Académie de Montpellier

Université Montpellier II

— Sciences et Techniques du Languedoc —

Thèse

présentée au Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier

Spécialité	:	**Informatique**
Formation Doctorale	:	**Informatique**
École Doctorale	:	**Information, Structures, Systèmes**

Arithmétique modulaire pour la cryptographie

par

Thomas Plantard

M. Michel Robert, Professeur, Université de Montpellier 2 .... Président
M. Philippe Langevin, Professeur, Université de Toulon .... Examinateur
M. Dominique Michelucci, Professeur, Université de Bourgogne .... Rapporteur
M. Robert Rolland, Maître de conférence, Université de Marseille 2........Rapporteur
M. Jean-Claude Bajard, Professeur, Université de Montpellier 2..............Directeur
M. Laurent Imbert, Chargé de recherche CNRS, LIRMM.....................Encadrant

Avant-Propos

Cette thèse porte sur l'arithmétique modulaire. L'arithmétique modulaire est utilisée dans différents contextes : traitement du signal, algorithmique, cryptologie ... Dans cette thèse, nous nous focalisons sur l'arithmétique modulaire pour la cryptographie. En particulier, nous essayons de voir ce qui peut être fait pour optimiser les différents protocoles de la cryptographie à clé publique.

Cette thèse se divise en quatre chapitres.

La premier chapitre est une introduction à la cryptographie à clé publique. Cette thèse ayant pour objectif d'améliorer les calculs modulaires, nous commençons par présenter dans quels contextes la cryptographie nécessite une arithmétique modulaire efficace. Les principaux protocoles (de ECC ou RSA) ont des besoins en arithmétiques modulaires.

Le deuxième chapitre est un état de l'art sur les différents algorithmes existants pour effectuer une arithmétique modulaire complète : addition, inversion et multiplication. Nous y répertorions aussi les différentes classes de moduli existantes. Celle ci sont utilisées en particulier pour l'implantation des protocoles d'ECC.

Dans le troisième chapitre, nous proposons un nouveau système de représentation. Nous y exposons les outils algorithmiques pour effectuer une arithmétique optimisée pour une classe de moduli, ainsi que ceux nécessaires à une arithmétique modulaire pour tous les autres moduli.

Dans le dernier chapitre, nous regroupons plusieurs résultats concernant l'arithmétique modulaire pour de "petits" moduli. Nous proposons un algorithme de multiplication modulaire pour modulo variable, une amélioration des changements de base pour le RNS ainsi qu'une algorithmique basée sur une table mémoire.

Table des matières

1 Besoins arithmétiques en cryptographie **7**
1.1 Principes généraux . . . 7
1.2 L'algorithme RSA . . . 9
1.3 Les protocoles basés sur les courbes elliptiques . . . 11
1.4 Besoins arithmétiques . . . 15

2 État de l'art sur l'arithmétique modulaire **17**
2.1 L'addition modulaire . . . 19
2.1.1 L'algorithme basé sur la retenue sortante d'Omura . . . 19
2.1.2 L'algorithme de Takagi en représentation redondante . . . 19
2.2 L'inversion modulaire . . . 21
2.2.1 Via l'algorithme d'Euclide étendu . . . 21
2.2.2 Via le théorème de Fermat . . . 23
2.3 La multiplication modulaire . . . 26
2.3.1 L'algorithme de Taylor avec mémorisation . . . 27
2.3.2 L'algorithme de Blakley . . . 28
2.3.3 L'algorithme de Montgomery . . . 28
2.3.4 L'approximation du quotient par Barrett . . . 31
2.3.5 L'utilisation d'une écriture redondante par Takagi . . . 33
2.3.6 Étude comparative . . . 33
2.4 La famille des nombres de Mersenne . . . 34
2.4.1 Les nombres de Mersenne . . . 35
2.4.2 La classe dense des nombres Pseudo Mersenne . . . 36
2.4.3 Une vision polynomiale : les nombres de Mersenne Généralisés . . . 41
2.4.4 Une extension des nombres de Mersenne Généralisés . . . 44
2.4.5 Étude comparative . . . 44

