coefficients. Une fois ces transformations effectuées, nous allons pouvoir appliquer la multiplication. Le résultat sera alors un vecteur d'éléments de l'anneau. Il faudra alors revenir depuis ce vecteur d'éléments de l'anneau à un vecteur d'éléments du corps en appliquant une transformée inverse. Travailler avec des éléments d'un anneau présente plusieurs avantages. D'abord, le

nombre d'opérations nécessaires à la multiplication peut être réduit. En effet, dans le chapitre précédent, nous avons vu que le polynome irréductible définissant le corps fini influe sur le nombre d'opérations *xor* à faire pour effectuer une multiplication. Grâce à la structure cyclique de l'anneau, les élements de la matrice génératrice binaire pourront être représentés sous la forme de simples vecteurs au lieu de matrices carrées. Cela permettra une simplicité dans le déroulement des opérations par le processeur, et donc un gain de performance. Cet avantage sera développé dans le prochain chapitre.

3.2.1 L'isomorphisme entre un corps fini et un idéal

Nous allons maintenant définir les outils algébriques qui vont nous permettre de passer d'un corps fini à un anneau et vice-versa.

Définition 13. *Soit $\mathbb{F}_{q^w}$ un corps fini avec q^w éléments.*

Définition 14. *Soit $R_{q,n} = \mathbb{F}_q[x]/(x^n - 1)$ l'anneau quotient des polynomes de l'anneau polynomial $\mathbb{F}_q[x]$ quotienté par l'idéal généré par le polynome $x^n - 1$.*

Définition 15. *Soit $p_1^{u_1}(x)p_2^{u_2}(x)\dots p_r^{u_r}(x) = x^n - 1$ la décomposition de $x^n - 1$ en polynomes irreductibles sur $\mathbb{F}_q$.*

Lorsque n et q sont premiers entre eux, il peut être démontré que $u_1 = u_2 = \dots = u_n = 1$ ([42]). En d'autres termes, si $q = 2$, et n est impair, nous avons $p_1(x)p_2(x)\dots p_r(x) = x^n - 1$.

Dans la suite de ce manuscrit, nous considérons que n et q sont premiers entre eux.

Proposition 1. *[35] L'anneau $R_{q,n}$ est égal à la somme directe de ses r idéaux minimaux de $A_i = ((x^n - 1)/p_i(x))$ pour $i = 1, \dots, r$.*

De plus, chaque idéal minimal A_i contient un unique idempotent $\theta_i(x)$. Une construction de cet idempotent est donnée dans [35], Chap. 8, Théoreme 6.
Comme $\mathbb{F}_q[x]/(p_i(x))$ est isomorphe au corps fini $B_i = \mathbb{F}_{q_i^w}$, où $p_i(x)$ est de degré w_i, nous avons :

Proposition 2. *[35] $R_{q,n}$ est isomorphe au produit Cartésien suivant :*

$$R_{q,n} \simeq B_1 \times B_2 \times \dots B_r$$

Pour chaque $i = 1, \dots, r$, A_i est isomorphe à B_i. L'isomorphisme est défini par :

$$\phi_i : \begin{array}{ccc} B_i & \rightarrow & A_i \\ b(x) & \rightarrow & b(x)\theta_i(x) \end{array} \tag{3.1}$$

et l'isomorphisme inverse est défini par :

$$\phi_i^{-1} : \begin{array}{ccc} A_i & \rightarrow & B_i \\ a(x) & \rightarrow & a(\alpha_i) \end{array} \tag{3.2}$$

où α_i est une racine de $p_i(x)$.

Prenons $q = 2$ et rappelons les deux classes de polynomes à coefficients dans $\mathbb{F}_2$ définies dans [26].

3.2.2 Les polynomes AOP

Définition 16. *Un polynome AOP (All One Polynomial) est un polynome plein de 1 et de degré w défini comme il suit :*

$$p(x) = x^w + x^{w-1} + x^{w-2} + \ldots + x + 1$$

Proposition 3. *Un polynome AOP $p(x)$ de degré w est irréductible sur $\mathbb{F}_2$ si et seulement si $w+1$ est premier et w génère $\mathbb{F}^*_{w+1}$, où $\mathbb{F}^*_{w+1}$ est le groupe multiplicatif dans $\mathbb{F}_{w+1}$ [57].*

Les valeurs de $w + 1$, telles que le polynome AOP de degré w est irréductible sont données dans la séquence A001122 de [53]. Nous considèrerons uniquement les polynomes AOP irréductibles.

D'après la proposition 2, $R_{2,w+1}$ est égal à la somme directe de ses idéaux principaux $A_1 = ((x^{w+1}+1)/p(x)) = (x+1)$ et $A_2 = ((x^{w+1}+1)/(x+1)) = (p(x))$, et $R_{2,w+1}$ est isomorphe au produit direct de $B_1 = \mathbb{F}_2[x]/(p(x)) = \mathbb{F}_{2^w}$ et $B_1 = \mathbb{F}_2[x]/(x+1) = \mathbb{F}_2$.

Proposition 4. *L'idempotent primitif θ_1 de l'idéal minimal A_1 dans l'anneau $R = \mathbb{F}_2[x]/(x^{w+1}+1)$ est égal à $p(x)+1$.*

Démonstration. Nous avons :

$$\begin{aligned} p(x) \cdot x &= (x^w + x^{w-1} + x^{w-2} + \ldots + x + 1).x \\ &= x^{w+1} + x^w + \ldots + x^2 + x \\ &= 1 + x^w + \ldots + x^2 + x \\ &= p(x) \end{aligned}$$

De la même manière, on a :

$$\begin{aligned} p(x) \cdot x \cdot x &= p(x) \cdot x \\ &= p(x) \end{aligned}$$

D'après la Proposition3, si w représente le degré du polynome AOP irréductible,

alors $w+1$ est premier, donc impair. On peut donc écrire :

$$\begin{aligned} p(x) \cdot p(x) &= p(x) \cdot (x^w + x^{w-1} + x^{w-2} + \ldots + x + 1) \\ &= p(x) \cdot x^w + p(x) \cdot x^{w-1} + \ldots + p(x) \cdot x + p(x) \\ &= \sum_{1}^{w+1} p(x) \\ &= p(x) \end{aligned}$$

Donc, $(p(x)+1) \cdot (p(x)+1) = p(x)^2 + 1 = p(x) + 1$.

De plus, comme $p(x)$ est une somme de $w+1$ termes, $p(x)+1$ est une somme de w termes. Comme w est pair, nous avons donc :

$$\begin{aligned} p(x) + 1 &= x^w + x^{w-1} + \ldots + x^2 + x + 1 + 1 \\ &= x^w + x^{w-1} + \ldots + x^2 + x \\ &= x.(x^{w-1}) + x^{w-1} + \ldots + x.x + x \\ &= (x+1).(x^{w-1}) + \ldots + (x+1).x \end{aligned}$$

Par conséquent, $p(x)+1$ est un multiple du générateur $(x+1)$ de A_1 et donc $p(x)+1$ appartient à A_1.

De plus, comme $(p(x)+1) \cdot (p(x)+1) = p(x)+1$, $p(x)+1$ est un idempotent de A_1.
Enfin, A_1 étant un idéal minimal de l'anneau, $p(x)+1$ est donc l'idempotent primitif de A_1. □

Cet idempotent est utilisé pour construire l'isomorphisme ϕ_1 entre A_1 et B_1. Le fait d'appartenir à un idéal va permettre d'appliquer des opérations et de rester dans cet idéal. Ainsi, l'isomorphisme inverse restera possible après ces opérations.

3.2.3 Les polynomes ESP

Définition 17. *Un polynome $g(x) = x^{sr} + x^{s(r-1)} + x^{s(r-2)} + \ldots + x^s + 1 = p(x^s)$, où $p(x)$ est de type AOP de degré r, est appelé "s-equally spaced polynomial" (s-ESP) de degré sr.*

Proposition 5. *[26] $g(x)$ est irréductible si et seulement si $p(x)$ est irréductible et avec t un entier, $s = (r+1)^{t-1}$ et $2^{r(r+1)^{t-2}} \neq 1$ mod $(r+1)^t$.*

Les premières valeurs des couples (r, s) pour lesquelles le polynome de type ESP est irréductible sont $(2,3), (2,9), (4,5), (2,27), \ldots$

Les polynomes ESP $g(x) = x^{sr} + x^{s(r-1)} + \ldots + x^s + 1$ divisent les polynomes $x^{s\cdot(r+1)} + 1$ car $x^{s\cdot(r+1)} + 1 = g(x) \cdot (x^s + 1)$. De plus, d'après la Proposition 1, si le

polynome ESP $g(x)$ est irréductible, le corps $\mathbb{F}_2[x]/(g(x)) = \mathbb{F}_{2^{r\cdot s}}$ est isomorphe à l'idéal A_1 généré par $(x^{s\cdot(r+1)} - 1)/g(x) = x^s + 1$.

Proposition 6. *L'idempotent primitif $\theta_1(x)$ de l'idéal minimal A_1 dans l'anneau $R = \mathbb{F}_2[x]/(x^{s\cdot(r+1)} + 1)$ est égal à $g(x) + 1$.*

Démonstration. Le polynome $g(x) = x^{sr} + x^{s(r-1)} + \ldots + x^s + 1$ est une somme de $r + 1$ termes.
Nous avons :

$$\begin{aligned} g(x) \cdot x^s &= (x^{sr} + x^{s(r-1)} + \ldots + x^s + 1) \cdot x^s \\ &= x^{s\cdot(r+1)} + x^{s\cdot r} + x^{s\cdot(r-1)} + \ldots + x^s \\ &= 1 + x^{s\cdot r} + x^{s\cdot(r-1)} + \ldots + x^s \\ &= g(x) \end{aligned}$$

De la même manière, on a :

$$\begin{aligned} g(x) \cdot x^s \cdot x^s &= g(x) \cdot x^s \\ &= g(x) \end{aligned}$$

Donc, si $g(x)$ est une somme de $r + 1$ termes, on peut écrire :

$$\begin{aligned} g(x) \cdot g(x) &= g(x) \cdot (x^{sr} + x^{s(r-1)} + \ldots + x^s + 1) \\ &= g(x) \cdot x^{sr} + g(x) \cdot x^{s\cdot(r-1)} + \ldots + g(x) \cdot x^s + g(x) \\ &= \sum_{1}^{r+1} g(x) \\ &= g(x) \end{aligned}$$

Dans le cas d'un polynome ESP irréductible de paramètres (r, s), r est le degré définissant le polynome AOP irréductible. Comme nous l'avons vu, un polynome AOP de degré r est irréductible si $r + 1$ est premier, et donc si r est pair.
Nous avons donc $g(x) \cdot g(x)$ qui est égal à une somme impaire de $g(x)$.
Par conséquent, $(g(x) + 1) \cdot (g(x) + 1) = g(x) \cdot g(x) + g(x) + g(x) + 1 = g(x) + 1$.

De plus, comme $g(x)$ est une somme de $r + 1$ termes, $g(x) + 1$ est une somme de r termes. Nous avons donc :

$$\begin{aligned} g(x) + 1 &= x^{s\cdot r} + x^{s(r-1)} + \ldots + x^{2s} + x^s \\ &= (x^s + 1) \cdot (x^{(r-1)\cdot s}) + \ldots + (x^s + 1) \cdot x^s \\ &= (x^s + 1) \cdot (x^{(r-1)\cdot s}) + \ldots + x^s) \end{aligned}$$

Par conséquent, $g(x) + 1$ est un multiple du générateur $(x^s + 1)$ de A_1 et donc $g(x) + 1$ appartient à A_1.

De plus, comme $(g(x)+1)\cdot(g(x)+1) = g(x)+1$, $g(x)+1$ est un idempotent de A_1.
Enfin, A_1 étant un idéal minimal de l'anneau, $g(x)+1$ est donc l'idempotent primitif de A_1.

□

3.3 Les différentes transformées

Dans leurs travaux, Itoh et Tsujii [26] proposent d'utiliser une seule et même transformée pour les données et les coefficients de la matrice. Or, comme nous l'avons rappelé, les coefficients de la matrice définissent les opérations à effectuer et représentent une quantité assez faible de données à transformer, alors que les vecteurs représentant les données peuvent être de très grande taille. Ces derniers doivent être transformés le plus rapidement possible. Des transformées aux propriétés différentes devraient donc être utilisées pour un obtenir code performant.

$$\begin{array}{lccc}
\mathbb{F}_{2^w}: & (\alpha_0,\ldots,\alpha_{k-1}) & \times \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix} & = (\beta_0,\ldots,\beta_{n-1}) \\
& \Downarrow & \Downarrow & \Uparrow \\
R_{2,n}: & (a_0,\ldots,a_{k-1}) & \times \begin{pmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} \end{pmatrix} & = (b_0,\ldots,b_{n-1})
\end{array}$$

Nous allons maintenant décrire plusieurs transformées utilisées entre le corps fini $B_1 = \mathbb{F}_{2^w} = \mathbb{F}_2[x]/(p(x))$ et l'anneau $R_{2,n} = \mathbb{F}_2[x]/(x^n+1)$. Chacune d'elles possède des propriétés différentes qui vont nous permetttre de les utiliser à différents endroits dans un code à effacement. Nous commencerons par décrire la transformée isomorphique basique, puis celle utilisée par Itoh, puis nous continuerons avec celles qui ont été définies dans ces travaux permettant d'obtenir une vitesse de codage performante.

3.3.1 La transformée isomorphique

La première méthode pour transformer un élément $b_B(x)$ d'un corps B_1 en un élément $b_A(x)$ d'un idéal A_1 est la simple application de l'isomorphisme entre B_1 et l'idéal A_1 de $R_{2,n}$ (voir la Proposition 2).

3.3.1.1 Cas général

Comme nous l'avons défini précédemment, les isomorphismes et leurs inverses sont définies comme il suit :

$$\phi_i(b_B(x)) = b_B(x).\theta_i(x) = b_A(x)$$

$$\phi_i^{-1}(b_A(x)) = b_A(\alpha_i).$$

En suivant la définition de l'isomorphisme, nous avons :

$$\phi_i^{-1}(\phi_i(u(x)).\phi_i(v(x))) = u(x).v(x)$$

3.3.1.2 Cas avec un polynome irréductible AOP

Grâce à la structure particulière des polynomes AOP, l'isomorphisme admet une version simplifiée dans ce cas.

Soit $W(b_B(x))$, le poids de Hamming de $b_B(x)$ un élément de B_1, défini comme étant le nombre de monomes dans la représentation polynomiale de $b_B(x)$.

Proposition 7.

$$\phi_1(b_B(x)) = b_A(x) = \begin{cases} b_B(x) & si\ W(b_B(x))\ est\ pair \\ b_B(x) + p(x) & sinon \end{cases}$$

Démonstration. Nous avons $\phi_1(b(x)) = b(x)\theta_1(x) = b(x)(p(x)+1) = b(x)p(x)+b(x)$. Nous pouvons voir que $b(x)p(x) = 0$ si $W(b(x))$ est pair et $b(x)p(x) = p(x)$ si $W(b(x))$ est impair. Une conséquence intéressante de cette proposition est que toutes les images par ϕ_1 sont de poids pair. □

Proposition 8.

$$\phi_1^{-1}(b_A(x)) = b_B(x) = \begin{cases} b_A(x) & si\ b_w = 0 \\ b_A(x) + p(x) & sinon \end{cases}$$

où b_w est le coefficient du monome de degré w de $b_A(x)$.

Démonstration. D'après la Proposition 7, on peut voir que si un élément de A_1 possède un coefficient $b_w \neq 0$, alors il a été nécessairement obtenu depuis la seconde règle, c'est à dire en ajoutant $p(x)$. Donc, son image dans B_1 peut être obtenue en soustrayant (ou ajoutant en binaire) $p(x)$. Si $b_w = 0$, alors $b_B(x)$ est obtenu directement. □

3.3.1.3 Cas avec un polynome ESP

De la même manière qu'un polynome de type AOP, nous pouvons simplifier le cas général pour un polynome de type ESP. D'après la Proposition 6, l'isomorphisme

entre le corps $B_1 = \mathbb{F}_2[x]/(g(x)) = \mathbb{F}_{2^w}$ et l'idéal A_1 de $R_{2,n}$ est défini à l'aide de l'idempotent primitif $\theta_1(x) = g(x) + 1$ de A_1 :

$$\phi_1 : \begin{array}{ccc} B_1 & \rightarrow & A_1 \\ b_B(x) & \mapsto & b_A(x) = b_B(x) \cdot (g(x) + 1) \end{array} \tag{3.3}$$

l'isomorphisme inverse est défini par :

$$\phi_1^{-1} : \begin{array}{ccc} A_1 & \rightarrow & B_1 \\ b_A(x) & \mapsto & b_A(x) \bmod g(x) \end{array} \tag{3.4}$$

Proposition 9. *Les éléments de l'idéal A_1 sont composés d'un entrelacement de s mots de poids pair de longueur $r + 1$.*

Démonstration. D'après la Proposition 6, tous les éléments de A_1 sont multiples du polynome générateur $x^s + 1$. Donc, un élément $a(x)$ de la forme $u(x).(x^s + 1) = (\sum_{i=0}^{s\cdot(r+1)-1} u_i x^i) \cdot (x^s + 1)$. $u(x)$ peut également s'exprimer sous la forme $\sum_{j=0}^{s-1} x^j v_j(x^s)$ où $v_j(x)$ est un polynome de degré $r - 1$, pour $j = 0, \ldots, s - 1$. De plus, nous avons $u(x) \cdot (x^s + 1) = \sum_{j=0}^{s-1} x^j v_j(x^s) \cdot (x^s + 1) = \sum_{j=0}^{s-1} x^j v'_j(x^s)$ où $v'_j(x) = v_j(x) \cdot (x + 1)$ dans $R = \mathbb{F}_2[x]/(x^r + 1)$, pour $j = 0, \ldots, s - 1$. Cela implique que le poids de chaque v'_j est pair. Cela signifie que chaque élément de A_1 peut être vu comme l'entrelacement de s éléments de poids pair de longueur $r + 1$. Le nombre d'éléments vérifiant cette propriété est exactement le nombre d'éléments de A_1, donc cette propriété caractérise les éléments de A_1. □

3.3.2 La transformée Embedding

Soit ϕ_E la fonction *Embedding* qui consiste simplement à considérer l'élément du corps comme un élément de l'anneau sans aucune transformation. Cette fonction a été initialement proposée dans [26]. Toutefois, le cadre général de [26] est très différent du nôtre. Dans cette partie, nous présentons une justification de l'utilisation de cette transformée dans notre contexte.

3.3.2.1 Cas général

Il s'agit de transformer un élément $u(x)$ de B_i en un élément de $R_{2,n}$.

$$\phi_e : \begin{array}{ccc} B_i & \rightarrow & R_{2,n} \\ u(x) & \mapsto & u(x) \end{array} \tag{3.5}$$

Notons que les images des éléments de B_i n'appartiennent pas nécessairement à A_i. Cependant, définissons la fonction ϕ_i^{-1} de $R_{2,n}$ vers A_i par $\phi_i^{-1}(b_A(x)) = b_A(\alpha)$, où α est une racine de $p(x)$. Cette fonction peut être vue comme une extension de la fonction ϕ_i^{-1} dans l'anneau entier.

$$\phi_i^{-1} : \begin{array}{ccc} R_{2,n} & \rightarrow & B_i \\ u(x) & \mapsto & \phi_i^{-1}(u(x)) = u(\alpha) \end{array} \tag{3.6}$$

Proposition 10. *[26] Pour chaque $u(x)$ et $v(x)$ dans B_i, nous avons :*

$$\phi_i^{-1}(\phi_E(u(x)) \cdot \phi_E(v(x))) = u(x) \cdot v(x)$$

Démonstration. La fonction *Embedding* correspond à la multiplication par 1 dans l'anneau. En fait, 1 est égal à la somme des l idempotents primitifs $\theta_i(x)$ des l idéaux minimaux A_i, pour $i = 1, \ldots, l$ [35, chapter 8, thm. 7]. Ainsi, $\phi_E(u(x)) = u(x) \cdot \sum_{i=1}^{l} \theta_i(x)$. Donc, $\phi_E(u(x)) \cdot \phi_E(v(x))$ est égal à $u(x) \cdot v(x) \cdot (\sum_{i=0}^{l} \theta_i(x))^2$.Grâce aux propriétés des idempotents, $\theta_i(x) \cdot \theta_j(x)$ est égal à $\theta_i(x)$ si $i = j$ et 0 sinon. Nous avons donc, $\phi_E(u(x)) \cdot \phi_E(v(x))$ qui est égal à $u(x) \cdot v(x) \cdot (\sum_{i=1}^{l} \theta_i(x))$. La fonction ϕ_i^{-1} est le calcul du reste modulo $p_i(x)$. Le polynome irréductible $p_i(x)$ correspond à l'idéal A_i. Donc $\theta_i(x) \bmod p(x)$ est égal à 1 si $i = 1$, sinon 0. □

Cette proposition prouve que la fonction *Embedding* peut être utilisée pour effectuer la multiplication dans l'anneau au lieu de la faire dans le corps. La fonction isomorphe vue précédemment possède également cette propriété, mais les transformées entre le corps et l'anneau sont plus complexes.

3.3.2.2 Cas avec un polynome irreductible AOP

Soit $b_B(x)$ un élément de B_1. La transformée ϕ_e est directe, donne un élément de $R_{2,w+1}$ et ne nécessite aucun calcul :

$$\phi_e : \begin{array}{ccc} B_1 & \rightarrow & R_{2,w+1} \\ b_B(x) & \mapsto & b_R(x) \end{array} \tag{3.7}$$

En effet, si $p(x)$ est le polynome AOP irréductible de degré w, alors, l'anneau défini par le polynome $x^{w+1} + 1$ à coefficients dans $\mathbb{F}_2$ est isomorphe au produit cartésien de $p(x)$ et $(x + 1)$. La multiplication par 1 dans l'anneau revient donc à multiplier par la somme des 2 idempotents principaux définissant les 2 idéaux minimaux A_1 et A_2 de l'anneau.

En revanche, la transformée inverse nécessite un calcul :

$$\phi_e^{-1} : \begin{array}{ccccc} R_{2,n} & \rightarrow & & & B_1 \\ b_R(x) & \mapsto & b_R(x) & & \bmod p(x) \end{array} \tag{3.8}$$

Dans le cas d'un polynome AOP, le calcul du reste modulo $p(x)$ se calcule de la manière suivante : soit $b_R(x) = a_0.1 + a_1.x + \cdots + a_w.x^{w+1}$. On a $\phi_e^{-1}(b_R(x)) = (a_0 + a_w) \cdot 1 + (a_1 + a_w) \cdot x + \cdots + (a_{w-1} + a_w) \cdot x^w)$.

3.3.2.3 Cas avec un polynome irréductible ESP

Dans le cas d'un polynome ESP irréductible $p(x)$ de degré w, l'anneau défini par le polynome $x^n + 1$ à coefficients dans $\mathbb{F}_2$ est isomorphe au produit cartésien de plusieurs polynomes. La transformée ϕ_e revient à multiplier par 1, soit la somme des l idempotents principaux définissant les l idéaux minimaux de l'anneau :

$$\phi_e : \begin{array}{ccc} B_1 & \rightarrow & R_{2,n} \\ b_B(x) & \mapsto & b_R(x) \end{array} \tag{3.9}$$

De la même manière que pour le cas précédent, la transformée inverse est le modulo $p(x)$ polynome irréductible définissant B_1, isomorphe à A_1 :

$$\phi_e^{-1} : \begin{array}{ccccc} R_{2,n} & \rightarrow & & & B_1 \\ b_R(x) & \mapsto & b_R(x) & \mod p(x) & \end{array} \tag{3.10}$$

3.3.3 La transformée Sparse

Soit ϕ_s la fonction *Sparse* qui consiste à appliquer la transformée isomorphique à un élément de B_1 et la possibilité d'y ajouter un élément de chaque idéal afin d'obtenir un élément de l'anneau dont le poids W est le plus petit possible. En effet, pour tout élément $b(x)$ de B_1, on peut faire correspondre n'importe lequel des éléments $r(x)$ de R tel que $\phi^{-1}(r(x)) = b(x)$.

3.3.3.1 Cas général

Soit $S_{b_B}(x) = \{\phi_1(b_B(x)) + r(x) | \forall r(x) = \phi_1^{-1}(r(x)) = 0\}$ et $W(s(x))$ le poids de Hamming $s(x)$, c'est à dire le nombre de monomes du polynome.

$$\phi_s : \begin{array}{ccc} B_i & \rightarrow & R_{2,n} \\ b_B(x) & \rightarrow & \underset{s(x) \in S_{b_B}(x)}{\arg\min} \; W(s(x)) \end{array} \tag{3.11}$$

Nous avons vu que $\phi_1(b_B(x)) \in A_1$. De plus, l'élément $r(x)$ étant un élément de l'anneau n'ayant pas de composante dans l'idéal minimal A_1 peut être vu comme une somme d'éléments des autres idéaux minimaux. Le résultat de la transformée ϕ_s est donc une somme d'éléments des idéaux principaux de l'anneau, donc un élément de l'anneau.

La transformée Sparse inverse consiste à revenir sur l'élément du corps B_i depuis un élément de l'anneau. Nous utilisons pour cela ϕ_i^{-1} :

$$\phi_s^{-1} : \begin{array}{ccccc} R_{2,n} & \rightarrow & & & B_1 \\ b_R(x) & \rightarrow & b_R(x) & \mod p_i(x) & \end{array} \tag{3.12}$$

Proposition 11. *Pour tout $u(x)$ et $v(x)$ dans B_i, nous avons :*

$$\phi_s^{-1}(\phi_S(u(x)).\phi_S(v(x))) = u(x).v(x)$$

Démonstration.

$$\begin{array}{lc} & \phi_S^{-1}(\phi_S(u(x)).\phi_S(v(x))) \\ = & \phi_1^{-1}(\phi_1(u(x)+r_1(x)).\phi_1(v(x)+r_2(x)) \\ = & \phi_1^{-1}(\phi_1(u(x)).\phi_1(v(x)) + \phi_1(u(x)).r_1(x) + \phi_1(v(x)).r_2(x) + r_1(x).r_2(x)) \\ = & \phi_1^{-1}(\phi_1(u(x)).\phi_1^{-1}(\phi_1(v(x)) + \phi_1^{-1}(\phi_1(u(x)).\phi_1^{-1}(r_1(x)) \\ + & \phi_1^{-1}(\phi_1(v(x)).\phi_1^{-1}(r_2(x)) + \phi_1^{-1}(r_1(x)).\phi_1^{-1}(r_2(x)) \\ = & \phi_1^{-1}(\phi_1(u(x)).\phi_1(v(x)) + 0 \end{array} \tag{3.13}$$

En effet, comme $\phi_1^{-1}(r_1(x)) = 0$ et $\phi_1^{-1}(r_2(x)) = 0$, on obtient $u(x).v(x)$.

□

Cette proposition montre que ϕ_S peut être utilisée pour effectuer la multiplication dans l'anneau. L'intérêt principal de cette transformée est que le poids de l'image de ϕ_S est faible, ce qui réduit la complexité de la multiplication dans l'anneau.

Nous verrons qu'en pratique, la transformée inverse Sparse n'est jamais utilisée. En effet, la transformée Sparse nous sert à transformer les coefficients des matrices génératrices, avant d'effectuer les calculs dans l'anneau. Revenir à des éléments du corps pour ces matrices ne sert à rien, seules les données à coder ou décoder doivent subir une transformée inverse.

3.3.3.2 Cas avec un polynome irreductible AOP

Lorsque un corps fini B_1 est défini par le polynome AOP irréductible $p(x)$ de degré w, ce corps est isomorphe à l'idéal A_1 de l'anneau défini par le polynome $x^{w+1}+1$. En effet, nous avons vu que dans ce cas, l'anneau est égal à la somme directe de idéaux minimaux A_1 et A_2. Tous ses éléments peuvent s'écrire comme une somme d'un élément de A_1 et de A_2. L'idéal minimal A_2 contient uniquement 2 éléments : 0 et le polynome plein de 1 de degré $w+1$. Par conséquent, un élément de l'anneau peut s'écrire comme un élément de A_1 auquel on pourra ou non ajouter le polynome plein de 1 de degré $w+1$, noté $q(x)$ afin d'obtenir un élément le plus creux possible :

$$\phi_s : \begin{array}{ccc} B_i & \rightarrow & R_{2,w+1} \\ b_B(x) & \rightarrow & \phi_i(b_B(x)) + \delta.q(x) \end{array} \tag{3.14}$$

où $\delta = 1$ si $W(\phi_i(b_B(x)) + q(x)) < W(\phi_i(b_B(x)))$ et 0 sinon.

La transformée inverse s'effectue en appliquant ϕ_i^{-1} sur l'élément de l'anneau :

$$\phi_s^{-1} : \begin{array}{ccccc} R_{2,w+1} & \rightarrow & & & B_i \\ b_R(x) & \rightarrow & b_R(x) & \mod & p_i(x) \end{array} \tag{3.15}$$

3.3.3.3 Cas avec un polynome irréductible ESP

En travaillant avec un corps fini B_1 défini par le polynome irréductible ESP $p_1(x)$ de degré w, on peut écrire un élément de l'anneau comme une somme d'un

élément de chaque idéal principal. Or, dans ce cas, l'anneau contient plus de 2 idéaux principaux et certains ont plus de 2 éléments. Par conséquent, le choix est plus important pour obtenir un élément creux. En supposant que l'anneau possède l idéaux principaux, la transformée Sparse est définie comme il suit :

$$\phi_s : \begin{array}{ccc} B_1 & \rightarrow & R_{2,n} \\ b_B(x) & \rightarrow & \phi_1(b_B(x)) + \sum_{j=2}^{l} q_j(x) \end{array} \tag{3.16}$$

où $q_j(x)$ est un élément de l'idéal A_j.

Comme dans le cas précédent, la transformée inverse s'effectue en appliquant ϕ_i^{-1} sur l'élément de l'anneau :

$$\phi_s^{-1} : \begin{array}{ccccc} R_{2,n} & \rightarrow & & & B_1 \\ b_R(x) & \rightarrow & b_R(x) & \mod p_1(x) & \end{array} \tag{3.17}$$

3.3.4 La transformée Parity

Nous allons présenter une dernière transformée, non isomorphique, mais nous verrons dans la suite de ce chapitre que dans certains cas, cette dernière peut avoir son utilité dans un code à effacement.

3.3.4.1 Cas général

Soit ϕ_p la transformée définie comme une bijection quelconque entre le corps B_1 et l'idéal A_1. Par définition, cette bijection permet d'associer à chaque élément du corps B_1 un élément de l'idéal A_1.

3.3.4.2 Cas avec un polynome irréductible AOP

Dans le cas d'un corps fini défini par un polynome irréductible AOP, nous avons :

Proposition 12. *L'idéal A_1 est composé des éléments de $R_{2,w+1}$ de poids pair.*

Démonstration. A partir de la proposition 7, on peut observer que toutes les images de ϕ_1 sont de poids pair. Comme le nombre d'éléments de poids pair dans $R_{2,w+1}$ est égal au nombre d'éléments de A_1, A_1 est composé des éléments de $R_{2,w+1}$ qui sont de poids pair. □

Considérons la fonction ϕ_P, de B_1 vers $R_{2,w+1}$, qui ajoute un bit de parité au vecteur correspondant à l'élément du corps. Par construction, l'élément ainsi obtenu est de poids pair, et d'après la proposition précédente, il appartient donc à A_1.

Définition 18. *La transformée Parity consiste simplement à ajouter un bit de parité à l'élément du corps.*

Comme les images par ϕ_P de deux éléments distincts sont distinctes, ϕ_P est une bijection entre B_1 et A_1. La fonction inverse, ϕ_P^{-1}, consiste à simplement supprimer le dernier coefficient de l'élément de l'anneau.

3.3.4.3 Cas avec un polynome irréductible ESP

Proposition 13. *D'après la proposition 17, l'idéal A_1 de l'anneau est composé des éléments représentés par s polynomes de poids pair et de degré $r+1$ entrelacés.*

Démonstration. Dans le corps $F = \mathbb{F}_2[x]/(g(x))$ avec $g(x)$ un polynome ESP irréductible de paramètres (r, s), les éléments sont composés de s polynomes de degré r entrelacés. Or, nous avons vu que r doit être pair pour que $g(x)$ soit un polynome ESP irréductible. Ainsi, en ajoutant un bit aux s mots de longueur r composant les éléments de F, nous obtenons s mots de poids pair de longueur $r+1$, soit les éléments de A_1. □

3.3.4.4 Particularité de la transformée Parity

La transformée ϕ_P n'est pas un isomorphisme. Il s'agit simplement d'une bijection entre un corps (B_1) et un idéal minimal de l'anneau (A_1). L'objectif ici est d'associer chaque élément du corps à un élément de l'idéal. Nous verrons dans la prochaine section que cette fonction peut être utilisée dans le contexte des codes à effacement et peut, dans certaines conditions, permettre d'obtenir de meilleures performances qu'avec les autres transformées.

3.4 Application des transformées

Nous avons donc à notre disposition 4 transformées qui vont nous permettre d'optimiser les opérations de codage et de décodage dans un code correcteur à effacement. Dans les codes à effacement utilisant des matrices binaires [14], la fonction de codage consiste à multiplier un vecteur d'information par la matrice génératrice binaire. Tous les processeurs sont capables d'effectuer des opérations `xor` sur des mots machines de l bits. Ainsi, l mots de code entrelacés sont codés en parallèle.

Dans cette section, nous considérons un système avec k vecteurs de données et m vecteurs de redondance. Le nombre total d'opérations *xor* est ainsi défini par la matrice génératrice binaire qui doit être la plus creuse possible.

Afin de réduire la complexité de codage, nous utilisons une matrice génératrice binaire $k \times (k+m)$ dans sa forme systématique composée d'une matrice identité $k \times k$ concaténée avec une matrice de Cauchy généralisée de $k \times m$ [49]. Comme vu dans le chapitre précédent, une matrice de Cauchy généralisée génère un code MDS systématique et contient uniquement des 1 dans sa première colonne et sa première ligne. Pour améliorer la densité de la matrice génératrice dans l'anneau, nous appliquons la transormée Sparse ϕ_S sur ses coefficients. Nous avons ainsi une matrice génératrice avec des coefficients appartenant à un anneau.

Concernant les vecteurs d'information, l'utilisation de la transformée Sparse n'a pas d'intérêt. En effet, rendre les données plus creuses en passant dans l'anneau n'accélère pas les opérations de codage : ces opérations manipulent les vecteurs par mots machine. Ces derniers sont sommés en fonction des éléments de la matrice génératrice. Le fait d'avoir les vecteurs représentant les données plus ou moins

creux ne change absolument pas le nombre d'opérations, ni leur vitesse : seuls les éléments de la matrice génératrice permettent cela. En revanche, utiliser les transformées Embedding et Parity permet de transformer une grande quantité de données rapidement car celles-ci sont moins complexes que ϕ_S.

3.4.1 Compatibilité des transformées Embedding et Sparse

Lors de la multiplication matrice vecteur dans le processus de codage ou de décodage, les vecteurs représentant les données et les coefficients de la matrice sont envoyés dans l'anneau par des transformées différentes. Puis, une fois dans l'anneau, toutes les additions et les multiplications sont effectuées. Le résultat, étant un vecteur d'éléments de l'anneau, est alors transformé en un vecteur d'éléments du corps.
Vérifions que les transformées Embedding et Sparse peuvent être utilisées conjointement :

$$\begin{array}{lccc}
\mathbb{F}_{2^w}: & (\alpha_0,\ldots,\alpha_{k-1}) & \times \begin{pmatrix} \gamma_{0,0} & \cdots & \gamma_{0,n-1} \\ \vdots & \ddots & \vdots \\ \gamma_{k-1,0} & \cdots & \gamma_{k-1,n-1} \end{pmatrix} & = (\beta_0,\ldots,\beta_{n-1}) \\
 & \Downarrow \phi_e & \Downarrow \phi_s & \Uparrow \phi_e^{-1} \\
R_{2,n}: & (a_0,\ldots,a_{k-1}) & \times \begin{pmatrix} g_{0,0} & \cdots & g_{0,n-1} \\ \vdots & \ddots & \vdots \\ g_{k-1,0} & \cdots & g_{k-1,n-1} \end{pmatrix} & = (b_0,\ldots,b_{n-1})
\end{array}$$

Proposition 14. *La somme des produits d'éléments du corps peut s'écrire de la forme :*

$$u_1.v_1 + u_2.v_2 = \phi_e^{-1}(\phi_e(u_1).\phi_s(v_1) + \phi_e(u_2).\phi_s(v_2))$$

Démonstration. Comme nous l'avons montré précédemment, nous avons : $\phi_e(u) = \phi_1(u) + r_1(x)$ et $\phi_s(v) = \phi_1(v).r_2(x)$ avec $r_1(x)$ et $r_2(x)$ des éléments de l'anneau n'ayant pas de composante dans A_1.

Ainsi, on peut écrire :

$$
\begin{aligned}
&\phi_e^{-1}(\phi_e(u_1).\phi_s(v_1) + \phi_e(u_2).\phi_s(v_2)) \\
&= \phi_1^{-1}(\phi_e(u_1).\phi_s(v_1) + \phi_e(u_2).\phi_s(v_2)) \\
&= \phi_1^{-1}((\phi_1(u_1) + r_{u_1}(x)).(\phi_1(v_1) + r_{v_1}(x)) + (\phi_1(u_2) + r_{u_2}(x)).(\phi_1(v_2) + r_{v_2}(x))) \\
&= \phi_1^{-1}((\phi_1(u_1) + r_{u_1}(x)).(\phi_1(v_1) + r_{v_1}(x))) + \phi_1^{-1}((\phi_1(u_2) + r_{u_2}(x)).(\phi_1(v_2) + r_{v_2}(x))) \\
&= \phi_1^{-1}(\phi_1(u_1) + r_{u_1}(x)).\phi_1^{-1}(\phi_1(v_1) + r_{v_1}(x)) + \phi_1^{-1}(\phi_1(u_2) + r_{u_2}(x)).\phi_1^{-1}(\phi_1(v_2) + r_{v_2}(x)) \\
&= (\phi_1^{-1}(\phi_1(u_1)) + \phi_1^{-1}(r_{u_1}(x))).(\phi_1^{-1}(\phi_1(v_1)) + \phi_1^{-1}(r_{v_1}(x))) \\
&+ (\phi_1^{-1}(\phi_1(u_2)) + \phi_1^{-1}(r_{u_2}(x))).(\phi_1^{-1}(\phi_1(v_2)) + \phi_1^{-1}(r_{v_2}(x))) \\
&= (u_1 + 0).(v_1 + 0) + (u_2 + 0).(v_1 + 0) \\
&= u_1.v_1 + u_2.v_2
\end{aligned}
$$

□

Lorsque Embedding est utilisée pour transformer les vecteurs d'information et Sparse pour la matrice génératrice, le résultat obtenu dans l'anneau peut être envoyé dans le corps en utilisant ϕ_1^{-1}

3.4.2 Compatibilité des transformées Parity et Sparse

Nous avons défini ϕ_p comme une bijection du corps vers l'idéal A_1 et ϕ_s la transformée Sparse qui transforme un élément du corps en un élément de l'anneau qui est la somme d'un élément de A_1 avec la possibilité d'ajouter un élément des autres idéaux minimaux.
Soit $\bar{u}$ le vecteur de $\mathbb{F}_{2^w}$ à transmettre et G la matrice génératrice $k \times n$ à coefficients dans $\mathbb{F}_{2^w}$.

Scénario 1

Supposons que le codage soit $\bar{u}.G = \bar{z}$ soit réalisé dans le corps et que le vecteur codé $\bar{z}$ soit transmis et soumis au pattern de perte P.

Scénario 2

Considérons maintenant le codage dans l'anneau $\phi_p(\bar{u}) \times \phi_s(G) = \bar{z}$. Supposons que $\bar{z}$ soit soumis au pattern de perte P lors de sa transmission.

De la même manière, nous pouvons vérifier l'utilisation conjointe de la transformée Parity et Sparse :

Proposition 15. *Le pattern de perte P peut être récupéré dans le scénario 2 si et seulement si il peut être récupéré dans le scénario 1.*

Démonstration. Soient $\phi_p(u_1)$ et $\phi_p(u_2)$ les images de 2 éléments du corps par la bijection ϕ_p du corps dans l'idéal.

Soient $\phi_s(v_1)$ et $\phi_s(v_2)$ les images de 2 éléments du corps par ϕ_s. Nous avons : $\phi_s(v_1) = \phi_1(v_1)+r_1$ et $\phi_s(v_2) = \phi_1(v_2)+r_2$ où r_1 et r_2 sont des éléments de l'anneau n'ayant pas de composantes dans A_1, c'est à dire que $\phi_1^{-1}(r_1) = \phi_1^{-1}(r_2) = 0$
Lors du codage dans l'anneau, les opérations sont du type :

$$\begin{aligned}
&\phi_p(u_1).\phi_s(v_1) + \phi_p(u_2).\phi_s(v_2) \\
&= \phi_p(u_1)(\phi_1(v_1) + r_1) + \phi_p(u_2)(\phi_1(v_2) + r_2) \\
&= \phi_p(u_1).\phi_1(v_1) + \phi_p(u_1).r_1 + \phi_p(u_2).\phi_1(v_2) + \phi_p(u_2).r_2 \\
&= \phi_p(u_1).\phi_1(v_1) + 0 + \phi_p(u_2).\phi_1(v_2) + 0
\end{aligned}$$

En effet, $\phi_p(u_1)$ est un élément de A_1 et les éléments r_1 et r_2 sont des éléments de l'anneau n'ayant pas de composantes dans A_1. Autrement dit, le produit de $\phi_p(u_1)$ avec des éléments n'appartenant pas à A_1 donne 0.

Ceci implique que les résultats obtenus par le codage sont les mêmes que ce que l'on aurait obtenu si on avait utilisé ϕ_1 plutôt que ϕ_s pour transformer la matrice. Comme ϕ_1 est un isomorphisme, les propriétés d'inversibilité de la matrice, et notamment la propriété MDS, sont conservés par ϕ_1. Comme la correction du pattern d'effacements ne dépend que de l'inversibilité des sous-matrices génératrices, tout pattern de perte corrigible dans le corps sera aussi corrigible dans l'anneau et inversement. □

Enfin, notons que nous utilisons la bijection définie dans la Section 3.3.4, mais cette proposition est valide pour n'importe quelle bijection entre le corps et un idéal de l'anneau.

3.5 Analyse de la complexité

Dans cette section, nous allons évaluer la complexité en terme de nombre d'opérations *xor* nécessaires dans le processus de codage et de décodage. Concernant les transformées Embedding et Parity utilisées sur les vecteurs d'éléments du corps fini représentant les données à coder, il s'agira de calculer le nombre d'opérations nécessaires pour transformer un élément du corps fini en un élément de l'anneau et vice versa. Concernant la transformée Sparse utilisée pour réduire le nombre d'opérations, il s'agira de calculer le poids moyen des éléments transformés, ce qui nous donnera directement le nombre de *xor* à effectuer représentés par la matrice binaire.

3.5.1 Complexité de codage

Le processus de codage est découpé en trois phases : la transformée du corps vers l'anneau, la multiplication matrice vecteur et la transformée inverse de l'anneau vers le corps. Les données à coder forment un vecteur de k éléments de $\mathbb{F}_{2^w}$. Concernant les transformées appliquées au données à coder, le nombre d'opérations

dépend du type du polynome irréductible utilisé (AOP ou ESP). De plus, les transformées sont faites sur les données en entrée, soit dans un code MDS les k symboles pour l'encodage, et les transformées inverses sont faites sur les données produites, soit pour l'encodage les $n - k$ symboles.

3.5.1.1 Parity avec AOP

Pour un polynome irréductible AOP de degré w, nous avons défini la transformée ϕ_p comme une fonction qui ajoute un bit de parité au vecteur correspondant à l'élément du corps fini afin d'obtenir un élément de poids pair. Il faudra ainsi w *xor* par élément du corps.
La transformée inverse ne nécessite en revanche aucune opération : il s'agit simplement de supprimer ce bit de parité.

3.5.1.2 Parity avec ESP

Le même type de fonction peut être utilisé pour les polynomes ESP de paramètre (r, s). En effet, pour un polynome irréductible ESP de degré w, nous avons défini la transformée ϕ_P comme une fonction qui ajoute un bit de parité à chacun des s mots de longeur r entrelacés. Il s'agira donc de calculer $s.(r-1)$ *xor*.
La transformée inverse ne nécessite pas d'opérations : on supprime les s bits de parités.

3.5.1.3 Embedding avec AOP

Pour un polynome irréductible AOP de degré w, nous avons défini la transformée ϕ_e comme une fonction qui considère simplement l'élément du corps comme un élément de l'anneau, sans aucune opération.
Concernant la transformée inverse, elle est le calcul du reste modulo le polynome AOP $p(x)$. Comme $p(x)$ est un polynome irréductible AOP, l'opération consiste à sommer le dernier bit du vecteur représentant l'élément de l'anneau à tous les autres coefficients : cette opération s'effectue en w *xor*.

3.5.1.4 Embedding avec ESP

De la même manière que pour les polynomes AOP, avec un polynome irréductible ESP de paramètres (r, s), la transformée ϕ_e considère simplement l'élément du corps comme un élément de l'anneau sans aucune opération.
Pour la transformée inverse, la fonction ϕ_e^{-1} calcule le reste modulo le polynome ESP $p(x)$ de degré $r.s$. Grâce à la forme entrelacées du polynome ESP, cette opération consiste à sommer le dernier bloc de s bits aux r blocs. Cette opération s'effectue en $s.r$ *xor*.

3.5.1.5 Sparse avec AOP

Soit $p(x)$ un polynome irréductible AOP de degré w. D'après la Proposition 3, pour que $p(x)$ soit irréductible, w doit être pair.
Considérons les éléments de $A_{w+1} = \mathbb{F}_2[x]/(x^{w+1}+1)$, représentés par tous les vecteurs binaires de longueur $w+1$.

Lemme 1. *Les images de ϕ_S sont les vecteurs de poids inférieur ou égal à $w/2$.*

Démonstration. Soit $b(x)$ un élément de $\mathbb{F}_{2^w} = B$. L'image de $b_B(x)$ par ϕ_S est $b(x)$ ou $b(x)+q(x)$ où $q(x)$ est le polynome plein de 1 de degré w.
On peut observer que $W(b(x)+q(x)) = w+1-W(b(x))$. Cela implique que $\min(W(b(x)+q(x)), W(b(x)) \leq w/2$ car si $W(b(x)) > w/2$, alors $W(b(x)+q(x)) \leq w/2$.
Donc, $W(\phi_S(b(x))) \leq w/2$. □

Proposition 16. *Le poids moyen des images non nulles par ϕ_S dans le cas d'un polynome AOP de degré w est :*

$$W_{\phi_S}(w) = \frac{w+1}{2^{w+1}-2}.\Big(2^w - \binom{w}{w/2}\Big)$$

Démonstration. D'après le Lemme précédent, le nombre d'images non nulles possibles est $2^w - 1$.
Pour $0 < i \leq w/2$, le nombre de vecteurs de poids i de longueur $w+1$ est $\binom{w+1}{i}$.
Le poids moyen W_{ϕ_S} des images non nulles par ϕ_S est donc :

$$\begin{aligned}
W_{\phi_S}(w) &= \frac{1}{2^w-1}\sum_{i=1}^{w/2} i.\binom{w+1}{i} \\
&= \frac{1}{2^w-1}\sum_{i=1}^{w/2} i.\frac{(w+1)!}{i!(w+1-i)!} \\
&= \frac{1}{2^w-1}\sum_{i=1}^{w/2} \frac{(w+1).w!}{(i-1)!.(w-(i-1))!} \\
&= \frac{1}{2^w-1}.(w+1)\sum_{i=1}^{w/2}\binom{w}{i-1} \\
&= \frac{1}{2^w-1}.(w+1)\sum_{i=0}^{w/2-1}\binom{w}{i}
\end{aligned}$$

Comme $\sum_{i=0}^{w}\binom{w}{i} = 2^w$ et que $\binom{w}{i} = \binom{w}{w-i}$, on a $\sum_{i=0}^{w/2-1}\binom{w}{i} = \frac{1}{2}(2^w - \binom{w}{w/2})$.