3 Systèmes de représentation adaptés **47**
3.1 Définition et propriétés des systèmes de représentation adaptés . . . 48
3.1.1 Les systèmes de numération simple de position . . . 48
3.1.2 Les systèmes de représentation modulaire . . . 49
3.1.3 Les systèmes de représentation adaptés . . . 52
3.2 Algorithmique sur les systèmes de représentation adaptés . . . 53
3.2.1 La multiplication . . . 53
3.2.2 L'addition . . . 55
3.2.3 Les conversions . . . 56
3.2.4 Principe de réduction des coefficients . . . 57

3.3 Une classe de moduli optimisée . . . 59
3.3.1 Principe de réduction interne . . . 59
3.3.2 Minimisation de la matrice de réduction . . . 63
3.3.3 Création d'un système de représentation adapté optimisé . . . 64
3.3.4 Densité de la classe de moduli . . . 65
3.3.5 Exemple dans le système de représentation adapté $\mathcal{B}_{256}$ . . . 67
3.4 Généralisation des systèmes de représentation adaptés . . . 73
3.4.1 Les réseaux euclidiens . . . 73
3.4.2 Théorème fondamental des systèmes de représentation adaptés . . . 76
3.4.3 Utilisation de tables mémoire . . . 82
3.4.4 Transcription d'algorithmes connus dans les systèmes de représentation adaptés 85

4 Arithmétique modulaire pour de petits moduli 89
4.1 Multiplication modulaire à précalculs bornés . . . 90
4.1.1 Approximation du quotient . . . 91
4.1.2 Évaluation de l'erreur sur le quotient . . . 92
4.1.3 Une nouvelle multiplication modulaire avec approximation du quotient . . . 94
4.1.4 Transformation de la division précalculée en suite de multiplications . . . 95
4.1.5 Hiérarchisation des moduli en fonction du coûts de leurs précalculs . . . 98
4.2 Étude du changement de base en RNS . . . 98
4.2.1 Le changement de base en RNS . . . 99
4.2.2 Des bases RNS optimisées . . . 100
4.2.3 Une nouvelle multiplication modulaire par un inverse . . . 101
4.2.4 La multiplication modulaire en représentation RNS . . . 103
4.3 Carré et multiplication modulaire via une table des inverses . . . 106
4.3.1 La décomposition de la multiplication en inversion . . . 106
4.3.2 L'utilisation d'une table mémoire . . . 106

A Moduli standards de la cryptographie 113
A.1 Moduli conseillés par le NIST . . . 113
A.2 Moduli conseillés par le SEC . . . 113

B Systèmes de représentations adaptés à cryptographie 115
B.1 Proposition de systèmes de représentation adaptés à différentes tailles de moduli . . 115
B.1.1 Modulo pour 128 bits . . . 115
B.1.2 Modulo pour 160 bits . . . 116
B.1.3 Moduli pour 192 bits . . . 116
B.1.4 Modulo pour 224 bits . . . 117
B.1.5 Modulo pour 256 bits . . . 117
B.1.6 Moduli pour 288 bits . . . 118
B.1.7 Moduli pour 320 bits . . . 119
B.1.8 Modulo pour 352 bits . . . 120
B.1.9 Moduli pour 384 bits . . . 120
B.1.10 Modulo pour 416 bits . . . 122
B.1.11 Moduli pour 448 bits . . . 123
B.1.12 Moduli pour 480 bits . . . 124
B.1.13 Modulo pour 512 bits . . . 126
B.2 Systèmes de représentation adaptés avec $X^n - 2$ et $\xi = X^i + 1$ . . . 126

Chapitre 1

Besoins arithmétiques en cryptographie

Sommaire

1.1 Principes généraux 7
1.2 L'algorithme RSA 9
1.3 Les protocoles basés sur les courbes elliptiques 11
1.4 Besoins arithmétiques 15

Cette thèse a pour objet l'étude des opérateurs modulaires pour la cryptographie. Les protocles de cryptographie à clé publique possèdent de forts besoins en arithmétique modulaire. Après un rappel des principes fondamentaux de la cryptographie , nous présentons les deux protocoles les plus utilisés en cryptographie à clé publique : l'algorithme de cryptage asymétrique RSA [58] est basé sur la difficulté du problème de la factorisation. Les courbes elliptique [36] permettent de construire des groupes sur lesquels le problème du logarithme discret est difficile : ceci permet d'implanter des protocoles "sûrs" sur ces groupes. Nous terminons ce chapitre par une présentation des besoins en arithmétiques modulaires qui ressortent de ces différents protocoles cryptographiques.