On a donc :

$$W_{\phi_S}(w) = \frac{1}{2^w - 1}.(w+1).\frac{1}{2}.(2^w - \binom{w}{w/2})$$
$$= \frac{(w+1)}{2^{w+1} - 2}.(2^w - \binom{w}{w/2})$$

□

Pour la suite, il sera intéressant de calculer également le poids moyen de toutes les images par ϕ_s, c'est à dire en intégrant l'élément nul :

Proposition 17. *Le poids moyen des images par ϕ_S dans le cas d'un polynome AOP de degré w est :*

$$W'_{\phi_S}(w) = \frac{w+1}{2^{w+1}}.\Big(2^w - \binom{w}{w/2}\Big)$$

Démonstration. D'après la démonstration précédente, nous avons :

$$W'_{\phi_S}(w) = \frac{1}{2^w}\sum_{i=1}^{w/2} i.\binom{w+1}{i}$$
$$= \frac{1}{2^w}.(w+1)\sum_{i=0}^{w/2-1}\binom{w}{i}$$

Aini, de la même manière que la démonstration précédente, on peut écrire :

$$W'_{\phi_S}(w) = \frac{1}{2^w}.(w+1).\frac{1}{2}.(2^w - \binom{w}{w/2})$$
$$= \frac{(w+1)}{2^{w+1}}.(2^w - \binom{w}{w/2})$$

Enfin, notons que ce poids moyen des images par ϕ_s peut également s'exprimer de la forme :

$$\phi_s = \frac{1}{2^w}\sum_{i=1}^{2^w} W_i$$

avec W_i le poids de l'image par ϕ_s de l'élément i. □

3.5.1.6 Sparse avec ESP

Soit $g(x)$ un polynome ESP irréductible de degré $w = s.r$. Comme nous l'avons vu, les éléments du corps $B_1 = \mathbb{F}_2[x]/(g(x))$ sont des éléments composés de s mots de longueur r entrelacés du corps $B_2 = \mathbb{F}_2[x]/(p(x))$ avec $p(x)$ un polynome AOP

irréductible de degré r.

Proposition 18. *Le poids moyen des images par ϕ_S dans le cas d'un polynome ESP de degré $w = s.r$ est :*

$$W_{\phi_s}(w) = s.W'_{\phi_s}(r)$$

Démonstration. Un élément de B_1 est composé s mots de longueur r appartenant au corps B_2 défini par un polynome AOP irréductible de degré r. L'application de ϕ_s sur les éléments de B_1 revient à appliquer indépendamment ϕ_s sur les s éléments de B_2.
Soit $W_{i,j}$ le poids de l'élément j qui compose les images par ϕ_S des éléments de B_2. Nous avons donc :

$$W_i = \sum_{j=1}^{s} W_{i,j}$$

Le poids moyen des images par ϕ_s dans le cas d'un polynome ESP peut donc s'écrire comme suit :

$$\begin{aligned} W_{\phi_s}(s.r) &= \frac{1}{2^{sr}} \sum_{i=1}^{2^{sr}} \sum_{j=1}^{s} W_{i,j} \\ &= \frac{1}{2^{sr}} \sum_{j=1}^{s} \sum_{i=1}^{2^{sr}} W_{i_s} \\ &= \sum_{j=1}^{s} \frac{1}{2^{sr}} \sum_{i=1}^{2^{sr}} W_{i_s} \\ &= s.W'_{\phi_s}(r) \end{aligned}$$

□

3.5.1.7 Résumé des opérations des transformées

La table 3.1 donne les complexités pour Embedding et Parity obtenues de la section précédente.

	corps vers anneau	anneau vers corps
Embedding	0	$m.w$
Parity	$k.w$	0

Tableau 3.1 – Nombre d'opérations nécessaires par Embedding et Parity

Le choix entre ces deux méthodes dépend donc des valeurs des paramètres du code (n, k) utilisé, avec $m = n - k$ et w le degré du polynome irréductible définissant le corps fini $\mathbb{F}_{2^w}$. Si $k > m$, la transformée Parity a une complexité plus faible que Embedding. Sinon, Embedding est meilleure.

Avant d'appliquer la multiplication matrice vecteur dans l'anneau, il est nécessaire de transformer les éléments de la matrice génératrice. Cela se fait en utilisant la transformée Sparse dont le but est de réduire le nombre d'opérations. Nous avons donc deux éléments de l'anneau à multiplier sous la forme matrice-vecteur. Le premier élément correspond à un coefficient de la matrice et le second à un symbole d'information. La complexité de la multiplication dans l'anneau ne dépend que du poids du premier élément, noté $w_1 \in \{0, 1, \ldots, n\}$. On peut donc exprimer la complexité de la multiplication par $n.w_2$ avec w_2 le poids d'un symbole d'information. Considérons maintenant certaines spécificités des différentes transformées appliquées aux données. Dans la transformée Parity, pour les éléments d'un corps fini défini par un polynôme irréductible AOP, le dernier bit correspondant à la somme des w premiers bits n'est pas transmis sur le canal : il est supprimé par la transformée inverse. Il est donc inutile de le calculer. Ainsi, la complexité de la multiplication devient $w \cdot w_2$. Pour les éléments d'un corps fini défini par un polynome irréductible ESP, le principe est le même car les éléments sont un entrelacement d'éléments d'un corps fini généré par un polynome AOP.
De la même manière, avec la transformée Embedding, le dernier bit du vecteur d'information est toujours égal à 0. Ainsi, il est inutile de considérer les opérations *xor* qui l'utilisent. La complexité de la multiplication est également $w \cdot w_2$.

Afin d'obtenir le nombre moyen d'opérations *xor* faits par la multiplication de la matrice génératrice par les données à coder, nous allons évaluer le poids moyen des coefficients de la matrice génératrice dans l'anneau.
Nous considérerons un code systématique défini par une matrice génératrice de Cauchy généralisée $G_{n,k} = (I|P)$ avec $m = n - k$ dont les coefficients appartiennent à $\mathbb{F}_{2^w}$. Ceux de la première colonne et la première ligne ont pour valeur l'élément 1. Les autres coefficients seront considérés comme étant aléatoires et différents de zéro. En appliquant la transformée Sparse, les éléments de poids 1 restent inchangés : il s'agit des éléments différents de zéro les plus creux possibles. Ainsi, dans l'anneau, les coefficients de la première ligne et la première colonne de cette matrice ont également pour valeur l'élément 1.
Soit w_{ϕ_s} le poids moyen des éléments de $\mathbb{F}_{2^w}$ transformés par Sparse. Dans notre cas, le nombre moyen de d'opérations *xor* à exécuter est donc :

$$(k + m - 1).w + (k - 1).(m - 1).w.w_{\phi_s}$$

En considérant le choix possible entre Parity et Embedding, nous obtenons :

$$(min(k, m) + k + m - 1).w + (k - 1).(m - 1).w.w_{\phi_s}$$

3.5.1.8 Exemple de la complexité de codage sur $\mathbb{F}_{2^4}$

Nous allons maintenant considérer le cas où $w = 4$. Nous avons donc comme matrice génératrice une matrice de Cauchy généralisée à coefficients dans $\mathbb{F}_{2^4}$. Nous

avons également $w_{\phi_S} = 1.66$. Dans [39], table III, l'auteur calcule la densité des matrices générées normalisées par le produit $k.m.w$ de manière à ce que le nombre d'opérations ne dépende pas de la taille de la matrice. Ces valeurs sont comparées à la limite basse de la densité des matrices (LB), soit $(kmw + (k-1)(m-1))$ et à la densité des matrices RDP généralisées, soit $(kw + k(m-1)(2w-1))$:

k	m	Min	Max	LB	RDP
3	3	1.72	1.72	1.11	1.44
4	3	1.65	1.75	1.12	1.44
5	3	1.67	1.87	1.13	1.44
3	4	1.44	2.08	1.12	1.50
4	4	1.88	1.88	1.14	1.50
3	5	2.03	2.03	1.13	1.53

Tableau 3.2 – Densités minimales et maximales normalisées des meilleures matrices pour $w = 4$ (extrait de [39])

Afin de comparer ces résultats aux travaux existants, nous allons calculer ce ratio en faisant varier k pour trois valeurs différentes de m : 3, 5 et 7. Pour chaque paire (k, m), 1000 matrices de Cauchy généralisées sont générées et nous calculons ce ratio :

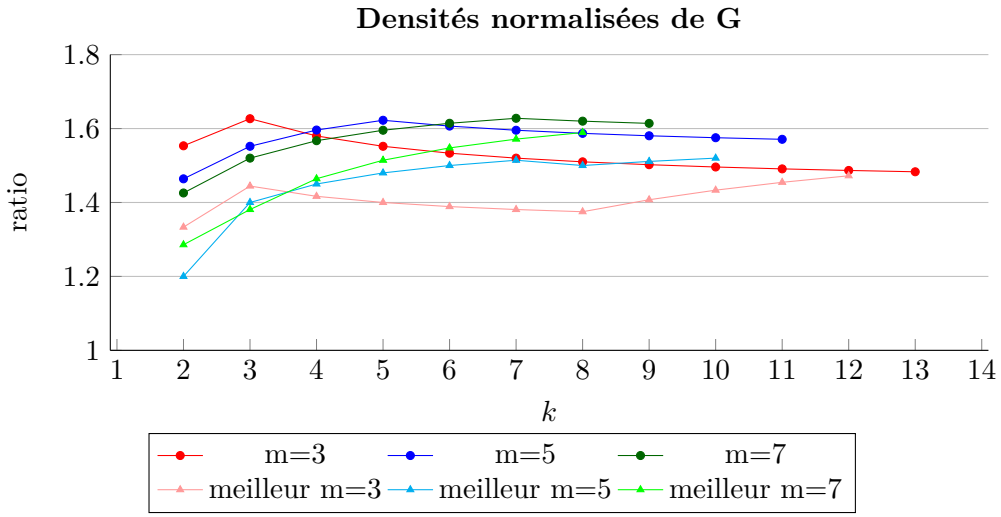
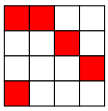
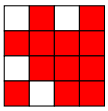


FIGURE 3.1 – Densité normalisée de G en fonction de m

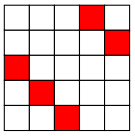
Nous pouvons observer que le fait de passer d'un corps fini à un anneau en utilisant la transformée Sparse pour les éléments de la matrice génératrice permet d'obtenir des matrices génératrices binaires très creuses, contenant un nombre très faible de 1, ce qui réduit la complexité de codage. En comparaison avec les travaux de [39], nous pouvons voir qu'en appliquant une transformée très simple à tous

les éléments d'une matrice, cela permet d'obtenir de meilleurs résultats : pour un code $(8, 3)$, nous obtenons une densité inférieure à 1.6 et pour le meilleur des cas, proche de 1.4 alors qu'en restant dans le corps et en appliquant un algorithme de réordonnancement, les matrices ont une densité au minimum de 1.67.

Une autre approche pour construire des matrice génératrices creuses est d'utiliser les propriétés cycliques de l'anneau. A la différence d'un élément d'un corps fini, un élément d'un anneau polynomial avec un poids de w sera représenté par une matrice binaire carrée composée de w diagonales. Prenons par exemple $w = 4$ et définissons le corps fini $F = \mathbb{F}_{2^4} = \mathbb{F}[x]/(p(x))$ avec $p(x) = x^4 + x + 1$. Les éléments d'une matrice génératrice à coefficients dans F seront représentés par une matrice binaire $w * w$ représentant les opérations *xor*. Soient deux éléments de F, $a(x) = 1 + x^3$ et $b(x) = x + x^3$. La multiplication d'un vecteur d'éléments de F par $a(x)$ ou $b(x)$ se traduira par ces opérations :

$a(x) = 1 + x^3 =$  , $b(x) = x + x^3$ 

Comme nous pouvons le voir, les deux polynomes ont le même poids, mais la multiplication par l'un ou l'autre n'aura pas la même complexité en raison du polynome irreductible utilisé pour le modulo. De manière générale, le seul élément du corps à être représenté par une matrice binaire ayant w entrées est l'élément 1. En revanche, dans l'anneau, le modulo est cyclique : $x^{w+1} = 1$. Dans un anneau polynomial à 2^{w+1} éléments , il existe donc $w + 1$ éléments représentés par une matrice binaire avec $w+1$ entrées. Soit $R_{2,5} = \mathbb{F}[x]/(x^5 - 1)$, le polynome $c(x) = x^2$ peut être réprésenté par la matrice :

$c(x) = x^2 =$ 

Ainsi, en utilisant ces éléments, il est possible de chercher parmi toutes les matrices possibles des matrices MDS. Par exemple, en considérant les éléments du corps $\mathbb{F}_{2^4}$ envoyés dans l'anneau $R_{2,5}$, la matrice de Vandermonde est définie par :

$$V = \left(x^{i.j}\right)_{i=0,\ldots,4;j=0,\ldots,4}$$

où x est un monôme de $R_{2,5}$ avec un nombre minimal de 1. Cette matrice est une matrice MDS et permet de générer un code MDS systématique :

Cette construction permet donc d'obtenir pour un code MDS systématique (n, k) avec $k \leq 5$ et $n - k \leq 5$, et dont le nombre total d'opérations *xor* nécessaires pour

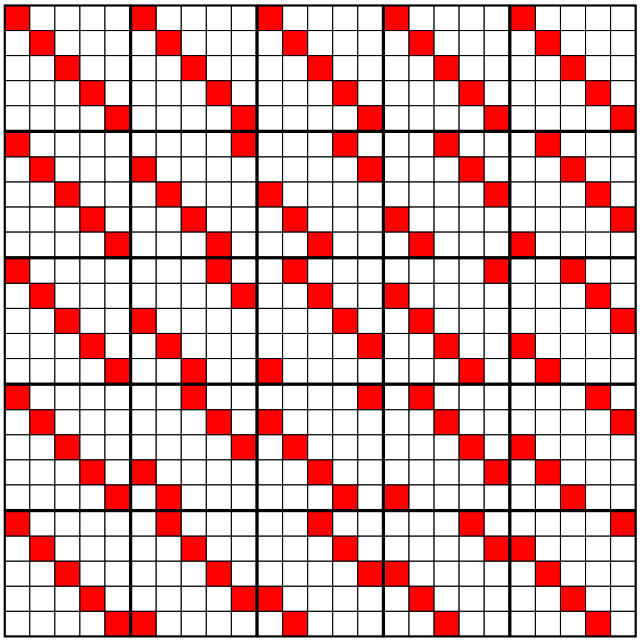


FIGURE 3.2 – Matrice génératrice de Vandermonde

la génération des symboles de redondance en incluant les transformées corps vers anneau et anneau vers corps est de :

$$(\min(k,m)+k+m-1).w+(k-1).(m-1).w=\min(k,m).w+k.m.w$$

Sa densité normalisée est donc de 1.2, ce qui est bien inférieur aux valeurs données dans la Figure 3.1 et très proche de la limite donnée dans [12].

3.5.2 Complexité du décodage

Le processus de décodage est composé de 2 opérations : d'abord, une sous-matrice de la matrice génératrice est inversée, puis une multiplication matrice vecteur utilise cette matrice inversée. Nous pouvons donc utiliser la même approche vue dans la section précédente pour effectuer la multiplication. L'inversion de la matrice peut s'effectuer dans le corps avec une table de correspondance des inverses des éléments. Une fois la matrice inversée, les éléments de cette matrice sont transformés en éléments de l'anneau. L'opération de multiplication matrice vecteur s'effectue alors dans l'anneau, puis les éléments du vecteur résultant de cette opération sont transformés de l'anneau vers le corps.
La complexité du décodage dépend donc de la complexité de l'inversion de la sous-matrice et de la complexité de la multiplication matrice vecteur. Cette dernière a été étudiée dans la section précédente. La complexité de l'inversion d'une matrice $r * r$ est généralement de $\mathcal{O}(r^3)$ opérations dans le corps. Cependant, si la matrice à inverser est une matrice de Cauchy, la complexité est réduite à $\mathcal{O}(r^2)$ [14].

Enfin, contrairement à la multiplication matrice vecteur, l'opération d'inversion de la matrice ne dépend pas de la taille des données à coder ou décoder. Lorsque la taille des données augmente, l'opération d'inversion de la matrice devient ainsi négligeable.

3.6 Ordonnancement des opérations

Il est possible d'optimiser la manière dont sont faites les opérations *xor* dans les codes à effacement MDS. Plusieurs techniques ont été proposées dans [13],[11], [10],[17]. Le principe consiste à factoriser certaines opérations redondantes dans la multiplication matrice vecteur. Nous montrons dans cette section que le fait de travailler dans un anneau permet de réduire significativement le nombre de *xor* nécessaires.

3.6.1 Réduction de la complexité

Pour des éléments d'un corps fini, le réordonnancement des opérations consiste à chercher les opérations redondantes dans la représentation binaire de la matrice génératrice. Autrement dit, sur $\mathbb{F}_{2^w}$, les matrices binaires $w * w$ représentant les éléments du corps n'ont pas de structure particulière et doivent être analysées en entier par l'algorithme de réordonnancement. En effet, comme nous l'avons vu précédemment, le poids d'un élément du corps n'est pas proportionnel au nombre d'opérations *xor* nécessaires.
En passant dans un anneau, grâce à la forme du polynome $x^{w+1}+1$ servant de molulo à la multiplication, les éléments représentés par des matrices binaire $(w+1)*(w+1)$ ne sont composées que par des diagonales, soit pleines de l'élément 0, soit pleines de l'élément 1. Cela signifie que pour connaitre toutes les opérations *xor* nécessaire à la multiplication par un élément de l'anneau, il suffit de connaitre la première colonne de la matrice binaire correspondante à cet élément ou sa représentation polynomiale.
Cela permet de réduire drastiquement la complexité des algorithmes de réordonnancement, leur permettant également de traiter des matrices plus grandes. D'un point de vue polynomial, la recherche de réordonnancement consiste à trouver des redondances dans les opérations de la multiplication.
Supposons le cas où trois polynomes $a_0(x)$,$a_1(x)$ et $a_2(x)$ sont utilisés pour générer trois combinaisons linéaires :

$$(a_0(x), a_1(x), a_2(x)) * \begin{pmatrix} 1+x^4 & 1 & 1 \\ x^2 & x^3 & 1 \\ x^3 & 1+x^3 & x^3 \end{pmatrix} = (p_0(x), p_1(x), p_2(x))$$

Pour estimer la complexité, nous considérons le nombre d'opérations effectuées sur les polynômes. En comptant le nombre de monômes dans tous les éléments de la matrice, nous avons 11 termes additionnés : 4 pour $p_0(x)$, 4 pour $p_1(x)$, 3 pour

$p_2(x)$.
Il est possible de réordonner les opérations en calculant un terme intermédiaire $p'(x) = a_0(x) + x^3 a_2(x)$. Nous avons alors le système suivant :

$$(a_0(x), a_1(x), a_2(x), p'(x)) * \begin{pmatrix} x^4 & 0 & 0 \\ x^2 & x^3 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} = (p_0(x), p_1(x), p_2(x))$$

Ainsi, avec un calcul intermédiaire permettant de réordonner les calculs, nous avons besoin au total de 10 termes à additionner : 2 pour calculer un terme intermédiaire $p'(x)$), 3 pour $p_0(x)$, 3 pour $p_1(x)$ et 2 pour $p_2(x)$.

3.6.2 Factorisation indirecte des opérations

Grâce à la structure cyclique des matrices binaires représentant des éléments de l'anneau, il est possible de factoriser des opérations qui ne sont pas factorisables directement avec des matrices binaires représentant des éléments du corps fini. En effet, dans l'anneau, la multiplication est faite modulo $x^n + 1$. Il est donc possible de factoriser des opérations qui sont multiples d'un monome.
Considérons le cas où trois polynomes $a_0(x)$,$a_1(x)$ et $a_2(x)$ sont utilisés par générer trois combinaisons linéaires :

$$(a_0(x), a_1(x), a_2(x)) * \begin{pmatrix} 1 & x^2 & x^2 \\ x^2 & x^3 & 1 \\ 1+x^2 & x+x^4 & x^2+x^3 \end{pmatrix} = (p_0(x), p_1(x), p_2(x))$$

Dans ce cas là, une factorisation comme montrée précédemment n'est pas faisable. En revanche, il est possible faire apparaitre des opérations pouvant être factorisées en réécrivant les combinaisons linéaires :

- $p_0(x) = a_0(x) + x^2 a_1(x) + (1 + x^2) a_2(x)$
- $p_1(x) = x^2 a_0(x) + x^3 a_1(x) + (x + x^4) a_2(x) = x^2 a_0(x) + x(x^2 a_1(x) + a_2(x)) + x^4 a_2(x))$
- $p_2(x) = x^2 a_0(x) + a_1(x) + (x^2 + x^3) a_2(x) = x^2 a_0(x) + x^3 (x^2 a_1(x) + a_2(x)) + x^2 a_2(x)$

En posant $p'(x) = x^2 a_1(x) + a_2(x)$, nous avons :

$$(a_0(x), a_1(x), a_2(x), p'(x)) * \begin{pmatrix} 1 & 1 & x^2 \\ 0 & 0 & 0 \\ 0 & x^2 & x^2 \\ 1 & x & x^3 \end{pmatrix} = (p_0(x), p_1(x), p_2(x))$$

Comme l'exemple précédent, avec un calcul intermédiaire permettant de réordonner les calculs, nous avons besoin au total de 10 termes à additionner au lieu de 11.

3.6.3 Exemple de réordonnancement pour le code $(12, 8)$

Prenons le cas d'un code systématique $(12, 8)$ sur $\mathbb{F}_{2^4}$. Le codage consiste à générer 4 combinaisons linéaires (r_0, r_1, r_2, r_3) de 8 symboles $(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7)$.

Les opérations sont définies par la matrice suivante :

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & x^2 & x^3+x^2+x & x^3 & x+1 & x & x^2+x+1 & x^3+x^2+x+1 \\ 1 & x^3+x^2+1 & x+1 & x^3+x^2+x & x^2 & x^3+x^2+x+1 & x^3+x+1 & x^3+1 \\ 1 & x^3+x^2+x+1 & x^2+1 & x^3+x^2 & x & x^3+x & x^3+x^2+x & x^2+x \end{pmatrix}$$

En appliquant la transformée Sparse, nous obtenons la matrice suivante :

$$G = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & x^2 & x^4+1 & x^3 & x+1 & x & x^4+x^3 & x^4 \\ 1 & x^4+x & x+1 & x^4+1 & x^2 & x^4 & x^4+x^2 & x^3+1 \\ 1 & x^4 & x^2+1 & x^3+x^2 & x & x^3+x & x^4+1 & x^2+x \end{pmatrix}$$

La première étape pour réduire la complexité est de générer 5 symboles intermédiaires, soit $t = (t_0, t_1, t_2, t_3, t_4)$, à partir des 8 symboles sources :

$$S_1 = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 0 & x^2 & 0 & 0 & 0 & 0 & x^3 & x^4 \\ 0 & 0 & 1 & x^3 & x & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & x & x^4 & 0 \end{pmatrix}$$

On a donc :

$$(s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7) \times [I_8 | S_1^t] = (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, t_0, t_1, t_2, t_3, t_4)$$

Puis, à partir de ces symboles intermédiaires et des symboles sources, on peut générer les symboles de redondance à partir de la matrice suivante :

$$S_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & x^4 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x^4 & x & x^4 & 0 \\ 0 & 0 & 0 & 0 & 0 & x^3 & 0 & 0 & x^2 & x^2 & 1 & 0 & 1 \end{pmatrix}$$

On a donc :

$$S_2 \times (s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, t_0, t_1, t_2, t_3, t_4)^t = (r_0, r_1, r_2, r_3)$$

Nous pouvons vérifier que $[I_8 | S_1^t] \times S_2^t = G$, ce qui permet de valider ces opérations.

Nous pouvons donc générer les symboles de redondance avec respectivement 15 et 19 *xor* pour S_1 et S_2, soit un total de 34 *xor*. C'est 24 % d'opérations en moins comparé aux 45 *xor* de la matrice G initiale dans l'anneau.

3.6.4 Résultats

Afin d'évaluer le gain potentiel qu'apportent ces méthode de réordonnancement des opérations, nous avons implémenté une recherche exaustive des meilleures factorisations sur les matrices génératrices. Cet algorithme a été appliqué sur plusieurs codes travaillant sur le corps fini $\mathbb{F}_{2^4}$. La table 3.3 représente les densités normalisées obtenues, définies comme étant le nombre total d'éléments à 1 dans la matrice génératrice sous sa forme binaire sur le nombre d'éléments à 1 des matrices MDS optimales, soit $k \cdot m \cdot w$.
En travaillant dans un anneau, nous incluons les opérations effectuées lors des transformées. Nous allons présenter ici les résultats obtenus pour les codes $(12, 8)$ et $(16, 10)$. Dans ces deux cas, la transformée Embedding est plus efficace que Parity pour transformer les données. Nous ajoutons donc $m \cdot w$ au nombre de 1 de la matrice génératrice dans sa forme binaire obtenue après réordonnancement.
Pour chaque cas, nous avons généré 1000 matrices de Cauchy généralisées aléatoires. Nous avons mesuré les valeurs suivantes :

- densité moyenne dans le corps : nombre moyen de 1 dans la matrice de Cauchy généralisée divisé par $k \cdot m \cdot w$
- meilleure densité dans le corps : nombre minimal de 1 dans la matrice de Cauchy généralisée divisé par $k \cdot m \cdot w$
- densité moyenne dans l'anneau : nombre moyen de 1 dans la matrice de Cauchy généralisée après transformée Sparse sans réordonnancement divisé par $k \cdot m \cdot w$
- densité minimale dans l'anneau : nombre minimal de 1 dans la matrice de Cauchy généralisée après transformée Sparse sans réordonnancement divisé par $k \cdot m \cdot w$
- densité moyenne avec réordonnancement : nombre moyen de 1 dans la matrice de Cauchy généralisée après transformée Sparse avec réordonnancement divisé par $k \cdot m \cdot w$
- densité minimale avec réordonnacement : nombre minimal de 1 dans la matrice de Cauchy généralisée après transformée Sparse avec réordonnancement divisé par $k \cdot m \cdot w$

$k+m, k$	$12, 8$	$16, 10$
densité moyenne dans le corps	1.79	1.90
densité minimale dans le corps	1.73	1.85
densité moyenne dans l'anneau	1.59	1.63
densité minimale dans l'anneau	1.5	1.58
densité moyenne avec réordonnancement	1.32	1.26
densité minimale avec réordonnancement	1.19	1.20

Tableau 3.3 – densités normalisées de la matrice génératrice pour $w = 4$

Ces résultats confirment que même sans réordonnancement ni factorisation, appliquer la multiplication dans l'anneau plutôt que dans le corps permet de réduire le

nombre d'opérations nécessaires, simplement en appliquant la transformée Sparse sur les éléments de la matrice génératrice. En appliquant un réordonnancement des opérations, il est possible d'obtenir un gain significatif de la complexité. En effet, la complexité est réduite de 20% avec les meilleures matrices. Ces résultats sont similaires à ceux obtenus dans la Section 3.5. Parmi les nombreux travaux effectués sur ce domaine, aucun ne parvient à atteindre les densités qui nous avons obtenues avec ces paramètres.
La structure cyclique de l'anneau nous permet d'utiliser des techniques que les autres méthodes de réordonnancement ne peuvent. De plus, les autres méthodes ont un temps d'exécution dépassant plusieurs minutes, voire plusieurs heures dans certains cas. De plus, ils se heurtent à un problème de passage à l'échelle : comme nous pouvons le constater, les exemples donnés dans [39] sont pour des petits codes. Concernant nos résultats, ils ont été générés par un script réalisé en Python, sans aucun travail d'optiminisation pour rendre l'exécution plus rapide. Pour un code $(12, 8)$, le script a besoin de 2.5 secondes par matrice pour donner le réordonnancement à faire. Pour un code $(16, 10)$, il faut un peu moins de 25 secondes.

3.7 Conclusion

Dans ce chapitre, nous avons étudié une méthode permettant d'optimiser les opérations de codage et décodage des codes correcteurs à effacement basés sur certains corps finis. Cette méthode n'est rendue possible que si le polynome irréductible définissant le corps fini utilisé fait partie de certaines classes des polynomes que nous avons définies. Nous avons montré qu'il est possible de réduire la complexité des opérations de codage et de décodage en utilisant simplement des transformées sur les éléments de ce corps fini. Plusieurs transformées ont été présentées, apportant des propriétés différentes qui leur permettent d'être utilisées à différents endroits dans les opérations de codage.
Cette méthode, à première vue contre-intuitive du fait qu'elle ajoute des opérations nécessaires pour appliquer les transformées, nous permet de réaliser les multiplications de manière beaucoup plus simple dans une anneau polynomial grâce à la structure cyclique de ce dernier. Nous avons également montré qu'en travaillant dans un anneau, il était possible de factoriser beaucoup plus facilement certaines opérations, grâce aux décalages cycliques des *xor* dans la représentation binaire des éléments de l'anneau.

Chapitre 4

Implémentations efficaces des codes à effacement basés sur les transformées polynomiales

Sommaire

4.1 Introduction

Dans ce chapitre, nous présentons une implémentation efficace et une évaluation de performance d'un code à effacement basé sur les transformées polynomiales vues dans le chapitre précédent. Nous appellerons ce codec Pyrit, pour PolYnomial RIng Transforms. Tout d'abord, nous détaillerons deux implémentations de ce codec. L'une est faite en utilisant un corps fini dont le polynôme irréductible est un polynôme AOP de degré 4, l'autre utilise un corps fini dont le polynôme irréductible est un polynôme ESP de degré 6. Puis nous montrerons comment implémenter efficacement les transformées étudiées dans le chapitre précédent. En effet, ces transformées ont la particularité de générer des données supplémentaires lorsque nous passons d'éléments du corps $\mathbb{F}_{2^w}$ à des éléments de l'anneau $R_{2,n}$, ce qui peut réduire les performances lors de l'écriture en mémoire. Nous montrerons également pourquoi et comment il est possible d'éviter ces écritures en mémoire, puis nous

détaillerons l'algorithme de codage et les conditions qui nous permettent d'obtenir les meilleures performances possibles. Puis, nous présenterons les résultats d'une analyse de performance de plusieurs codes à effacement MDS basés sur les corps finis en évaluant ces deux codecs sur deux architectures en les comparant aux codecs les plus performants à notre connaissance. Bien que plusieurs implémentations de l'arithmétique des corps finis existent [21],[41],[30], l'utilisation des instructions SIMD a permis d'améliorer drastiquement les performances des codes à effacement basés sur les corps finis en utilisant les tables de correspondances pour la multiplication et l'inverse des éléments du corps. Nous nous comparerons donc à des codecs implémentant la méthode dite "Split-tables", utilisant des tables de correspondance et les instructions SIMD. Dans notre cas, nous n'utilisons pas de tables pour le calcul des combinaisons linéaires. La quantité de données échangées entre la mémoire et le processeur est donc complètement différente, ce qui peut avoir un impact sur les performances lorsqu'un code s'exécute en parallèle sur plusieurs coeurs du processeur. Nous analyserons donc les performances des deux types d'implémentation dans ce contexte afin de vérifier leur passage à l'échelle.
Concernant les architectures étudiées, la première sera l'architecture x86 qui est très utilisée dans l'industrie. Puis nous étudierons les performances sur l'architecture ARM qui est utilisée dans la plupart des smartphones et présente donc également un intérêt particulier pour ces codes.

4.2 Code à effacement sur $\mathbb{F}_{2^4}$

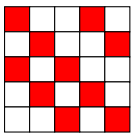
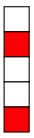
En analysant la liste des polynômes AOP, il s'avère que le polynôme AOP $p(x)$ de degré $w = 4$ est un polynôme irréductible sur $\mathbb{F}_2$ et définit donc le corps fini $\mathbb{F}_{2^4} = \mathbb{F}_2[x]/(p(x))$ avec $p(x) = x^4 + x^3 + x^2 + x + 1$. De plus, sur $\mathbb{F}_2$, nous avons $x^5 + 1 = (x^4 + x^3 + x^2 + x + 1) \cdot (x + 1)$.
Nous pouvons spécifier les propositions définies dans le chapitre précédent pour ces polynômes :
On peut définir $R_{2,5} = \mathbb{F}_2[x]/(x^5 + 1)$ comme l'anneau quotient des polynômes du polynôme $\mathbb{F}_2[x]$ quotienté par l'idéal généré par le polynôme $x^5 + 1$. Sur $\mathbb{F}_2$, le polynôme $x^5 + 1$ est le produit de $p_1(x) = x^4 + x^3 + x^2 + x + 1$ et $p_2(x) = x + 1$. En d'autres termes, tout élément de $R_{2,5}$ peut être écrit de manière unique comme la somme de deux composants $u_1(x) + u_2(x)$, où $u_1(x) \in A_1$ et $u_2(x) \in A_2$. En effet, les idéaux A_1 et A_2 sont des idéaux minimaux avec pour idempotents primitifs respectivement $\theta_1(x) = x^4 + x^3 + x^2 + x$ et $\theta_2(x) = x^4 + x^3 + x^2 + x + 1$. Comme $\mathbb{F}_q[x]/(p_i(x))$ est isomorphe à $B_i = \mathbb{F}_{q^{w_i}}$ où $p_i(x)$ est de degré w_i, $R_{2,5}$ est isomorphe au produit cartésien suivant $R_{2,5} \simeq B_1 \times B_2$ avec $B_1 = \mathbb{F}_2[x]/(p(x)) = \mathbb{F}_{2^4}$ et $B_2 = \mathbb{F}_2[x]/(x + 1) = \mathbb{F}_2$. L'isomorphisme entre $B_1 = \mathbb{F}_2[x]/(p(x)) = \mathbb{F}_{2^4}$ et A_1 est défini comme suit :

$$\phi_1 : \begin{array}{ccc} B_1 & \rightarrow & A_1 \\ b(x) & \rightarrow & \bar{b}(x) = b(x)(p(x) + 1) \end{array} \tag{4.1}$$

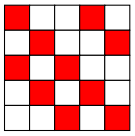
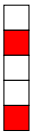
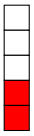
et l'isomorphisme inverse est :

$$\phi_1^{-1} : \begin{array}{ccc} A_1 & \rightarrow & B_1 \\ \bar{b}(x) & \rightarrow & \bar{b}(x) \bmod p(x) \end{array} \tag{4.2}$$

Considérons maintenant deux éléments de $R_{2,5}$: $a(x) = 1 + x^2$ et $b(x) = x + x^4$. Nous souhaitons réaliser la multiplication matrice vecteur avec ces éléments. Ces derniers sont alors représentés de la manière suivante :

$a(x) \longrightarrow$  et $b(x) \longrightarrow$ 

Pour effectuer la multiplication, il suffit alors de multiplier la matrice par le vecteur :

$a(x).b(x) \longrightarrow$  $*$  $=$  $\longrightarrow c(x) = x^3 + x^4$

Nous pouvons vérifier que nous obtenons le même résultat avec la représentation polynomiale dans l'anneau :

$$\begin{aligned} a(x).b(x) &= (1 + x^2).(x + x^4) \bmod (x^5 + 1) \\ &= x + x^3 + x^4 + x^6 \bmod (x^5 + 1) \\ &= x + x^3 + x^4 + x \\ &= x^3 + x^4 \end{aligned}$$

4.2.1 Transformée efficace de la matrice génératrice

L'un des principaux défis dans la construction des codes à effacement est le choix de la matrice génératrice. Comme nous l'avons vu, cette dernière définit les opérations à appliquer, et donc la complexité du code. Nous allons nous intéresser aux codes systématiques. La matrice génératrice devra vérifier deux propriétés :

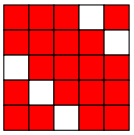
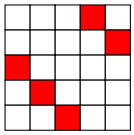
— Elle doit être aussi creuse que possible. En effet, le nombre de *xor* effectués sur les données est défini par cette matrice.

— Chaque sous-matrice carrée devra être inversible afin de vérifier la propriété MDS du code.

Afin de limiter le nombre d'opérations de codage, nous pouvons utiliser une propriété intéressante vue dans le chapitre précédent sur la correspondance entre un élément du corps fini et un élément de l'anneau. Le nombre d'éléments de l'anneau est 2^n, alors que le nombre d'éléménts du corps est 2^w avec $n > w$.
Comme nous l'avons vu, le passage d'un corps fini vers un anneau n'est rendu

possible que par les idéaux de ce dernier : l'anneau possède un idéal isomorphe au corps fini que l'on souhaite utiliser pour définir le code à effacement.

Le corps $\mathbb{F}_{2^4} = \mathbb{F}_2[x]/(x^4 + x^3 + x^2 + x + 1)$ est isomorphe à l'idéal A_1 de l'anneau $R_{2,5}$ généré par $(x + 1)$. Soit $u(x) = x^2$ un élément du corps. D'après les propositions du chapitre précédent, l'image de $u(x)$, notée $\bar{u}(x)$, dans A_1 par l'isomorphisme ϕ_1 est égale à $u(x).\theta_1(x) = x^2.(x + x^2 + x^3 + x^4) = 1 + x + x^3 + x^4$. De plus, d'après le paragraphe précédent, pour effectuer la multiplication avec des éléments de l'anneau, il est possible d'utiliser n'importe quel élément de la forme $\bar{u}(x) + r(x)$ où $r(x)$ n'a pas de composante dans A_1. Dans notre cas, il s'agira de $2^{n-w} = 2$ éléments : 0 et $p(x) = 1 + x + x^2 + x^3 + x^4$. Par conséquent, effectuer une multiplication par $u(x)$ dans le corps peut se faire de deux manière dans l'anneau : en multipliant par $\bar{u}(x)$ ou par $\bar{u}(x) + p(x)$. Les matrices correspondantes sont les suivantes :

$\bar{u}(x) \longrightarrow$  $\bar{u}(x) + p(x) \longrightarrow$ 

La matrice binaire représentant $\bar{u}(x) + p(x)$ est beaucoup plus creuse que celle représentant $\bar{u}(x)$. Elle est donc choisie pour effectuer la multiplication correspondant à $u(x)$ dans l'anneau.
Le choix de l'élément le plus creux dans l'anneau pour un élément du corps donné est fait par la transformée ϕ_S.

4.2.2 Transformées rapides des données à coder

Nous allons voir maintenant comment transformer efficacement les données à coder. Il s'agit d'éléments du corps $\mathbb{F}_{2^4}$. Comme nous l'avons vu dans le chapitre précédent, deux transformées sont possibles. En fonction des paramètres k et m, on choisira l'une ou l'autre.

4.2.2.1 La transformée Parity

La première fonction qui nous permet de travailler dans un anneau calcule simplement un bit de parité sur les données à coder. La fonction inverse consistera à simplement supprimer cet élément. Comme nous l'avons vu dans le chapitre précédent, cette transformée n'est pas un isomorphisme, mais une bijection. Cela signifie que si un codeur effectue les opérations de multiplication dans le corps, le décodeur devra obligatoirement en faire de même, et ne pourra pas effectuer les opérations dans l'anneau en utilisant la transformée Parity. Donc le codeur et le décodeur devront tous les deux utiliser cette même méthode.
Nous avons vu que pour un polynôme AOP irreductile de degré w, l'isomorphisme qui envoie les éléments du corps fini dans l'idéal A_1 de l'anneau $R_{2,w+1}$ consiste en la multiplication du polynôme représentant un élément du corps par l'idempotent définissant l'idéal. Néammoins, nous avons montré que l'idéal A_1 est composé des

polynômes de poids pair. Ainsi, la fonction ajoutant un bit de parité au vecteur représentant un élément du corps fini, permet d'obtenir des éléments de poids pair. Elle peut donc être utilisée comme une bijection entre le corps et l'idéal A_1 de l'anneau.
Dans $\mathbb{F}_{2^4}$, chaque élément est représenté de la manière suivante : $u(x) = a + b.x + c.x^2 + d.x^3$, avec a, b, c, d dans $\mathbb{F}_2$. La transformée ϕ_P est définie comme suit :

$$\phi_P : u(x) \rightarrow (a + b + c + d).x^4 + u(x)$$

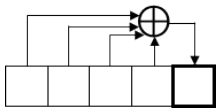


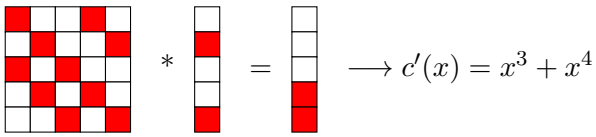
FIGURE 4.1 – Transformée Parity d'un élément de $\mathbb{F}_{2^4}$ vers $R_{2,5}$

Une fois cette transformée appliquée sur nos données, nous effectuons la multiplication matrice vecteur. Le vecteur représente uniquement des éléments de l'idéal A_1 tandis que la matrice représente des éléments de l'anneau n'appartenant pas forcément à l'idéal A_1, mais simplement à l'anneau. La multiplication entre un élément de l'idéal avec un élément de l'anneau donne un élément de l'idéal. Ainsi, le vecteur resultant de la multiplication matrice vecteur contiendra alors uniquement des éléments de l'idéal. Pour appliquer la transformée inverse sur des éléments de l'idéal et obtenir des éléments du corps, il suffira de supprimer le bit de parité pour chaque élément de l'anneau. La transformée ϕ_P^{-1} est définie comme suit :

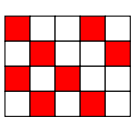
$$\phi_P^{-1} : e.x^4 + u(x) \rightarrow u(x)$$

En conclusion, cette transformée ajoute simplement un bit de parité et peut donc être implémentée de manière très rapide en machine. La transformée inverse ne nécessite aucune opération car il s'agit de supprimer ce bit de parité. En pratique, la transformée sera appliquée sur les données à coder, passant d'éléments représentés par 4 bits à des éléments représentés par 5 bits. Puis, la multiplication vecteur matrice est faite, et la transformée inverse est appliquée sur le vecteur de résultat.

Illustrons ces opérations sur un exemple. Soient $a(x) = 1 + x^2$ et $b(x) = x$ deux éléments de $\mathbb{F}_{2^4}$. Nous souhaitons effectuer une multiplication matrice vecteur de ces deux éléments de manière aussi efficace que possible. Nous utilisons d'abord la transformée ϕ_S sur $a(x)$ et nous obtenons $a'(x) = 1 + x^2$. Nous utilisons ensuite la transformée ϕ_P sur $b(x)$ et nous obtenons $b'(x) = x + x^4$. L'opération de multiplication matrice vecteur est alors :

$a'(x).b'(x) \longrightarrow$  $\longrightarrow c'(x) = x^3 + x^4$

Il faut maintenant appliquer la transformée inverse ϕ_P^{-1} sur le vecteur $c'(x)$, ce qui consiste à supprimer l'élément x^4 pour obtenir $c(x) = x^3$. On peut alors observer que toutes les opérations nécessaires pour calculer le monome x^4 sont inutiles. On peut donc éviter toutes les opérations permettant de la calculer, soit la dernière ligne de la matrice binaire. Autrement dit, en utilisant la transformée ϕ_P et son inverse, la matrice binaire représentant l'élément $b(x)$ dans l'anneau est une matrice $4 * 5$ et la transformée inverse est implicitement réalisée par la suppression des opérations, et on obtient directement l'élément du corps en résultat :

$a'(x).b'(x) \longrightarrow$  $*$  $=$  $\longrightarrow c(x) = x^3$

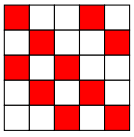
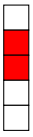
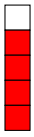
Notons qu'un inconvénient de cette fonction est qu'elle n'est pas un isomorphisme. Par conséquent, si un codeur utilise cette fonction, le décodeur doit impérativement l'utiliser.

4.2.2.2 La transformée Embedding

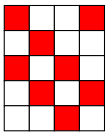
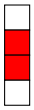
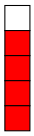
Une seconde transformée peut être utilisée pour transformer les données. Dans le chapitre précédent, nous avons vu que lorsque nous passons du corps à 2^w éléments vers l'anneau à 2^{w+1} éléments, la représentation de ce dernier est toujours plus grande que celle de l'élément du corps. Cela veut donc dire que tous les éléments du corps peuvent être vus comme des éléments de l'anneau. Cette transformée consiste donc simplement à considérer les éléments du corps comme un élément de l'anneau. Elle correspond à celle proposée par Itoh et Tsujii dans [26].
Soit $b(x) = x + x^2$ un élément appartenant à $\mathbb{F}_2[x]/(p(x)) = \mathbb{F}_{2^4}$ représentant nos données à coder. La transformée ϕ_E se fait de la manière suivante :

$$\phi_E : b(x) \rightarrow b'(x) = b(x)$$

Soit $a'(x) = 1 + x^2$ un élément de la matrice génératrice déja transformé par ϕ_S. Le résultat de l'opération de multiplication matrice vecteur est alors :

$a'(x).b'(x) \longrightarrow$  $*$  $=$  $\longrightarrow c'(x) = x + x^2 + x^3 + x^4$

On peut cependant constater que le monome de poids fort de $b'(x)$ est systématiquement égal à 0. Ainsi, effectuer les opérations le concernant n'est pas utile. On peut ainsi supprimer la dernière colonne de la matrice binaire . Nous obtenons alors l'opération suivante :

$a'(x).b'(x) \longrightarrow$  $*$  $=$  $\longrightarrow c'(x) = x + x^2 + x^3 + x^4$

La transformée inverse est effectuée en utilisant l'isomorphisme standard, c'est à dire l'opération modulo $p(x)$ où $p(x)$ est le polynôme irréductible qui définit le corps fini. Dans notre cas, $p(x)$ est un polynôme AOP : $x^4+x^3+x^2+x+1$. Donc, d'après la Section 3.3.2.2, l'opération modulo $p(x)$ consiste à ajouter le dernier monôme de l'élément à tous les autres :

$$\phi_E^{-1} : e.x^4 + b(x) \rightarrow (a+e) + (b+e).x + (c+e).x^2 + (d+e).x^3$$

En reprennant $c'(x) = x + x^2 + x^3 + x^4$ et en lui appliquant la transformée ϕ_E inverse, nous obtenons $c(x) = 1$.