1.1 Principes généraux

La cryptographie est l'art de modifier une information pour qu'elle ne soit compréhensible que des personnes autorisées.

La cryptographie à clé secrète est basée sur l'échange préalable d'une clé secrète commune. Cette clé a une taille variable ; elle peut être sous la forme d'un chiffre, d'une lettre, d'un livre de code ... Avec cette clé commune, les communicants peuvent crypter ou décrypter un message. Aujourd'hui, de nombreux algorithmes de cryptage à clé secrète sont utilisés : DES [49], triple DES [49], AES [51] ... Ces algorithmes sont basés sur des permutations et des transformations de blocs et n'utilisent pas d'arithmétique modulaire.

Les algorithmes de cryptage à clé privée nécessite un échange de clé préalable. Jusque dans les années 1970, cet échange de clé était la principale difficulté de la cryptographie à clé secrète. En 1970, W. Diffie and M. E. Hellman proposent un moyen fiable d'échanger une clé de façon publique [23] : c'est le début de la cryptographie à clé publique. La cryptographie à clé publique permet d'échanger des informations de façon sécurisée par un canal qui ne l'est pas. La cryptographie à clé publique se différencie de la cryptographie à clé secrète par le fait qu'elle ne doit nécessiter aucun

échange secret préalable. La cryptographie à clé publique peut paraître peu intuitive : l'échange sécurisé d'une information secrète par des communications publiques et sans avoir au préalable échangé la moindre information parait illusoire. Pour illustrer la cryptographie à clé publique, il parait pertinent d'exposer ce simple protocole d'échanges pour introduire son principe.

Algorithme 1 : Protocole d'échanges standard

Input : Alice possède un cadeau
Output : Bob possède le cadeau d'Alice
begin

Alice	Bob
Alice met son cadeau dans un coffre	
Alice ferme le coffre avec son cadenas A	
Alice envoie le coffre à Bob	
	Bob rajoute son propre cadenas B sur le coffre
	Bob envoie le coffre à Alice
Alice enlève son cadenas A du coffre	
Alice envoie le coffre à Bob	
	Bob enlève son cadenas du coffre
	Bob récupère le cadeau

end

Ce simple protocole démontre bien la possibilité d'existence de véritables protocoles cryptographiques à clé publique. Il faut noter qu'aucun échange préalable n'a été nécessaire et surtout qu'à aucun moment le coffre n'a été transporté sans être sécurisé par un cadenas. Ce cadenas correspond à une *fonction trappe*. Une fonction trappe est une opération facile d'en un sens (fermeture du cadenas) et difficile dans un autre (ouverture du cadenas).

Le premier algorithme à clé publique a été proposé par Diffie et Hellman en 1976 [23]. Diffie et Hellman utilisent l'exponentiation modulaire comme fonction trappe. L'exponentiation modulaire est bien une fonction trappe : calculer $g^a \bmod p$ est "facile" mais retrouver a à partir de $g^a \bmod p$ l'est beaucoup moins.

Algorithme 2 : Algorithme d'échange de clés de Diffie Hellman

Input : public : p et un générateur g de $\mathbb{Z}_p^*$
Output : privé : une clé K secrète et commune
begin

Alice	Bob
Alice choisit un entier a	Bob choisit un entier b
Alice envoie $A = g^a \bmod p$ à Bob	Bob envoie $B = g^b \bmod p$ à Alice
Alice calcule $K_a = B^a \bmod p$	Bob calcule $K_b = A^b \bmod p$

end

Nous pouvons vérifier que le secret K créé est bien commun.

$$K = K_a = B^a \bmod p = g^{ab} \bmod p = A^b \bmod p = K_b$$

Exemple 1 *Observons un exemple sur le tableau 1.1. Nous prendrons le corps $\mathbb{Z}_{23}^*$ dont un des générateurs est $g = 5$.*

Alice et Bob ont pu créer une clé secrète $K = 12$.

Une remarque importante est qu'un attaquant doit être capable pour retrouver la clé secrète commune à Alice et Bob de trouver g^{ab} à partir de g, g^a, g^b et p.