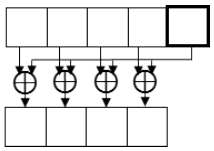


FIGURE 4.2 – Transformée inverse Embedding d'un élément de $R_{2,5}$ vers $\mathbb{F}_{2^4}$

Cette transformée étant isomorphe, le résultat de cette multiplication est le même que celui appliqué dans le corps fini défini par le polynôme $p(x)$:

$$a(x).b(x) = (x + x^2).(1 + x^2) = x + x^3 + x^2 + x^4 = 1$$

Enfin, comme nous l'avons vu dans le précédent chapitre, le choix de la transformée pour les données impacte la complexité de codage : la transformée Parity est plus efficace en nombre d'opérations lorsque $n - k > k$. Dans les autres cas, Embedding est plus efficace.

4.3 Code à effacement sur $\mathbb{F}_{2^6}$

La même approche que pour le corps précédent peut être utilisée avec le corps $\mathbb{F}_{2^6}$. En effet, en analysant la liste des polynômes ESP, il s'avère que le polynôme $g(x) = x^6 + x^3 + 1$ est un polynôme ESP irréductible de degré 6 et de paramètres $s = 3$ et $r = 2$, définissant le corps fini $\mathbb{F}_{2^6} = \mathbb{F}_2[x]/g(x)$. D'après la Proposition 6, on peut donc définir $R_{2,9} = \mathbb{F}_2[x]/(x^9+1)$ comme étant l'anneau $\mathbb{F}_2[x]$ quotienté par l'idéal généré par le polynôme $x^9 - 1$. Sur $\mathbb{F}_2$, le polynôme $x^9 + 1$ est le produit de $g_1(x) = x^6+x^3+1$, $g_2(x) = x^2+x+1$ et $g_3(x) = x+1$. L'anneau $R_{2,9}$ est la somme directe de l'idéal A_1 généré par le polynôme $(x^9 + 1)/g_1(x) = x^3 + 1$, l'idéal A_2 généré par le polynôme $(x^9+1)/g_2(x) = x^7+x^6+x^4+x^3+x+1$ et l'idéal A_3 généré par le polynôme $(x^9+1)/g_3(x) = x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1$. En d'autres

termes, tout élément de $R_{2,9}$ peut être écrit de manière unique comme la somme de trois composants $u_1(x)+u_2(x)+u_3(x)$, où $u_1(x) \in A_1$ et $u_2(x) \in A_2$ et $u_3(x) \in A_3$. En effet, les idéaux A_1,A_2 et A_3 sont des idéaux minimaux avec pour idempotents primitifs respectivement $\theta_1(x) = x^6 + x^3$, $\theta_2(x) = x^7 + x^6 + x^4 + x^3 + x + 1$ et $\theta_3(x) = x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1$. Comme $\mathbb{F}_q[x]/(g_i(x))$ est isomorphe à $B_i = \mathbb{F}_{q^{w_i}}$ où $g_i(x)$ est de degré w_i, $R_{2,9}$ est isomorphe au produit cartésien suivant $R_{2,9} \simeq B_1 \times B_2 \times B_3$ avec $B_1 = \mathbb{F}_2[x]/g_1(x) = \mathbb{F}_{2^6}$ et $B_2 = \mathbb{F}_2[x]/g_2(x) = \mathbb{F}_{2^2}$ et $B_3 = \mathbb{F}_2[x]/g_3(x) = \mathbb{F}_2$. L'image par l'isomorphisme de $b(x) \in B_1$ en A_1 est $\bar{b}(x) = \phi_1(b(x)) = b(x).\theta_1(x)$. De même, l'image d'un élément $\bar{b}(x)$ de A_1 par l'isomorphisme inverse est égale à $b(x) = \phi_i^{-1}(\bar{b}(x)) = \bar{b}(x) \bmod p_1(x)$.

4.3.1 Transformée efficace de la matrice génératrice

Comme pour le cas précédent, nous allons appliquer la transformée ϕ_s aux coefficients de la matrice génératrice. Le corps $\mathbb{F}_{2^6} = \mathbb{F}_2[x]/(x^6 + x^3 + 1)$ est isomorphe à l'idéal A_1 de l'anneau $R_{2,9}$ généré par $(x^3 + 1)$. Soit $v(x)$ un élément du corps. La transformée ϕ_S envoie $v(x)$ sur l'élément de la forme $\bar{v}(x) + r(x)$ où $r(x)$ n'a pas de composante dans A_1 et $\bar{v}(x) = \phi_1(v(x))$. Dans ce cas là, il existe $2^{n-w} = 8$ éléments qui peuvent s'écrire sous cette forme. Par conséquent, effectuer une multiplication par $v(x)$ dans le corps peut se faire de plusieurs manières dans l'anneau. Le choix de l'élément le plus creux parmi les 8 éléments possibles se fera par la transformée ϕ_s.

4.3.2 Transformées rapides des données à coder

Nous allons maintenant définir les transformées Parity et Embedding à appliquer aux éléments de $\mathbb{F}_{2^6}$ représentant les données à coder.

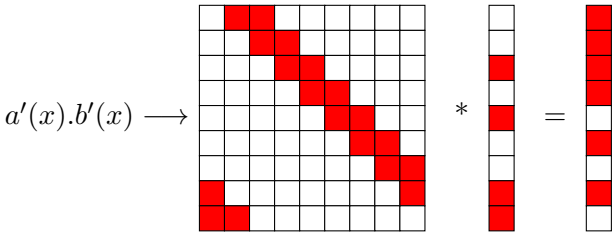
4.3.2.1 La transformée Parity

Comme nous l'avons dit, l'idéal correspondant au corps fini défini par le polynôme irréductible ESP $g(x)$ est l'ensemble des éléments qui peuvent être vus comme un entrelacement de s mots de poids pair et de longueur r. Comme AOP, nous proposons d'ajouter un bit de parité à chaque mot entrelacé de longeur r. Soit l'élément de $\mathbb{F}_{2^6}$ $b(x) = a + b.x + c.x^2 + d.x^3 + e.x^4 + f.x^5$. Nous avons donc la fonction ϕ_P définie par :

$$\phi_P : b(x) \rightarrow b(x) + (a + d).x^6 + (b + e).x^7 + (c + f).x^8$$

Prenons un élément $a(x) = x + x^2 + x^4 + x^5$ appartenant à $\mathbb{F}_{2^6}$, représentant un élément de la matrice génératrice. La transformée ϕ_S appliquée sur cet élément nous donne : $a'(x) = x^7 + x^8$. Supposons $b(x) = x^2 + x^4$ un élément représentant une partie des données à coder. La transformée ϕ_P appliquée à cet élément nous

donne $b'(x) = x^2 + x^4 + x^7 + x^8$. L'opération de multiplication matrice vecteur est alors :



Concernant la fonction inverse, pour les mêmes raisons que les polynômes AOP, il suffit simplement de supprimer les bits de parités du vecteur obtenu après la multiplication matrice vecteur :

$$\phi_P^{-1} : b(x) + (a+d).x^6 + (b+e).x^7 + (c+f).x^8 \rightarrow b(x)$$

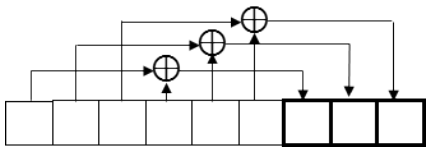
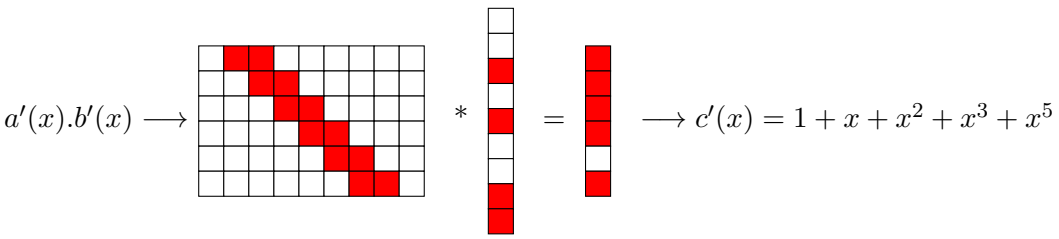


FIGURE 4.3 – Transformée Parity d'un élément de $\mathbb{F}_{2^6}$ vers $R_{2,9}$

En appliquant maintenant la transformée inverse ϕ_P^{-1} sur le vecteur $c'(x)$, nous supprimons les éléments x^6, x^7 et x^8 pour obtenir $c(x) = x+x^3+x^4+x^5$. Toutes les opérations nécessaires pour obtenir ces troix monômes peuvent donc être évitées, ce qui consiste à supprimer les trois dernières lignes de la matrice binaire représentant $b'(x)$. Cette dernière devient donc une matrice 6*9 et la transformée inverse est implicitement réalisée par la suppression de ces opérations, le résultat de la multiplication permet donc d'obtenir directement l'élément du corps :



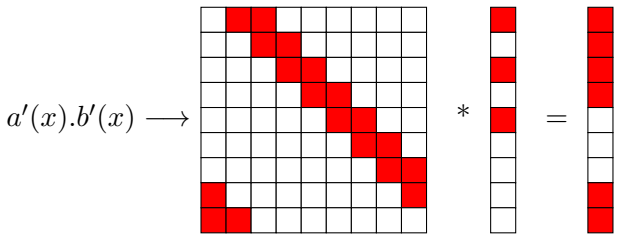
Comme pour le cas précédent, cette fonction n'est pas isomorphe. Si un codeur utilise cette fonction, le décodeur doit impérativement l'utiliser également.

4.3.2.2 La transformée Embedding

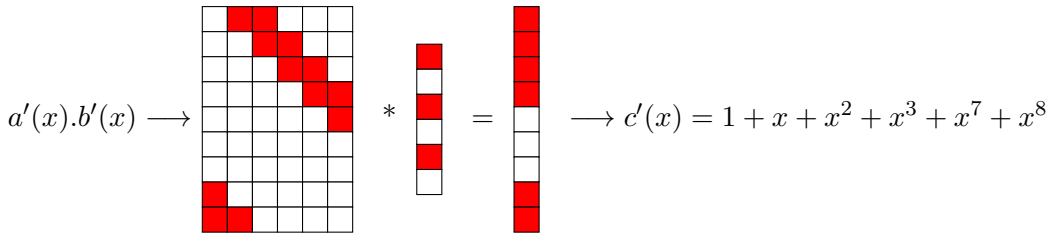
Comme pour les polynômes AOP, la fonction Embedding est directe pour les polynômes ESP :

$$\phi_E : b(x) \rightarrow b'(x) = b(x)$$

Soit un élément $a(x) = x + x^2 + x^4 + x^5$ appartenant à $\mathbb{F}_{2^6}$, représentant un élément de la matrice génératrice. La transformée ϕ_S appliquée sur cet élément nous donne : $a'(x) = x^7 + x^8$. Soit $b(x) = 1 + x^2 + x^4$ un élément représentant une partie des données à coder. La transformée ϕ_E appliquée à cet élément nous donne $b'(x) = 1 + x^2 + x^4$. L'opération de multiplication matrice vecteur est alors :



Comme pour le cas précédent, les 3 monomes de poids fort sont systématiquement à 0. Effectuer ces opérations est donc inutile. On peut donc, dans la représentation par une matrice binaire de l'élément $a(x)$, supprimer les trois dernières colonnes. Nous obtenons ainsi l'opération suivante :



La fonction inverse consiste à calculer le reste modulo $g(x)$, un polynôme ESP de degré $r.s$. Grâce à la structure entrelacée des polynômes ESP, nous pouvons observer que cette opération est équivalente à sommer le dernier bloc de s bits aux r blocs de s bits :

$$\phi_E^{-1} : \begin{array}{c} b(x) + g.x^6 + h.x^7 + i.x^8 \rightarrow \\ (a+g) + (b+h).x + (c+i).x^2 + \\ (d+g).x^3 + (e+h).x^4 + (f+i).x^5 \end{array}$$

Avec $c'(x) = 1 + x + x^2 + x^3 + x^7 + x^8$ et en appliquant ϕ_E^{-1}, nous obtenons $c(x) = 1 + x^3 + x^4 + x^5$.

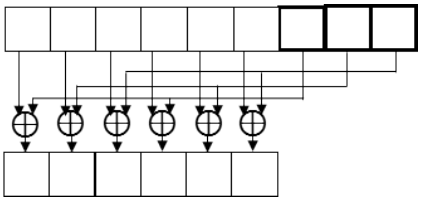


FIGURE 4.4 – Transformée inverse Embedding d'un élément de $R_{2,9}$ vers $\mathbb{F}_{2^6}$

Cette transformée étant isomorphe, le résultat de cette multiplication est le même que celui appliqué dans le corps fini par le polynôme $p(x)$:

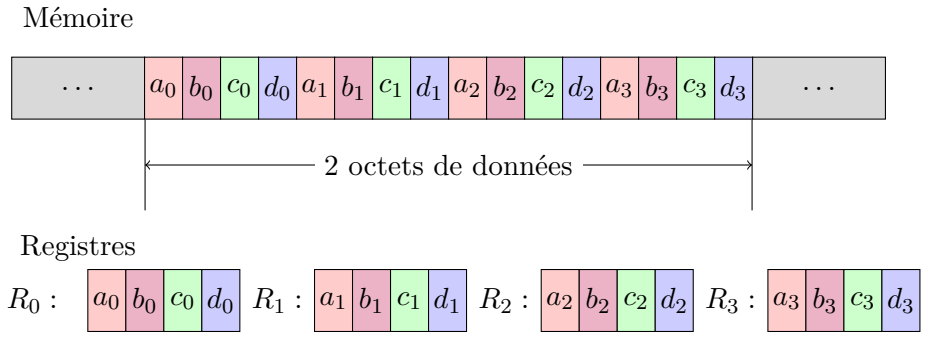
$$a(x).b(x) = (x + x^2 + x^4 + x^5).(1 + x^2 + x^4) = 1 + x^3 + x^4 + x^5.$$

En conclusion, les fonctions de codage et de décodage ne doivent pas obligatoirement utiliser toutes les deux cette transformée pour être compatibles : l'un peut appliquer la multiplication dans le corps, et l'autre peut travailler dans l'anneau avec cette transformée.

4.4 Implémentation efficace d'un code à effacement

Nous allons maintenant détailler une implémentation efficace permise par le passage dans l'anneau des éléments du corps fini. Nous détaillerons les optimisations qui sont permises grace aux transformées vues précédemment.

Cette méthode a été implémentée dans un code à effacement MDS générant des combinaisons linéaires sur deux corps finis ($\mathbb{F}_{2^4}$ et $\mathbb{F}_{2^6}$) en utilisant des matrices de Cauchy généralisées. Cependant, il est tout à fait possible d'utiliser cette méthode sur tout code utilisant un corps fini, qu'il soit MDS ou non. Le code est écrit en C, avec dans certains cas, quelques parties écrites en assembleur. En effet, dans la plupart des machines modernes, le processeur est séparé de la mémoire principale. Il faut donc, pour exécuter des instructions sur des données, les charger depuis la mémoire dans les registres du processeur. Ce chargement peut être long et les processeurs sont capables d'exécuter plusieurs centaines d'instructions pendant que ces données sont en train d'être lues ou écrites dans la mémoire. Pour réduire ce délai, les processeurs sont équipés de plusieurs niveaux de caches. Ils permettent de stocker temporairement des données par bloc de 64 octets. Ainsi, lorsqu'un code à effacement doit transformer S octets de données, chaque élément de $\mathbb{F}_{2^4}$, représenté par 4 bits, sera transformé en un élément de l'anneau à 2^5 éléments, représenté par 5 bits. Ce sont donc $\frac{5}{4} * S$ octets écrits en mémoire. En transformant les éléments de $\mathbb{F}_{2^6}$ en éléments de l'anneau à 2^9 éléments, ce sont $\frac{9}{6} * S$ octets qui devront être écrits en mémoire. De plus, en travaillant sur $\mathbb{F}_{2^4}$ ou $\mathbb{F}_{2^6}$, les éléments sont représentés par des mots de 4 ou 6 bits. Or, dans les processeurs, les données sont manipulées par multiple de 8 bits, allant parfois jusqu'à 512 bits en utilisant des instructions SIMD. Il faut donc trouver une solution afin de faire correspondre au mieux les éléments



des corps finis avec les registres de la machine. Enfin, par construction, l'information supplémentaire générée par ces transformée est détruite lors de la transformée inverse de l'anneau vers le corps fini. Autrement dit, comme seul le processeur a besoin de cette information, il est totalement inutile voire contre-performant d'écrire en mémoire les bits supplémentaires. Dans cette implémentation, nous proposons de réaliser toutes les opérations des transformées dans le processeur, en les réalisant sur les registres, sans y écrire le résultat de l'opération en mémoire. De cette manière, le transfert entre la mémoire et le processeur n'est fait que sur les données à coder.
Pour réaliser cela, une solution est donc de considérer les données à charger de manière entrelacée. En travaillant sur $\mathbb{F}_{2^w}$, nous utilisons w registres de s_r bits pour stocker au total s_r éléments du corps.
Considérons les éléments du corps de la forme $a_0 + a_1.x + a_2.x^2 + a_3.x^3$. Le premier registre contient alors les premiers monomes a_0 des s_r éléments. Dans l'exemple suivant, nous considérons des registres de 4 bits et 4 éléments de $\mathbb{F}_{2^4}$ entrelacés en mémoire. En pratique, nous utilisons les registres SIMD de 128 bits ou plus. Il est donc possible de transformer plusieurs dizaines d'éléments en parallèle.

Une fois les données chargées dans les registres du processeur, nous allons pouvoir effectuer la tranformée. Sur $\mathbb{F}_{2^4}$, avec Parity, nous utiliserons un registre supplémentaire qui sera la somme des 4 premiers registres. Avec Embedding, il suffira de mettre à 0 un registre supplémentaire au lieu de faire la somme, et la transformée inverse consistera à sommer ce registre aux 4 premiers. Sur $\mathbb{F}_{2^6}$, avec Parity, nous aurons besoin de 3 registres suplémentaires, chacun étant la somme de 2 registres. Avec Embedding, il suffira de mettre à 0 3 registres et d'exécuter les opérations. La transformée inverse sera alors la somme de chacun de ces 3 registres. Ces registres serons alors sommés entre eux pour calculer la redondance.

Lorsque les données sont transformées pour passer d'éléments d'un corps fini à des éléments de l'anneau, l'étape suivante consiste à effectuer les opérations *xor* directement avec ces registres. Pour cela, la matrice contenant les coefficients utilisés pour les combinaisons linéaires est transformée en éléments de l'anneau. Cette étape

est faite avec une table de correspondance. Que ce soit sur $\mathbb{F}_{2^4}$ ou sur $\mathbb{F}_{2^6}$, les coefficients de la matrice seront stockés sur 1 octet. En revanche, après transformation, les éléments seront respectivement codés sur 1 et 2 octets.

Dans les codes dits *xor-based*, les éléments de la matrice sont transformés en matrice binaire. Ainsi, en travaillant sur $\mathbb{F}_{2^w}$, il faudra w^2 bits pour stocker un coefficient de la matrice en mémoire. Ces w^2 bits sont lus séquentiellement et si un bit est à 1, un *xor* est alors effectué. Dans le cas d'un code qui effectue la multiplication dans l'anneau, cette étape est différente. En raison du décalage cyclique, les opérations peuvent être directement interprétées en lisant la représentation polynomiale des éléments, soit w bits. Chaque bit lu, représentant un monome du polynôme, se traduira par plusieurs *xor* entre des registres indépendants.

Algorithm 1 Multiplication d'un buffer par un élément de l'anneau avec Embedding sur $\mathbb{F}_{2^4}$

```
1:  function ADDMUL(dst,src,e,sz)
2:      for i from 0 to sz step 64 do
3:          r0 ← dst[i]
4:          r1 ← dst[i + 1 * 16]
5:          r2 ← dst[i + 2 * 16]
6:          r3 ← dst[i + 3 * 16]
7:          r4 ← 0
8:          r5 ← source[i]
9:          r6 ← source[i + 1 * 16]
10:         r7 ← source[i + 2 * 16]
11:         r8 ← source[i + 3 * 16]
12:         if e & 1 then
13:             r0 ← r0 xor r5
14:             r1 ← r1 xor r6
15:             r2 ← r2 xor r7
16:             r3 ← r3 xor r8
17:         end if
18:         if e & 2 then
19:             r1 ← r1 xor r5
20:             r2 ← r2 xor r6
21:             r3 ← r3 xor r7
22:             r4 ← r4 xor r8
23:         end if
24:         if e & 4 then
25:             r2 ← r2 xor r5
26:             r3 ← r3 xor r6
27:             r4 ← r4 xor r7
28:             r0 ← r0 xor r8
29:         end if
30:         if e & 8 then
31:             r3 ← r3 xor r5
32:             r4 ← r4 xor r6
33:             r0 ← r0 xor r7
34:             r1 ← r1 xor r8
35:         end if
36:         if e & 16 then
37:             r4 ← r4 xor r5
38:             r0 ← r0 xor r6
39:             r1 ← r1 xor r7
40:             r2 ← r2 xor r8
41:         end if
42:
43:         r0 ← r0 xor r4
44:         r1 ← r1 xor r4
45:         r2 ← r2 xor r4
46:         r3 ← r3 xor r4
47:
48:         dst[i] ← r0
49:         dst[i + 1 * 16] ← r1
50:         dst[i + 2 * 16] ← r2
51:         dst[i + 3 * 16] ← r3
52:     end for
53: end function
```

L'algorithme 1 présente la méthode permettant d'ajouter à un buffer le résultat de la multiplication d'un autre buffer par un élément de l'anneau. Nous supposons

que nous travaillons avec des registres de 128 bits, soit 16 octets. La première étape est de charger depuis la mémoire les données à modifier (ligne 3 à 6). Comme nous travaillons sur $\mathbb{F}_{2^4}$ avec la transformée Embedding, il faut ajouter à chaque élément du corps un bit à 0 pour obtenir un élément de l'anneau. Cela est fait en mettant à 0 le registre r4 ligne 7. Ensuite, ligne 8 à 11, nous chargeons dans 4 autres registres les données à multiplier. Les 5 blocs "if" parcourent l'élément de la matrice génératrice e, donnée en paramètre. Pour chaque bit à 1, équivalent à une diagonale complète dans la représentation en matrice binaire, nous effectuons 4 *xor* sur les données. Enfin, lignes 43 à 46, nous effectuons la transformée ϕ_E^{-1}, puis nous écrivons le résultat en mémoire lignes 48 à 51.
Le même principe peut s'appliquer en travaillant sur $\mathbb{F}_{2^6}$. La seule condition est d'avoir suffisamment de registres à disposition pour garder dans le processeur les données à coder. Généralement, les processeurs possédant un jeu d'instruction SIMD proposent au moins 16 registres. C'est le cas avec SSE et AVX sur Intel et Neon sur ARM. La prochaine version d'AVX, appellée AVX512 proposera 32 registres de 512 bits. Cela permettrait d'avoir suffisamment de registres pour travailler sur $\mathbb{F}_{2^{10}}$ utilisant l'anneau $R_{2,11}$.

4.5 Evaluation de performances

Dans cette section, nous allons évaluer les performances de ces deux codecs. Les tests ont été réalisés sur deux machines. La première est équipée d'un processeur Intel core i7-6700 à 3.4 GHz et de 16 Go de RAM. La seconde machine, un Raspberry pi 3, est équipée d'un processeur ARM Cortex-A53. Trois cas d'utilisations ont été définis :

— un code (12,8) sur $\mathbb{F}_{2^4}$ en utilisant la transformée Embedding,
— un code (60,40) sur $\mathbb{F}_{2^6}$ en utilisant la transformée Embedding,
— un code (60,20) sur $\mathbb{F}_{2^6}$ en utilisant la transformée Parity.

Afin de comparer les vitesses de codage, nous avons cherché les meilleurs codes disponibles. Nous avons retenu deux projets : Jerasure [4] et ISA-L[3]. Jerasure est une bibliothèque de codecs pour codes à effacement. Elle a été développée par James Plank et est maintenue aujourd'hui par une petite communauté de développeurs. Elle est écrite en C et fonctionne sur plusieurs architectures, notamment celles qui nous interessent, x86 et ARM. Dans [41], les auteurs ont comparé les différentes méthodes de codage sur différents corps finis. Il en ressort que la méthode la plus efficace est l'utilisation des instructions SIMD avec la méthode « split tables »sur $\mathbb{F}_{2^w}$ et que les performances sont quasiment identiques pour $w = 4$ et $w = 8$. ISA-L est une bibliothèque de fonctions entièrement écrites en assembleur, spécifiquement pour les processeurs Intel. Elle propose un codec Reed-Solomon pour le canal à effacement en travaillant sur le corps fini $\mathbb{F}_{2^8}$ avec la méthode « split tables »sur SSE et AVX, les instructions SIMD d'Intel. Etant écrit en assembleur, ce codec est à notre connaissance le plus performant disponible pour l'architecture x86. Lorsque cela sera possible, c'est à dire lorsque nous utiliserons des processeurs Intel, nous

comparerons Pyrit avec ISA-L. Pour les autres cas, nous comparerons Pyrit avec le codec de Jerasure le plus performant selon les paramètres du code.

Le processus de codage revient à calculer $n - k$ combinaisons linéaires des k symboles sources, la matrice génératrice étant calculée avant. Quant au décodage, il correspond à la mesure dans le pire cas, soit la perte de $n - k$ symboles. Le processus de décodage revient donc à une inversion de matrice de dimension $k * k$, puis au calcul de $n - k$ combinaisons linéaires des k symboles. Pour de grandes quantités de données, l'inversion de la matrice au décodage est négligeable dans le calcul de la vitesse, mais cela n'est pas le cas pour les petites quantités de données à coder.

Les vitesses affichées correspondent à la quantité totale de données, c'est à dire les n paquets d'un bloc, divisée par le temps nécessaire au calcul. Pour certaines courbes, nous afficherons une limite représentant le moment où la quantité de données d'un bloc dépasse la taille du cache de dernier niveau : cache L3 sur x86 et cache L2 sur ARM. Nous expliquerons pourquoi lors de l'analyse des résultats.

4.5.1 Analyse de performance sur architecture x86

Nous allons d'abord comparer les vitesses de codage de Pyrit avec celles du codec d'ISA-L sur une machine équipée d'un processeur x86 Intel Core i7 6700. Nous utiliserons pour les deux codecs les instructions AVX. Elles permettent de manipuler les données par bloc de 256 bits à l'aide de 16 registres.

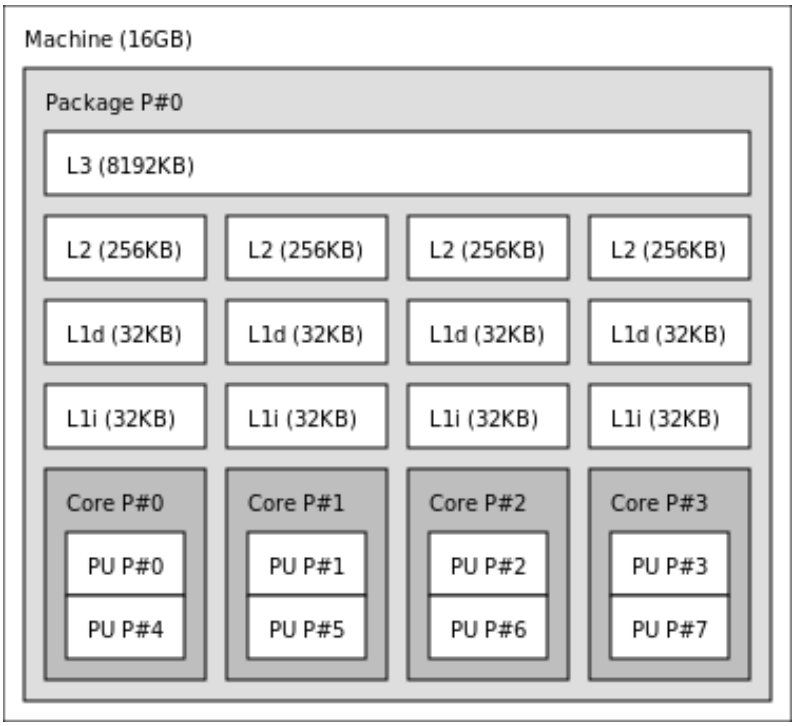


FIGURE 4.5 – Topologie du processeur Intel Core i7 6700

La Figure 4.5 représente la topologie du processeur sur lequel nous avons effectué l'analyse de performance. On peut voir que le processeur dispose de 8 coeurs logiques s'appuyant sur 4 coeurs physiques. Chaque coeur possède un cache L1 et L2, respectivement de taille 32 Ko et 256 Ko. Le cache de niveau 3, L3, de 8Mo est lui partagé entre tous les coeurs.

4.5.1.1 Le code (12,8)

Dans le chapitre précédent, nous avons présenté quelques résultats sur le réordonnancement des opérations permettant de réduire le nombre d'opérations nécessaire au processus de codage. Dans la première partie, nous analyserons les performances générales des codecs Pyrit et ISA-L. Puis, nous analyserons les performances réelles de ce réordonnancement en comparant un codage avec Pyrit utilisant une matrice génératrice MDS non modifiées avec deux autre codes utilisant des symboles intermédiaires permettant de réduire le nombre d'opérations.

Performances sans réordonnancement

Pour un code de paramètres (12,8), avec Pyrit, il est possible de travailler avec le corps fini $\mathbb{F}_{2^4}$. En revanche, ISA-L utilisera toujours le corps fini $\mathbb{F}_{2^8}$.

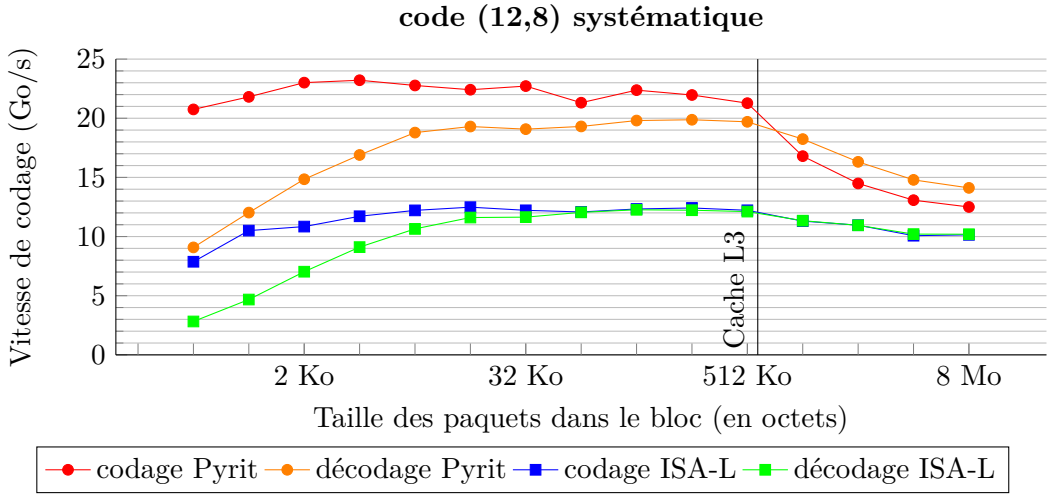


FIGURE 4.6 – Vitesses de codage pour un code à effacement de paramètres (12,8)

Nous pouvons voir que lorsque la quantité de données, c'est à dire les $n = 12$ paquets de données compris dans le bloc, est inférieure à la taille du cache de niveau 3, le codec Pyrit est beaucoup plus performant qu'ISA-L. Le gain est d'environ 100%. En effet, le cache L3 est le dernier niveau de cache du processeur, le plus proche de la mémoire. Il permet de masquer les latences des acccès mémoires, prenant plusieurs centaines de cycles CPU. On constate également que le décodage est beaucoup plus lent que le codage pour les petites tailles de données. En effet,

le décodage nécessite l'inversion d'une matrice de taille $(n-k)*(n-k)$. Pour des données de petite taille, l'inversion de la matrice n'est pas négligeable dans le processus de décodage. Lorsque la taille des données est supérieure à la taille du cache L3, les vitesses de Pyrit et ISA-L ont tendance à converger. En effet, beaucoup plus d'accès mémoires sont nécessaires. Nous en discuterons plus en détails dans la conclusion de cette section.

Performances avec réordonnancement

Dans le chapitre précédent, Section 3.6, nous avons présenté des résultats concernant la réduction du nombre d'opérations au codage. Cette réduction est permise par la "factorisation" de certaines opérations générant quelques symboles intermédiaires. Nous allons ici comparer les vitesses de codage d'un code (12,8) dont la matrice génératrice est une matrice de Cauchy généralisée avec deux codes dont la matrice a été optimisée par un réordonnancement des opérations. La première méthode, appelée "codage 1 inter" dans le graphique, utilise un seul symbole intermédiaire. Cette fonction de codage garde ce symbole dans les registres du processeur sans l'écrire dans la mémoire. La seconde méthode, appelée "codage 5 inter", reprendra les matrices utilisées dans la Section 3.6.3, utilisant cinq symboles intermédiaires. En revanche, étant donné le nombre de registres disponibles, ces 5 symboles seront générés et écrits dans la mémoire.
Le codage avec la matrice de Cauchy généralisée nécessite 48 *xor* de symboles pour générer les 4 symboles de redondance. La méthode avec un symbole intermédiaire nécessite 38 *xor* tandis que la dernière méthode avec 5 symboles intermédiaires nécessite 34 *xor*.

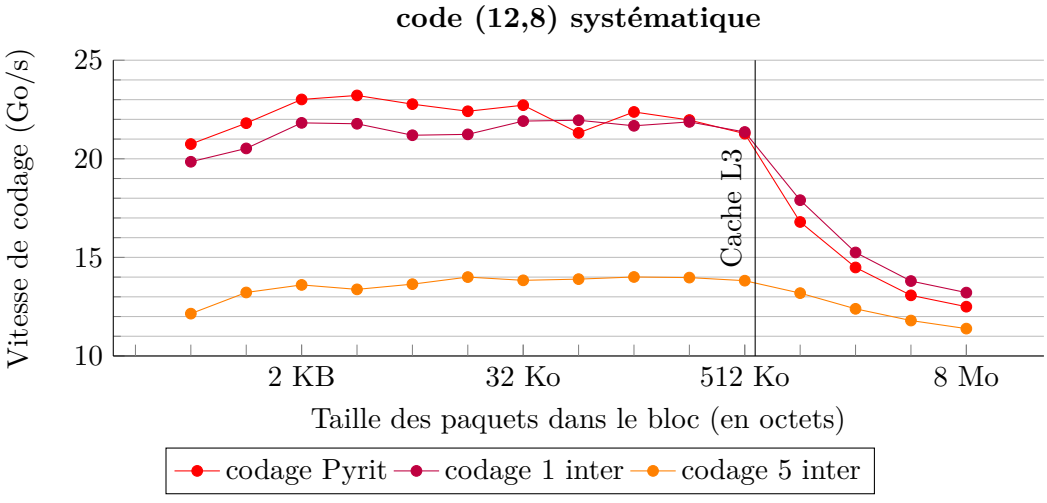


FIGURE 4.7 – Vitesses de codage de Pyrit pour un code à effacement de paramètres (12,8)

Le fait de générer un symbole intermédiaire à la volée, sans écrire le résultat de

la combinaison linéaire dans la mémoire permet de réduire le nombre de *xor* de 48 à 38 *xor* par symbole de redondance généré. On peut cependant constater que les deux approches ont la même vitesse de codage. Avec ces deux approches, la quantité de données lue et écrite depuis et vers la mémoire est identique. En travaillant sur $\mathbb{F}_{2^4}$, les calculs n'ont que peu d'impact comparés aux accès mémoires. En revanche, générer 5 symboles intermédiaires permet de réduire encore un peu le nombre de *xor* pour atteindre 34 *xor*. Mais le fait d'écrire le résultat du calcul de ces combinaisons linéaires en mémoire fait chuter drastiquement les performances.

Conclusions sur les performances du code (12,8)

Le processeur sur lequel ont été effectuées les mesures est équipé de 4 coeurs cadencés à 3.4 GHz, soit un cycle d'horloge effectué en 0.3 nanosecondes. En ce qui concerne la lecture et l'écriture depuis et vers la mémoire, pour ce processeur, Intel annonce une latence de l'ordre de 60 nanosecondes (42 cycles + 51 nanosecondes), soit environ 200 cycles ([2]). Autrement dit, le temps de charger un bloc de données depuis la mémoire, soit une ligne de cache de 64 octets, le processeur a eu le temps d'exécuter 200 instructions par coeur. Bien entendu, ces chiffres ne considèrent pas que les données soient copiées dans les caches L1, L2 ou L3 des processeurs Intel. En revanche, ces derniers ont pour rôle de réduire cette latence. On peut donc affirmer que les performances des codecs dépendent principalement des accès mémoires. C'est ce qui explique que le codec ISA-L soit moins performant que Pyrit : l'algorithme "Split tables" utilisé par ISA-L sollicite énormément la mémoire pour aller chercher les résultats des multiplications dans deux tables de correspondance. Le codec Pyrit, étant un codec dit *xor-based* n'a pas à effectuer cette opération et calcule directement le résultat de la multiplication par une série de *xor*.

4.5.1.2 Le code (60,40)

Ce deuxième cas permet d'évaluer Pyrit sur $\mathbb{F}_{2^6}$ en utilisant la transformée Embedding avec le codec d'ISA-L.

La Figure 4.8 montre les vitesses de codage et de décodage des codecs Pyrit et ISA-L pour un code de paramètres $(60, 40)$. Comme le cas précédent, le codec Pyrit se montre plus performant si les données rentrent dans le cache de niveau 3 du processeur. La différence est cependant moins importante que pour le $(12, 8)$. Cela s'explique par le fait que plus de *xor* sont nécessaires sur $\mathbb{F}_{2^6}$, et donc, les accès mémoires représentent proportionnellement moins de temps.

Lorsque la taille des données dépasse celle du cache L3, le codec Pyrit présente un gain de vitesse d'environ 50 % pour le codage et le décodage. Le codec d'ISA-L nécessitant de nombreux accès mémoire pour effectuer les multiplications dans le corps, sature beaucoup plus rapidement les caches et par conséquent nécessite beaucoup plus d'accès mémoires que Pyrit.

Enfin, les vitesses de codage et de décodage convergent car le nombre de combinaisons linéaires à calculer au codage et au décodage est le même et l'opération

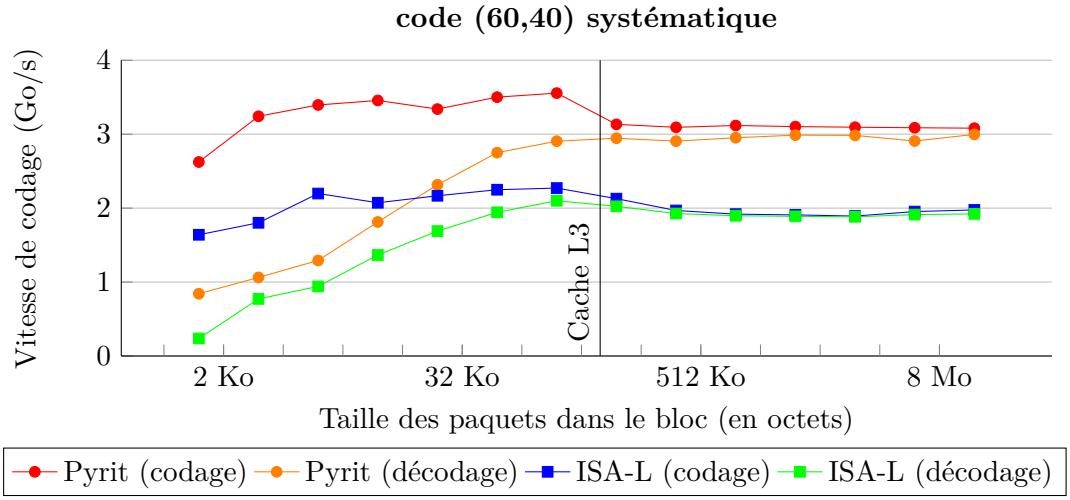


FIGURE 4.8 – Vitesses de codage pour un code à effacement de paramètres (60,40)

d'inversion de matrice, présente uniquement au décodage, devient négligable face aux calculs des combinaisons linéaires lorsque les données a traiter sont de grande taille.

4.5.1.3 Le code (60,20)

Le dernier cas étudié permet de mesurer les performances du codec Pyrit en utilisant la transformée Parity.

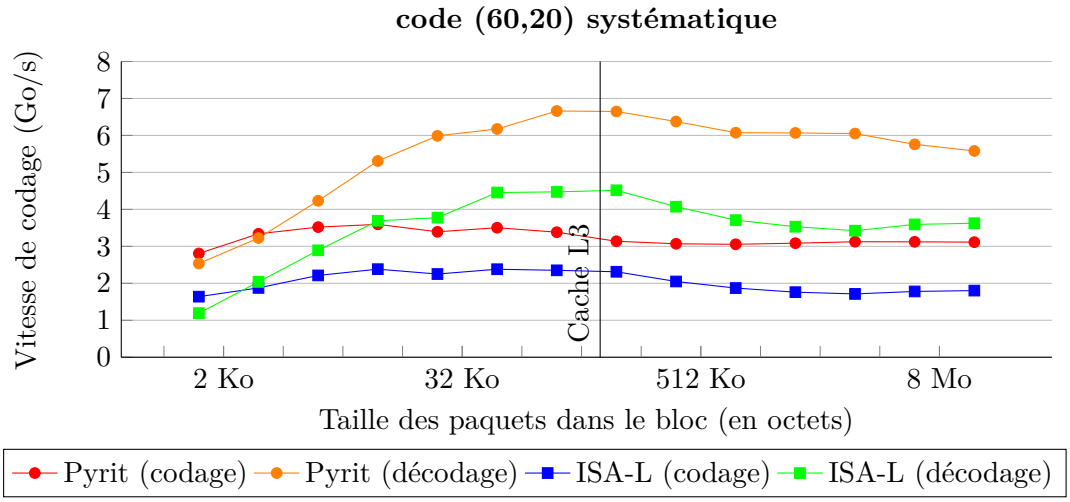


FIGURE 4.9 – Vitesses de codage pour un code à effacement de paramètres (60,20)

Ce cas est intéressant car il illustre le cas d'un code avec n beaucoup plus grand

que k. Il est utilisé lorsque beaucoup de redondance doit être générée. A la différence du code $(60, 40)$, le cache L3 semble avoir moins d'impact sur les performances. Avec ces paramètres, le nombre de combinaisons linéaires à calculer est plus important que le cas précédent. Avec k pertes parmis les n paquets, le codage demande deux fois plus de combinaisons linéaires, ce qui explique la différence de vitesse entre le codage et le décodage. Avec Pyrit, on constate toujours un gain moyen de 50 % pour le codage et le décodage.

4.5.1.4 Conclusions sur les performances sans parallélisation

Sur une architecture x86, pour un code comprennant moins de 64 symboles, le codec Pyrit permet d'obtenir un gain de 50 % pour les vitesses au codage et au décodage. Notons que lors des tests, la machine n'exécutait aucun traitement, rendant totalement disponible le processeur pour les calculs.
Avec un seul des quatre coeurs du processeur qui effectue le codage ou le décodage, les caches, en particulier le cache L3, minimisent les latences pour le codec ISA-L lorsque celui-ci sollicite la mémoire en accédant aux tables de correspondances pour effectuer les multiplications. Il est donc intéressant de voir les performances que donne une exécution parallèle de ces codes.

4.5.2 Exécution parallèle sur x86

Dans cette partie, nous analysons les performances du codec Pyrit face au codec ISA-L dans le cas d'un code s'exécutant en parallèle sur plusieurs coeurs du processeur. En effet, depuis plus de 10 ans, tous les processeurs possèdent plusieurs coeurs permettant d'exécuter du code en parallèle. Le codec Pyrit fonctionnant différemment du codec d'ISA-L, il nous a semblé intéressant d'évaluer les performances d'un processus de codage réparti sur plusieurs coeurs du processeur. De plus, les processeurs qui équipent les serveurs aujourd'hui possèdent plusieurs dizaines de coeurs de calcul. Ce passage à l'échelle est donc important car il représente réellement le cas d'usage de ces codes dans l'industrie.

Pour mesurer les performances, nous avons donc apporté le support du multithreading aux codecs Pyrit et ISA-L. Pour cela, les codecs découpent les données à coder et chaque coeur du processeur travaille sur une partie des données différentes.

En effet, pour paralléliser les opérations de codage, plusieurs approches sont possibles. Il aurait été possible de répartir la génération des combinaisons linéaires sur les coeurs de calcul. Cependant, cette approche aurait nécéssité la recopie des données à coder dans les caches L1 et L2 de tous les coeurs. Avec notre approche, chaque coeur de calcul travaille sur une zone mémoire différente. Enfin, nous nous sommes concentrés sur les cas complexes, c'est à dire sur les codes $(60, 40)$ et $(60, 20)$. Pour chacun des cas, nous mesurons les vitesses de codage et décodage avec les codecs configurés pour utiliser 1,2,4 et 8 threads.

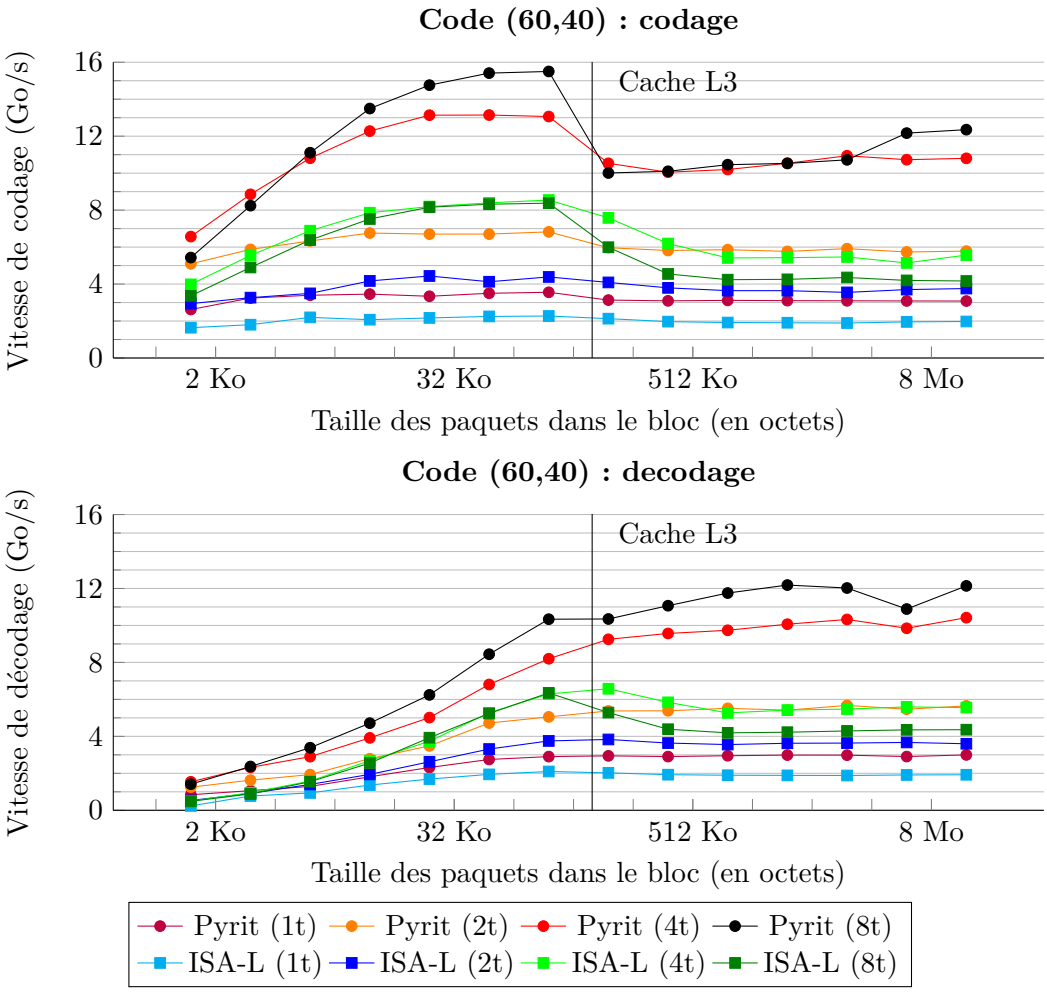


FIGURE 4.10 – Vitesses de codage et de décodage multithread pour un code à effacement de paramètres (60,40)

4.5.2.1 Le code (60,40)

A la différence du codec d'ISA-L, le codec Pyrit n'utilise pas de tables de correspondances pour calculer les combinaisons linéaires. Ainsi, seules les données à coder transitent depuis la mémoire vers le processeur par les caches. Par conséquent, Pyrit est capable de traiter beaucoup plus de données avant que les caches ne saturent. Le codec d'ISA-L est même moins performant lorsqu'il est configuré pour utiliser 8 threads au lieu de 4, ce qui n'est absolument pas le cas pour Pyrit. Au final, cela permet d'observer un gain de 100 % pour le codage et le décodage en prennant en compte les deux meilleures configurations pour les codecs, soit 8 threads pour Pyrit et 4 threads pour ISA-L.