Protocole Diffie Hellman	
Alice choisit $a = 3$	Bob choisit une entier $b = 14$
Alice envoie $A = 5^3 \bmod 23 = 10$ à Bob	Bob envoie $B = 5^{14} \bmod 23 = 13$ à Alice
Alice calcule $K = 13^3 \bmod p = 12$	Bob calcule $K = 10^{14} \bmod p = 12$

Tab. 1.1 – Exemple d'échange de clés avec Diffie Hellman.

Le problème du log discret est de trouver l'entier x à partir de g^x sur un groupe fini. Le problème du log discret est reconnu difficile sur un *groupe générique*. Un groupe générique est un groupe sur lequel on ne peut utiliser que les propriétés définissant un groupe. Un des algorithmes les plus connus pour résoudre le log discret sur un groupe quelconque est l'algorithme rho de Pollard [56]. Cet algorithme a une complexité dite exponentielle [1] en $O(\sqrt{p})$ où p est la taille de ce groupe. Cet algorithme a été prouvé optimal pour un groupe générique [63].

Le problème de Diffie Hellman est reconnu dans certains cas aussi difficile que le problème du *logarithme discret* [43, 44]. On ne connait aucun exemple où le problème de Diffie Hellman est “facile” et où le problème du log discret est “difficile”. Cette complexité aurait pu nous permettre de savoir sur quelle taille de modulo l'on doit travailler pour avoir une sécurité raisonnable.

Certaines attaques sous-exponentielles ont pu être mises au point en profitant de certaines propriétés de ce groupe [1] ; le corps $\mathbb{Z}_p^*$ n'est pas un groupe générique.

La sécurité demandée en ce moment est de 2^{80} opérations. Pour savoir quelles tailles de corps sont utilisables pour que le protocole de Diffie Hellman soit “sûr”, il faut en fait choisir un corps sur lequel les meilleurs attaques utilisent au moins 2^{80} opérations. Pour avoir une telle sécurité, il faut opérer sur des moduli p de taille n au moins 1000 bits. Pour résumer, il faut donc être capable de faire des exponentiations modulo un nombre premier p de taille au moins 1000 bits. Le modulo p n'a pour obligation que d'être premier et peut être choisi pour rendre ces opérations arithmétiques efficaces (les nombres de Mersenne par exemple...).

1.2 L'algorithme RSA

RSA est sûrement le plus connu des algorithmes cryptographiques à clé publique. En proposant ce protocole en 1978 [58], Rivest Shamir et Adelman ouvrent un nouveau pan de la cryptographie : la *cryptographie asymétrique*.

La cryptographie asymétrique se nomme ainsi car elle est à sens unique. La cryptographie symétrique utilise une seule clé pour crypter comme pour décrypter. La cryptographie asymétrique utilise une clé (publique) pour crypter et une autre clé (secrète) pour décrypter.

L'idée est que Bob crée deux clés qui lui sont associées. Une clé publique est utilisée par Alice ou toute personne qui désire lui envoyer un message. Cette clé est associée à un protocole et, puisqu'elle est publique, toute personne peut crypter un message pour l'envoyer à Bob. Une seconde clé, secrète et connue uniquement par Bob, permet à Bob d'utiliser une fonction trappe pour décrypter les messages qu'il a reçus. En fait, après avoir crypté un message avec la clé publique, Alice n'est pas capable de décrypter le message qu'elle a crypté. C'est ce qui rend sûr la cryptographie asymétrique et c'est ce qui permet à Bob de publier sa clé publique : elle ne permet pas de décrypter le message.

Pour reprendre l'exemple des cadenas, nous pourrions dire que Bob laisse en libre accès un cadenas B dont lui seul à la clé b. Ainsi quand Alice veut lui envoyer un cadeau, elle récupère le cadenas de Bob. Elle peut alors fermer le coffre dans lequel elle a rangé son cadeau avec le cadenas de Bob. Lorsque Bob reçoit le coffre, il peut l'ouvrir sans problème.

---

[1] Si l'on note n la taille de p en base 2 ($n = log_2(p)$), on obtient comme complexité $O(e^{\frac{1}{2}n})$.