Afin d'avoir une vue plus détaillée des performances de Pyrit, la Figure 4.11 représente le gain obtenu face à ISA-L en fonction du nombre de threads utilisés

pour le codage. On constate que plus les données à traiter sont importantes, meilleur Pyrit est face à ISA-L.

En utilisant uniquement un thread, quel que soit la quantité de données à coder, le gain est constant. En revanche, lorsque le nombre de threads et la quantité de données augmentent, le gain est plus important.

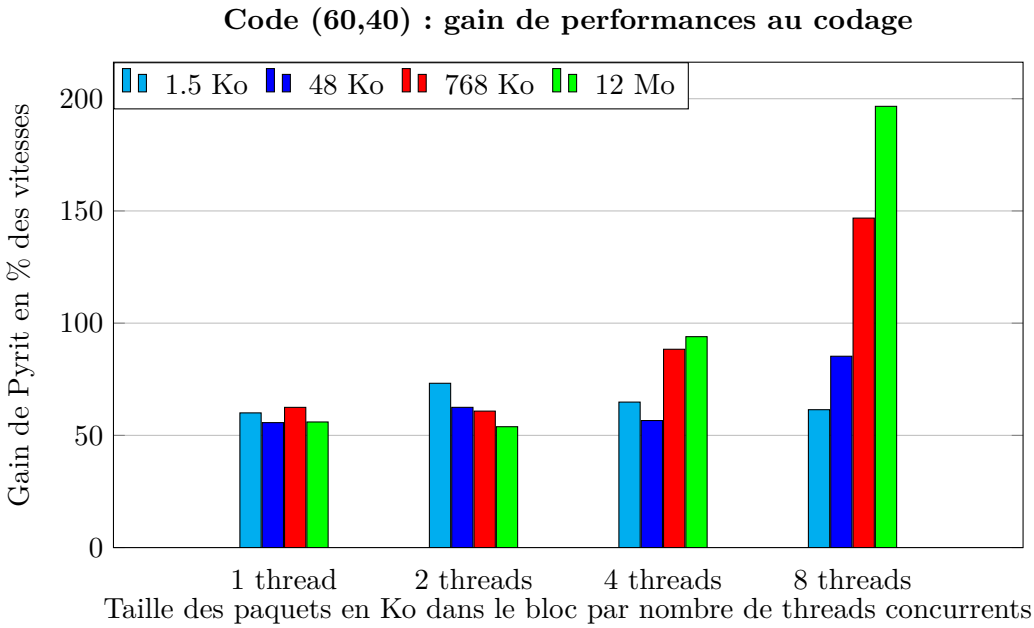


FIGURE 4.11 – Gain de performance du codec Pyrit par rapport au codec d'ISA-L

4.5.2.2 Le code (60,20)

Nous avons évalué les performances sur plusieurs threads pour le code $(60, 20)$. La Figure 4.12 présente les résultats.

Le premier constat que nous pouvons faire est qu'on retrouve une baisse des performances lorsque les données dépassent la taille du cache L3. Cela veut dire qu'à la différence d'un calcul sur un seul thread, ce cas est limité par les performances du cache L3, cela pour les deux codecs.

Pour les deux codecs, les vitesses sont maximales avec 8 threads, et le gain du codec Pyrit représente encore 100 %. En ce qui concerne l'évolution du gain en fonction du nombre de threads et de la quantité de données à traiter, celui-ci semble constant, encore une fois, limité par les performances du cache L3, comme le montre la Figure 4.13.

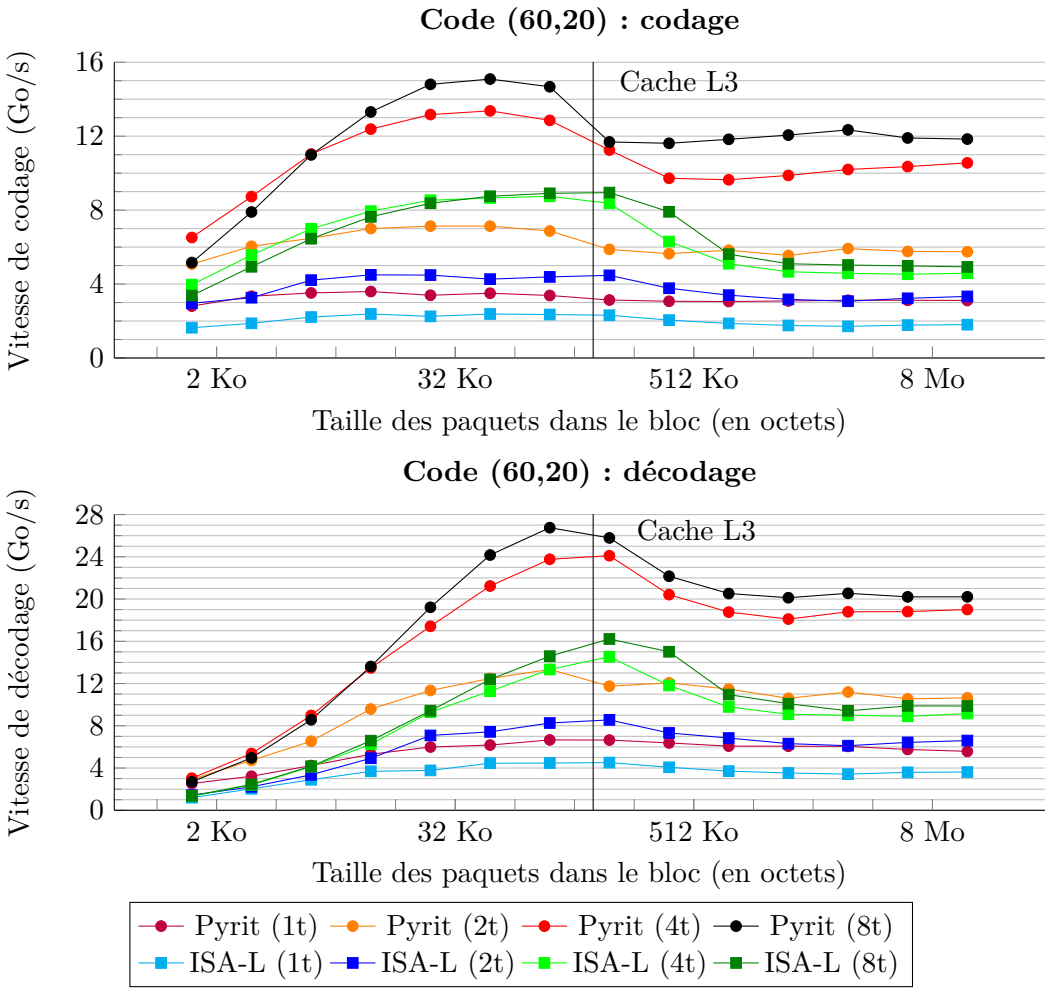


FIGURE 4.12 – Vitesses de codage et de décodage multithread pour un code à effacement de paramètres (60,20)

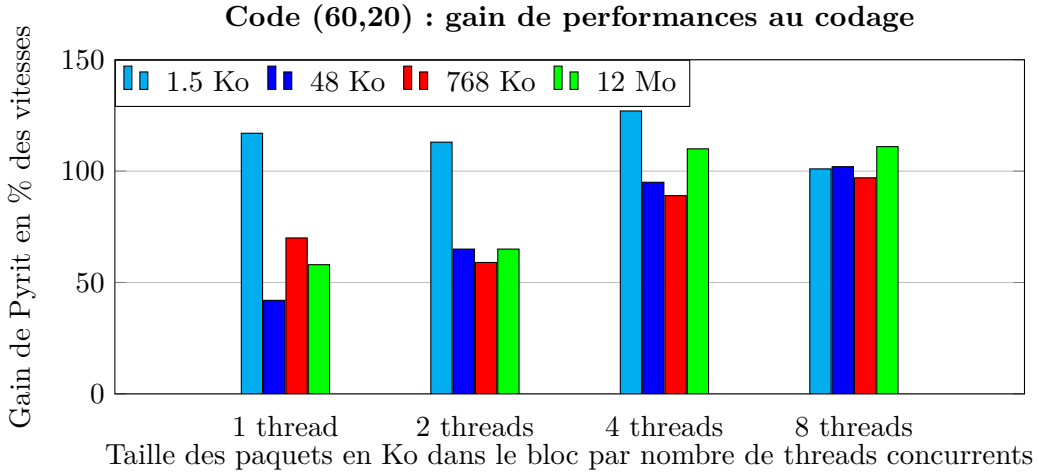


FIGURE 4.13 – Gain de performance du codec Pyrit par rapport au codec d'ISA-L

4.5.2.3 Conclusion

Avec une implémentation exploitant plusieurs coeurs d'exécutions, nous avons pu montrer que le gain du codec Pyrit augmente. Cela s'explique en partie par le fait que le codec Pyrit réalise moins d'accès mémoires que ISA-L car il n'utilise pas de tables de correspondances pour la multiplication. Comme nous pouvons le voir, la performance des deux codecs augmente avec le nombre de threads utilisés, même si Pyrit est toujours plus rapide que ISA-L. En revanche, dans certains cas, lorsque le nombre de threads est supérieur au nombre de coeurs du processeur, le codec ISA-L devient moins performant alors que ce n'est pas le cas pour Pyrit. Ce comportement apparait lorsque le cache L3 est fortement sollicité. On peut penser que ce sera souvent le cas dans l'industrie lorsque ces codecs ne seront qu'une partie de l'application s'exécutant sur les serveurs. Par conséquent, dans ce contexte, le codec Pyrit semble être la bonne approche.

4.5.3 Analyse de performances sur architecture ARM

Afin de compléter l'analyse de performance, nous avons mesuré les vitesses de codage sur un Raspberry pi, équipé d'un processeur ARMv7. En effet, ces processeurs équipent la majorité des smartphones modernes et il nous a donc semblé pertinent de valider ces travaux sur ce genre d'équipement. En revanche, le codec d'ISA-L ne fonctionnant que sur des processeurs de type x86, nous avons choisi de comparer Pyrit aux meilleurs codecs disponibles sur ARM, c'est à dire ceux du projet Jerasure[4]. Ces codecs sont écrits en C et exploitent les instructions SIMD 128 bits sur ARM, appelées Neon. Pour les trois cas d'études, nous avons préalablement testé tous les codecs présents dans Jerasure. Il s'avère que le plus performant pour tous les cas est encore une fois le codec Split table utilisant les instructions SIMD. Nous avons donc porté le code de Pyrit sur ARM en utilisant également les instructions Neon. Et pour être le plus juste dans la comparaison, les deux codecs sont donc écrits en C. En revanche, étant donné la complexité du projet Jerasure, lui-même dépendant du projet GF-complete des mêmes auteurs, nous n'avons pas apporté le support du multithreading. Nous comparerons donc uniquement les performances de codage sur un seul thread. De plus, à la différence des codecs Pyrit et ISA-L, le décodage dans Jerasure est un décodage en 2 temps : d'abord, les paquets d'information sont soustraits des paquets de redondance, ce qui génère une écriture en mémoire, puis après l'inversion de la matrice, l'opération de multiplication matrice vecteur génère à nouveau une écriture mémoire. Ce qui entraine systématiquement un décodage deux fois plus lent que le codage.

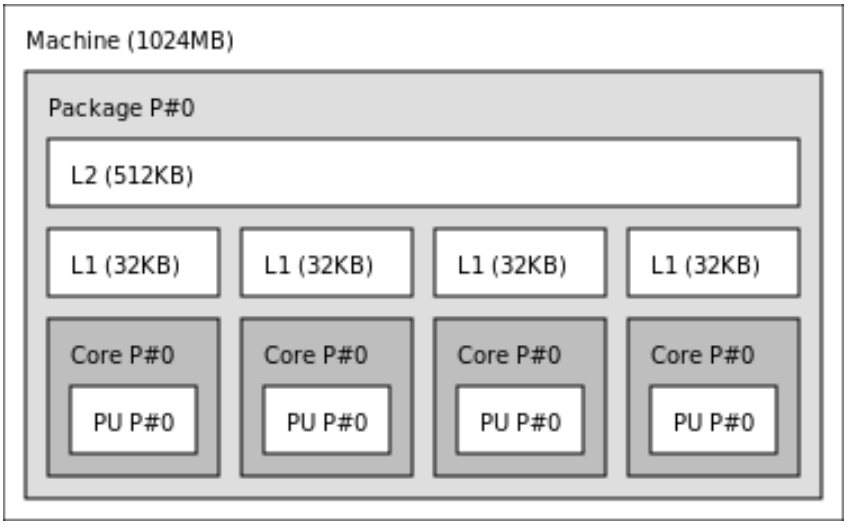


FIGURE 4.14 – Topologie du processeur ARM Cortex-A53 Broadcom BCM2837

La Figure 4.14 représente la topologie du processeur qui équipe le Rapsberry Pi 3. A la différence du processeur x86 utilisé précédemment, on peut constater que ce processeur ne possède que deux niveaux de caches. Ces derniers sont également bien plus petits que ceux sur x86. Chaque coeur possède un cache L1 de taille 32 Ko et un cache de niveau 3 de 512Ko partagé entre tous les coeurs.

4.5.3.1 Le code (12,8)

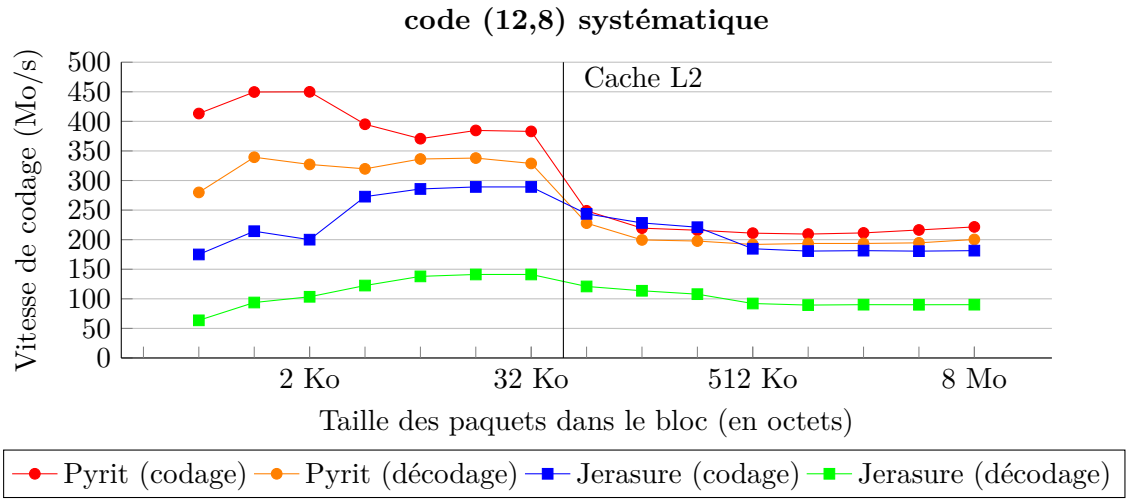


FIGURE 4.15 – Vitesses de codage des codes à effacement MDS en fonction de la taille des données sur Raspberry pi

Pour ce type d'architecture et ces paramètres de code, les gains de Pyrit semblent

moins prononcés que sur x86. On constate cependant une nette dégradation des performances de Pyrit lorsque la taille des données dépasse celle du cache L2, soit 512 Ko. Des opérations d'écritures et de lectures supplémentaires en mémoire sont donc nécessaires, ralentissant considérablement la vitesse d'exécution. Pour des données de très grande taille, les vitesses des deux codecs semblent donc converger. . En effet, le temps de calcul dépend de plus en plus des accès mémoires. Notons toutefois que pour des paquets de taille supérieure à 512 Ko, Pyrit est légèrement plus rapide avec un gain situé entre 20 et 30 Mo/s. Pour ce cas là, le codec Pyrit est donc au moins aussi performant que Jerasure.

4.5.3.2 Le code (60,40)

De la même manière que sur x86, analysons les performances sur ARM d'un codec Pyrit sur $\mathbb{F}_{2^6}$ présentées sur la Figure 4.16 :

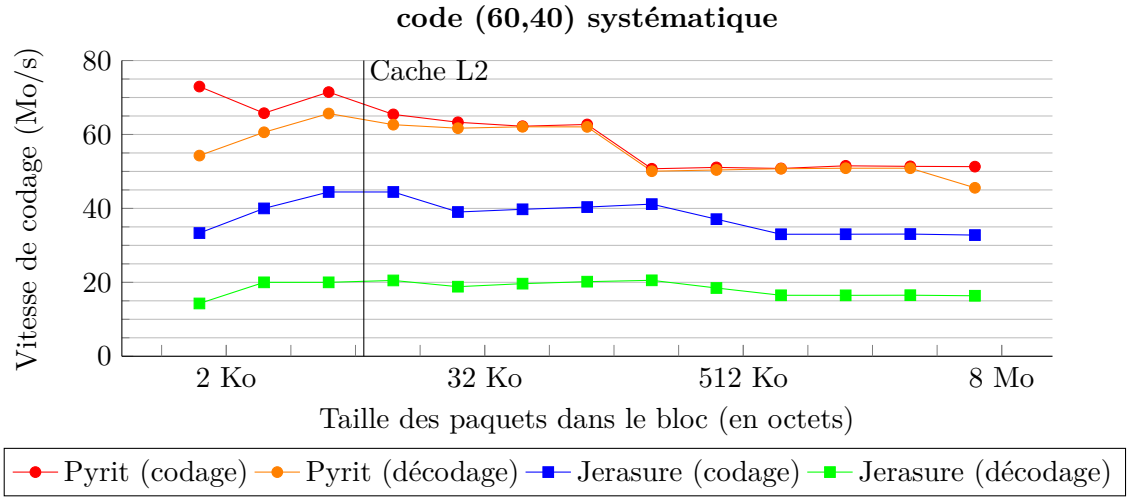


FIGURE 4.16 – Vitesses de codage des codes à effacement MDS en fonction de la taille des données sur Raspberry pi

Pour rappel, Jerasure utilise un décodage en deux passes, générant deux écritures mémoires. Cela explique pourquoi le décodage est deux fois plus lent que le codage. En optimisant le code, il est tout à fait possible d'obtenir les mêmes vitesses. Considérons donc uniquement les vitesses de codage pour Jerasure. Comme sur les processeurs x86, Pyrit réduit le transfert entre la mémoire et le processeur par le fait qu'il n'utilise aucune table de correspondance pour calculer les combinaisons linéaires. Pour ces paramètres, le codec Pyrit permet un gain d'environ 50 % par rapport au codec de Jerasure. Ce gain représente entre 40 et 20 Mo/s, soit le même gain que celui obtenu pour le code $(12, 8)$. On peut donc penser que ce gain équivaut à la réduction du transfert des données entre la mémoire et le processeur.

4.5.3.3 Le code (60,20)

Ce dernier cas permet de valider l'utilisation de la transformée Parity sur un code travaillant sur $\mathbb{F}_{2^6}$.

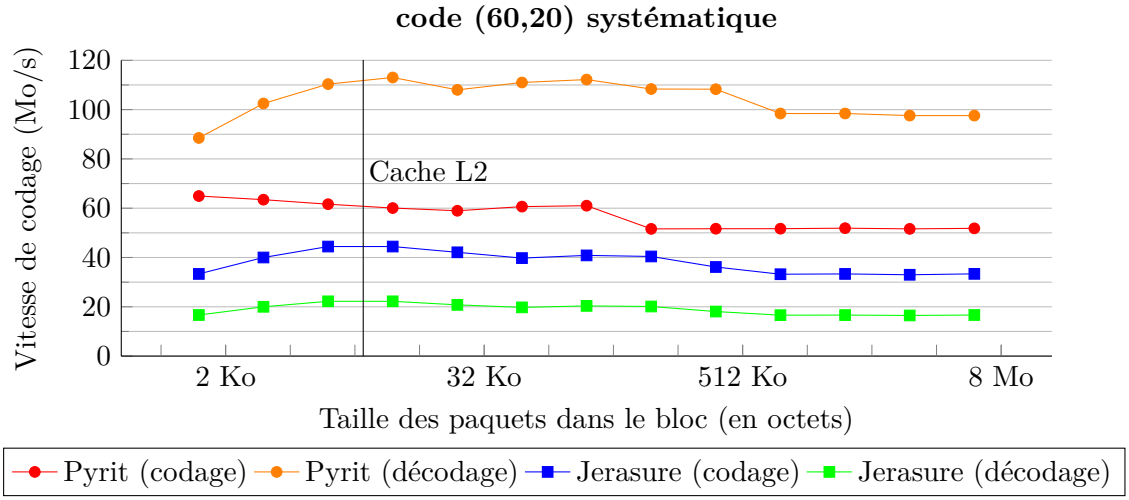


FIGURE 4.17 – Vitesses de codage des codes à effacement MDS en fonction de la taille des données sur Raspberry pi

Le codec Pyrit, avec Parity, présente des performances excellentes face au codec de Jerasure. Encore une fois, sans considérer le décodage en deux étapes de Jerasure, Pyrit permet tout de même un gain de plus de 100 % pour le décodage et 50 % pour le codage. La différence de vitesse entre le codage et le décodage s'explique par le nombre de combinaisons linéaires à calculer. A l'encodage, nous avons $n - k$ combinaisons linéaires à calculer alors qu'au décodage, seulement k paquets ont besoin d'être reconstruits, soit deux fois moins de combinaisons linéaires. Enfin, pour des paquets de petite taille, le décodage est pénalisé par l'inversion de la matrice, ce qui explique la différence de vitesse avec des paquets de plus grande taille.
Le cache L3 semble ne pas avoir d'impact sur les performances. Les performances, pour ce cas là, semblent limitées par les capacités de calcul du processeur, et non le transfert de données depuis la mémoire.

4.6 Conclusion

Dans ce chapitre, nous avons détaillé la construction d'un codec, appelé Pyrit, pour les codes à effacement. Celui-ci fonctionne avec deux corps finis : soit le corps fini $\mathbb{F}_{2^4}$ défini par un polynôme irréductible AOP de degré 4, soit le corps fini $\mathbb{F}_{2^6}$ défini par le polynôme irréductible ESP de degré 6.
Les opérations de codage et décodage entre des éléments du corps fini sont faites dans un anneau polynomial de caractéristique 2, ce qui implique l'utilisation de

transformées. Nous avons montré qu'il est possible d'appliquer les transformées présentées dans le chapitre précédent de manière très rapide, sans jamais stocker dans la mémoire l'information supplémentaire qu'elles génèrent.
Nous avons analysé les performances de ce codec sur deux architectures totalement différentes et trois codes différents, ce qui permet de mettre en avant les avantages des différentes transformées. Dans tous les cas étudiés, le codec Pyrit est plus performant que le meilleur codec disponible pour l'architecture utilisée. Il nous a semblé important d'évaluer les performances en utilisant plusieurs coeurs en parallèle. Dans ce cas là, le gain entre le codec Pyrit et les autres codecs est encore plus important. En effet, comme nous l'avons vu, Pyrit n'utilise pas de tables de correspondances pour calculer les combinaisons linéaires, à la différence des codecs de type "Split-table" de Jerasure ou ISA-L. Ces accès mémoires ralentissent le traitement de l'information par le processeur. Les caches permettent de limiter cet impact. Avec Pyrit, le processeur n'est pas ralenti par les accès mémoire, malgré la présence des caches, et peut donc traiter plus rapidement les données à coder.
Nous avons vu que les gains obtenus dépendent principalement de la topologie des processeurs. Pour les petits codes, c'est à dire lorsque peu de données sont lues depuis la mémoire, le gain de Pyrit n'est pas aussi important que sur les plus grands codes. Le nombre de threads utilisés a également un fort impact sur les performances. Nous avons vu que pour le codec Pyrit, plus le nombre de threads est important, plus les performances augmentent. Pour ISA-L, il semble y avoir une limite à partir de laquelle les performances n'augmentent plus avec le nombre de threads. En effet, comparé à Pyrit, beaucoup plus de données étant lues depuis la mémoire, le cache L3, commun à tous les coeurs du processeur, ne permet pas d'absorber une telle quantité de données. Avec les implémentations présentées dans ce chapitre, pour tout code MDS de longueur $n < 64$, Pyrit est bien le codec le plus performant sur les deux architectures testées.

En revanche, les codecs auxquels sont comparés Pyrit, bien qu'étant les plus rapides pour les codes testés sans considérer Pyrit, permettent de supporter des codes de dimension plus grande en travaillant sur $\mathbb{F}_{2^8}$. En effet, avec ces implémentations, Pyrit ne peut pas travailler avec des codes MDS de longueur $n >= 64$ alors que ISA-L et Jerasure permettent supportent des codes avec $n <= 255$.
Comme il n'existe pas de corps fini défini par un polynome irréductible de degré 8 étant AOP ou ESP, des travaux sont en cours afin d'étudier les performances d'un code utilisant un corps fini défini par un polynome AOP de degré 10 sur des machines équipées de processeurs x86 possédant 32 registres SIMD.
Un autre codec actuellement en cours de développement utilise un corps fini construit "par composition" [21]. En effet, nous nous sommes limités ici à des corps finis de la forme $\mathbb{F}_{p^q}$, avec $p = 2$. Ainsi, chaque multiplication entre deux éléments de ce corps peut se décomposer en une série d'opérations dans $\mathbb{F}_p$, soit $\mathbb{F}_2$, c'est à dire uniquement des *xor*. Mais il est tout à fait possible de travailler dans un corps fini de la forme $\mathbb{F}_{(p^q)^r}$. Autrement dit, ce corps fini est composé de polynomes de degré $r - 1$ et dont les coefficients appartiennent au corps fini $\mathbb{F}_{p^q}$. Ainsi, avec

$p = 2$, $q = 4$ et $r = 2$, il est possible de travailler dans $F = \mathbb{F}_{(2^4)^2}$. Dans ce cas, les combinaisons linéaires sont faites sur un corps fini possédant 2^8 élements et l'opération de multiplication peut se décomposer en plusieurs multiplications dans $\mathbb{F}_{2^4}$. En choisissant correctement le polynome irréductible définissant ce corps fini, ces multiplications peuvent être réalisées dans l'anneau $R_{2,5}$, reprennant certains travaux détaillés dans ce chapitre.

Chapitre 5

Conclusion et perspectives

Dans ce chapitre, nous résumons tout d'abord les principales contributions de ce manuscrit, puis nous donnons quelques perspectives futures avant de conclure sur ces travaux.

Résumé des contributions

Dans cette thèse, l'objectif principal était de réduire la complexité des codes correcteurs d'effacement basés sur les corps finis.
Pour cela, nous avons d'abord présenté une méthode permettant de remplacer les opérations faites dans la multiplication matrice-vecteur effectuée dans un corps fini en transformant les éléments de ce dernier en éléments d'un anneau. Ces opérations, telles que la multiplication et l'addition, sont présentes dans n'importe quel code linéaire basé sur les corps finis. Nous avons commencé par rappeler les différentes classes de polynomes irréductibles permettant de définir les corps finis à partir desquels il est possible de transformer les éléments en éléments d'un anneau. Puis, nous avons défini un ensemble de transformées ayant des propriétés différentes exploitées par les codes à effacement. Ainsi, nous avons montré qu'avec certaines transformées il était possible de réduire la complexité de la multiplication par la matrice génératrice, tandis qu'avec d'autres, il était possible de transformer plus rapidement un grand nombre d'éléments d'un corps fini en éléments de l'anneau et vice versa.
Nous avons aussi démontré la compatibilité de ces transformées en expliquant comment il était possible, en utilisant simultanément plusieurs transformées différentes, de réaliser des multiplications matrice-vecteur dans un anneau à partir d'éléments d'un corps fini.
Bien que l'utilisation de ces transformées rajoute de la complexité, le fait de passer dans l'anneau permet de réaliser l'opération de multiplication beaucoup plus rapidement qu'en restant dans le corps fini.

Nous avons également évalué la complexité en nombre d'opérations de cette méthode dans le processus de codage et de décodage des codes MDS. Les résultats montrent que, simplement par construction, le fait de transformer une matrice génératrice dont les coefficients appartiennent à un corps fini en éléments d'un anneau polynomial permet de réduire drastiquement sa densité et donc le nombre d'opérations à effectuer pour réaliser la multiplication matrice-vecteur.
Une autre conséquence intéressante de la transformation de la matrice est qu'elle

permet d'appliquer un réordonnancement des opérations beaucoup plus facilement que sur des élements d'un corps fini. En effet, la structure cyclique de l'anneau nous permet de factoriser, directement et indirectement, certaines opérations, ce qui réduit considérablement la complexité totale.

Après avoir présenté ces résultats théoriques, nous avons proposé une analyse de performance de notre méthode dans le contexte des codes MDS. Pour cela, à partir des deux classes de polynomes irréductibles permettant de construire un corps fini dont les éléments peuvent être transformés en éléments d'un anneau pour y appliquer la multiplication, nous avons construit deux codecs travaillant sur $\mathbb{F}_{2^4}$ et $\mathbb{F}_{2^6}$. Nous avons présenté les opérations de transformée et montré que celles-ci peuvent s'appliquer de manière très rapide sur des données à coder.
La première analyse a permis de montrer que sur les petits codes, les performances dépendaient principalement des accès mémoires. Par conséquent, le fait de réduire le nombre d'opérations *xor* dans la matrice génératrice binaire ne présente que peu d'intérêt si le transfert d'information par les caches entre le processeur et la mémoire n'est pas également réduit.
Nous nous sommes ensuite intéressés aux vitesses de codage de ces deux codecs sur deux architectures matérielles différentes en les comparant à des codecs permettant la construction de codes MDS. Tous ces codecs sont basés sur la méthode "Split-table", et utilisent des tables précalculées. Que ce soit sur un processeur d'architecture x86 ou ARM, le codec Pyrit est toujours plus performant que les autres codecs avec qui il a été comparé.
Nous avons également montré que dans un contexte d'exécution parallèle du codage, générant beaucoup plus de transferts entre le processeur et la mémoire, plus on utilisait de coeurs de calculs, meilleur était le gain de performances par rapport aux codecs utilisant des tables précalculées. Nous avons également montré que la limite des codecs n'était pas le nombre d'opérations par secondes réalisable par les processeurs, mais plutot la bande passante entre le processeur et la mémoire. Les processeurs ayant de plus en plus de coeurs de calculs, les implémentations de type Pyrit, favorisant le transfert d'information entre la mémoire et le processeur, devraient se montrer de plus en plus performantes avec l'évolution des architectures matérielles étudiées.

Perspectives futures

Bien que cette méthode permettant de transformer les éléments de certains corps finis en éléments d'un anneau soit générale, l'implémentation présentée se limite pour le moment aux corps finis $\mathbb{F}_{2^6}$ et donc aux codes MDS de taille $n < 64$. Il serait donc intéressant d'étudier par la suite les possibilités d'une telle méthode avec des codes de plus grande taille.
Pour cela, nous envisageons deux approches. La première est une extension directe des codecs présentés pour des corps finis plus grands, notamment $\mathbb{F}_{2^{10}}$. La principale

difficulté consistera à adapter ce codec aux processeurs de dernière génération, en utilisant au mieux les différents registres SIMD et en limitant au maximum les transferts de données entre le processeur et la mémoire.
La deuxième approche possible est basée sur la construction "par composition" des corps finis. Par exemple, les opérations dans $\mathbb{F}_{2^{4^2}}$, corps fini à 2^8 éléments, peuvent être décomposées en plusieurs opérations dans $\mathbb{F}_{2^4}$, sur lequel il est possible d'utiliser la méthode de passage dans l'anneau et pour lequel il existe dejà un codec efficace.

Bibliographie

[1] In *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2*, page 1064. (Cité en pages 28 et 34.)

[2] In *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. (Cité en page 80.)

[3] Intelligent storage acceleration library. (Cité en pages 29 et 76.)

[4] Jerasure : Erasure coding library. In *http ://jerasure.org/*. (Cité en pages 76 et 86.)

[5] Leopard-rs : O(n log n) mds reed-solomon block erasure codes. (Cité en page 32.)

[6] The OpenFEC project : http ://www.openfec.org. (Cité en pages 18 et 26.)

[7] Rozo systems : High-performance storage software for the data center and public cloud. (Cité en page 22.)

[8] Rozofs : Scale-out storage using erasure coding. (Cité en page 22.)

[9] P. H. Anvin. The mathematics of RAID-6, 2009. (Cité en page 28.)

[10] M. Blaum. A family of mds array codes with minimal number of encoding operations. In *2006 IEEE International Symposium on Information Theory*, pages 2784–2788, July 2006. (Cité en page 58.)

[11] M. Blaum, J. Bruck, and A. Vardy. Mds array codes with independent parity symbols. *IEEE Transactions on Information Theory*, 42(2) :529–542, March 1996. (Cité en page 58.)

[12] M. Blaum and R. M. Roth. On lowest density mds codes. *IEEE Transactions on Information Theory*, 45(1) :46–59, Jan 1999. (Cité en pages 30, 34 et 57.)

[13] Mario Blaum, Jim Brady, Jehoshua Bruck, and Jai Menon. Evenodd : An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Trans. Computers*, 44 :192–202, 1995. (Cité en pages 30 et 58.)

[14] J. Bloemer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme. In *Technical Report ICSI TR-95-048*, August 1995. (Cité en pages 17, 27, 46 et 57.)

[15] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1) :68 – 79, 1960. (Cité en page 14.)

[16] David G Cantor. On arithmetical algorithms over finite fields. *Journal of Combinatorial Theory, Series A*, 50(2) :285 – 300, 1989. (Cité en page 32.)

[17] Peter Corbett, Bob English, Atul Goel, Tomislav Grcanac, Steven Kleiman, James Leong, and Sunitha Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST'04, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association. (Cité en pages 30 et 58.)

[18] Sylvain David, Pierre Evenou, Jean-Pierre Guedon, Nicolas Normand, and Benoît Parrein. Procédé et appareil permettant de reconstruire un bloc de données. Number WO2015055450A1, 2013. (Cité en page 22.)

[19] P. Elias. Coding for two noisy channels. *Information Theory, The 3rd London Symposium*, pages 61–76, september 1955. (Cité en page 9.)

[20] Robert G. Gallager. Low-density parity-check codes, 1963. (Cité en page 17.)

[21] K. M. Greenan, E. L. Miller, and S. J. Thomas J. E. Schwarz. Optimizing galois field arithmetic for diverse processor architectures and applications. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10, Sept 2008. (Cité en pages 64 et 90.)

[22] Jeanpierre V. Guedon, Dominique Barba, and Nicole Burger. Psychovisual image coding via an exact discrete radon transform. volume 2501, pages 2501 – 2501 – 11, 1995. (Cité en page 20.)

[23] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2) :147–160, april 1950. (Cité en page 14.)

[24] Alexis Hocquenghem. Codes correcteurs d'erreurs. *Chiffres*, 2(2) :147–56, 1959. (Cité en page 14.)

[25] C. Huang, J. Li, and M. Chen. On optimizing xor-based codes for fault-tolerant storage applications. In *2007 IEEE Information Theory Workshop*, pages 218–223, Sept 2007. (Cité en page 31.)

[26] Toshiya Itoh and Shigeo Tsujii. Structure of parallel multipliers for a class of fields $gf(2^m)$. *Information and Computation*, 83(1) :21 – 40, 1989. (Cité en pages 34, 36, 37, 39, 41, 42 et 68.)

[27] jonathan.detchart@isae.fr, Emmanuel Lochin, Jerome Lacan, and Vincent Roca. Tetrys, an On-the-Fly Network Coding protocol. Internet-Draft draft-detchart-nwcrg-tetrys-04, Internet Engineering Task Force, March 2018. Work in Progress. (Cité en page 24.)

[28] R. Koetter and M. Medard. Beyond routing : an algebraic approach to network coding. In *Proceedings.Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 1, pages 122–130 vol.1, 2002. (Cité en page 23.)

[29] J. Lacan and J. Fimes. Systematic mds erasure codes based on vandermonde matrices. *IEEE Communications Letters*, 8(9) :570–572, Sept 2004. (Cité en page 16.)

[30] H. Li and Q. Huan-yan. Parallelized network coding with simd instruction sets. 1 :364–369, Dec 2008. (Cité en pages 28 et 64.)

[31] S. J. Lin, T. Y. Al-Naffouri, Y. S. Han, and W. H. Chung. Novel polynomial basis with fast fourier transform and its application to reed-solomon erasure codes. *IEEE Transactions on Information Theory*, 62(11) :6284–6299, Nov 2016. (Cité en page 32.)

[32] S. J. Lin and W. H. Chung. An efficient (n, k) information dispersal algorithm for high code rate system over fermat fields. *IEEE Communications Letters*, 16(12) :2036–2039, December 2012. (Cité en page 31.)

[33] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, and D. A. Spielman. Efficient erasure correcting codes. *IEEE Transactions on Information Theory*, 47(2) :569–584, Feb 2001. (Cité en page 19.)

[34] Michael Luby. Lt codes. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, FOCS '02, pages 271–, Washington, DC, USA, 2002. IEEE Computer Society. (Cité en page 19.)

[35] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes*. North-Holland, 1977. (Cité en pages 17, 35 et 42.)

[36] Emin Martinian and C-EW Sundberg. Low delay burst erasure correction codes. In *Communications, 2002. ICC 2002. IEEE International Conference on*, volume 3, pages 1736–1740. IEEE, 2002. (Cité en page 22.)

[37] Kazuhisa Matsuzono, Vincent Roca, and Hitoshi Asaeda. Structured random linear codes (SRLC) : bridging the gap between block and convolutional codes. *CoRR*, abs/1408.5663, 2014. (Cité en page 23.)

[38] Nicolas Normand, Andrew Kingston, and Pierre Evenou. A geometry driven reconstruction algorithm for the mojette transform. In *Discrete Geometry for Computer Imagery*, pages 122–133, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cité en page 21.)

[39] J. S. Plank. Xor's, lower bounds and mds codes for storage. In *2011 IEEE Information Theory Workshop*, pages 503–507, Oct 2011. (Cité en pages 31, 55 et 62.)

[40] J. S. Plank, K. M. Greenan, and E. L. Miller. A complete treatment of software implementations of finite field arithmetic for erasure coding applications. Technical Report UT-CS-13-717, University of Tennessee, October 2013. (Cité en pages v, 28 et 29.)

[41] James S. Plank, Kevin M. Greenan, and Ethan L. Miller. Screaming fast galois field arithmetic using intel SIMD instructions. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 298–306, San Jose, CA, 2013. USENIX Association. (Cité en pages 28, 34, 64 et 76.)

[42] A. Poli and L. Huguet. *Error correcting codes : theory and applications*. Prentice Hall, 1992. (Cité en page 35.)

[43] Carl Pomerance and J. W. Smith. Reduction of huge, sparse matrices over finite fields via created catastrophes. *Experimental Mathematics*, 1(2) :89–94, 1992. (Cité en page 23.)

[44] Michael O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *J. ACM*, 36(2) :335–348, April 1989. (Cité en page 17.)

[45] J. Radon. Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten. *Akad. Wiss.*, 69 :262–277, 1917. (Cité en page 20.)

[46] J. Radon. On the determination of functions from their integral values along certain manifolds. *IEEE Transactions on Medical Imaging*, 5(4) :170–176, Dec 1986. (Cité en page 20.)

[47] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2) :300–304, 1960. (Cité en page 14.)

[48] L. Rizzo. Effective Erasure Codes For Reliable Computer Communication Protocols. *ACM Computer Communication Review*, 27(2) :24–36, April 1997. (Cité en page 26.)

[49] Ron M Roth and Gadiel Seroussi. On generator matrices of mds codes. *IEEE Trans. Inf. Theor.*, 31(6) :826–830, November 1985. (Cité en page 46.)

[50] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27 :623–656, 1948. (Cité en page 5.)

[51] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6) :2551–2567, June 2006. (Cité en page 20.)

[52] R. Singleton. Maximum distance q-nary codes. *Information Theory, IEEE Transactions*, pages 116–118, april 1964. (Cité en page 14.)

[53] N. J. A. Sloane. A001122 sequence. In *https ://oeis.org/A001122*, 1991. (Cité en page 36.)

[54] A. Soro and J. Lacan. Fnt-based reed-solomon erasure codes. In *2010 7th IEEE Consumer Communications and Networking Conference*, pages 1–5, Jan 2010. (Cité en page 31.)

[55] Chris Studholme and Ian F. Blake. Random matrices and codes for the erasure channel. *Algorithmica*, 56(4) :605–620, Apr 2010. (Cité en page 22.)

[56] Pierre-Ugo Tournoux, Emmanuel Lochin, Jérôme Lacan, Amine Bouabdallah, and Vincent Roca. On-the-fly erasure coding for real-time video applications. *IEEE Transactions on Multimedia*, 13(4) :797–812, March 2011. (Cité en page 25.)

[57] P.K.S. Wah and M.Z. Wang. Realization and application of the massey-omura lock. In *Proceedings, International Zurich Seminar*, pages 175–185, 1984. (Cité en page 36.)

Contributions

Publications scientifiques

— **Jonathan Detchart** and Jérôme Lacan, PYRIT : Polynomial Ring Transforms for Fast Erasure Coding, Usenix FAST'17 Work-in-Progress Reports (WiPS), Santa Clara US, 2017
— **Jonathan Detchart** and Jérôme Lacan, Fast Xor-based Erasure Coding based on Polynomial Ring Transforms, 2017 IEEE International Symposium on Information Theory, ISIT 2017, Aachen, Germany, June 25-30, 2017
— **Jonathan Detchart** and Jérôme Lacan, Improving the Coding Speed of Erasure Codes with Polynomial Ring Transforms, 2017 IEEE Global Communications Conference, GLOBECOM 2017, Singapore, December 4-8, 2017
— Doriane Perard and Jérôme Lacan and Yann Bachy and **Jonathan Detchart**, Erasure code-based low storage blockchain node, IEEE Conference on blockchain, 2018

Brevets

— **Jonathan Detchart**, Jérome Lacan, Emmanuel Lochin, Codage et décodage correcteur d'erreurs par matrice génératrice avec multiplications simplifiées dans le corps de galois, 2018, WO 2018/109346 Al
— **Jonathan Detchart**, Jérôme Lacan, Emmanuel Lochin, point-to-point transmitting method based on the use of an erasure coding scheme and a TCP/IP protocol - EU PATENT Application number 17305951

Logiciels

Codec Pyrit :
— code optimisé utilisant SSE ou AVX pour les architectures x86_64 et Neon pour les architectures ARM ;
— codec fonctionnant sur $\mathbb{F}_{2^4}$ et $\mathbb{F}_{2^6}$;
— support du multithreading pour toutes les méthodes de codage ;
— travaux d'optimisations en cours pour le codec fonctionnant sur $\mathbb{F}_{2^8}$;

Documents de standardisation IETF

— Jonathan Detchart, Emmanuel Lochin, Jerome Lacan, Vincent Roca, Tetrys, an On-the-Fly Network Coding protocol, IRTF NWC Research Group, draft-detchart-nwcrg-tetrys-04 (Work in Progress), March 2018
— Vincent Roca, Jonathan Detchart, Cédric Adjih, Morten Pedersen, Generic API for Sliding Window FEC Codes, IRTF NWC Research Group, draft-roca-nwcrg-generic-fec-api-02 (Work in Progress), July 2018

Présentations à des groupes de recherche

— Jonathan Detchart, Pyrit : Polynomial Ring Transforms for Fast Erasure Coding, IRTF NWC Research Group, slides-99-nwcrg-06-detchart-pyrit, July 2017

Résumé

Les codes correcteurs d'effacements sont aujourd'hui une solution bien connue utilisée pour fiabiliser les protocoles de communication ou le stockage distribué des données. La plupart de ces codes sont basés sur l'arithmétique des corps finis, définissant l'addition et la multiplication sur un ensemble fini d'éléments, nécessitant souvent des opérations complexes à réaliser. En raison de besoins en performance toujours plus importants, ces codes ont fait l'objet de nombreuses recherches dans le but d'obtenir de meilleures vitesses d'exécution, tout en ayant la meilleure capacité de correction possible.
Nous proposons une méthode permettant de transformer les éléments de certains corps finis en éléments d'un anneau afin d'y effectuer toutes les opérations dans le but de simplifier à la fois le processus de codage et de décodage des codes correcteurs d'effacements, sans aucun compromis sur les capacités de correction. Nous présentons également une technique de réordonnancement des opérations, permettant de réduire davantage le nombre d'opérations nécessaires au codage grâce à certaines propriétés propres aux anneaux utilisés. Enfin, nous analysons les performances de cette méthode sur plusieurs architectures matérielles, et détaillons une implémentation simple, basée uniquement sur des instructions *xor* et s'adaptant beaucoup plus efficacement que les autres implémentations à un environnement d'exécution massivement parallèle.

Mots clés : codes à effacement, transformées, corps fini, optimisation, implémentation

Abstract

Erasure codes are widely used to cope with failures for nearly all of today's networks communications and storage systems. Most of these codes are based on finite field arithmetic, defining the addition and the multiplication over a set of finite elements. These operations can be very complex to perform. As a matter of fact, codes performance improvements are still an up to date topic considering the current data growth explosion.
We propose a method to transform the elements of some finite fields into ring elements and perform the operations in this ring to simplify both coding and decoding of erasure codes, without any threshold on the correction capacities. We also present a scheduling technique allowing to reduce the number of operations thanks to some particular properties of the ring structure. Finally, we analyse the performance of such a method considering several hardware architectures and detail a simple implementation, using only *xor* operations, fully scalable over a multicore environment.

Keywords : erasure codes, transforms, finite field, optimization, implementation