La cryptographie asymétrique a de nombreux avantages. Premièrement, il suffit de connaître la clé publique pour pouvoir crypter un message pour Bob. Tout le monde peut envoyer des message sécurisés à Bob en récupérant simplement sa clé publique. De plus, il n'y a plus besoin d'échange comme pour Diffie-Hellman. Alice n'attend pas que Bob lui réponde pour lui envoyer des messages : Bob n'est pas actif durant l'envoi. Il peut décrypter plus tard tous les messages qui lui ont été envoyés. Enfin, le nombre global de clés chute. La cryptographie classique utilisait une clé par couple de communiquants. Potentiellement un groupe de n personnes devait posséder $n-1$ clés par personne soit $n(n-1)$ clés à mémoriser au total. Avec la cryptographie asymétrique, ce nombre se réduit à deux clés par personne, soit $2n$ clés en tout.

Rivest, Shamir et Adelman ont proposé un protocole basé sur ce principe de cryptographie à clé publique. Ils utilisent comme fonction trappe la factorisation ; il est "facile" de faire la multiplication de deux nombres p et q mais "difficile" de factoriser un produit pq pour récupérer les entiers p et q.

La première étape est donc la création de la clé publique de Bob.

Algorithme 3 : Création de clé publique pour RSA

Output : Bob possède une clé publique (n, e) et une clé privée b
begin
  Bob choisit deux grands nombres premiers p et q
  Bob calcule $n = pq$ et $\phi(n) = (p-1)(q-1)$
  Bob choisit un entier e premier avec $\phi(n)$
  Bob calcule b l'inverse de e modulo $\phi(n)$
end

Le nombre $\phi(n)$, appelé *indicateur d'Euler*, est le nombre d'entiers premiers avec n compris entre 1 et $n-1$.

Bob peut désormais proposer sa clé publique qui est la paire d'entiers (n, e). Sa clé privée est b. La sécurité du protocole RSA vient du fait que seul Bob connaît la factorisation de n. Il est le seul à pouvoir calculer $\phi(n)$ et par conséquent à pouvoir inverser e pour obtenir sa clé privée b.

Désormais, toute personne ayant accès à la clé publique de Bob peut lui envoyer des messages sécurisés. A aucun moment, Alice n'est intervenue dans la création de la clé publique de Bob. Pourtant elle peut désormais lui envoyer un message sécurisé selon le protocole de RSA.

Algorithme 4 : Protocole RSA

Data : Alice connaît le clé publique de Bob (n, e) et Bob a calculé sa clé privée b
Input : Alice possède un message m avec $m < n$
Output : Bob possède ce message m
begin
  Alice calcule $c = m^e \bmod n$
  Alice envoie c à Bob
  Bob calcule $m = c^b \bmod n$
end

Exemple 2 *Bob commence par créer sa clé publique. Il n'effectue cette opération qu'une seule fois pour sécuriser la totalité de ces communications. Pour ceci il utilise l'algorithme 3 comme nous pouvons le voir dans le tableau 1.2.*

Maintenant qu'il a créé sa clé publique $(n = 713, e = 7)$*, Bob la rend publique et conserve secrète sa clé privée* $d = 283$.

En utilisant la clé publique $(n = 713, e = 7)$ *de Bob, Alice crypte son message* $m = 123$ *en*

Création de la clé publique
Bob choisit deux grands nombres premiers $p = 31$ et $q = 23$.
Bob calcule $n = pq = 713$ et $\phi(n) = (p-1)(q-1) = 660$
Bob choisit $e = 7$ avec $pgcd(7, 660) = 1$
Bob calcule $b = e^{-1} \bmod \phi(n) = 283$

Tab. 1.2 – Exemple de création de clé publique pour RSA.

utilisant l'algorithme 4. Elle envoie son message crypté à Bob. Bob le décrypte en utilisant sa clé secrète (voir le tableau 1.3).

Protocole RSA	
Alice calcule $c = 123^7 \bmod 713 = 495$	
Alice envoie $c = 495$ à Bob	
	Bob calcule $m = 495^{283} \bmod 713 = 123$

Tab. 1.3 – Exemple d'utilisation du protocole RSA.

Un attaquant doit décrypter le message m à partir de $n, e, m^e \bmod n$. Si l'attaquant sait factoriser $n = pq$ en p et q, il retrouve sans problème le message m. La factorisation donne une première majoration de la sûreté du protocole RSA. Ce problème a été choisi par Rivest, Shamir et Adelman comme fonction trappe car il est "difficile".

Mais il existe d'autres attaques contre le protocole RSA. Certains cryptanalistes profitent des différentes propriétés de RSA (factorisation par deux premiers uniquement, e "petit"...) pour créer des attaques sous exponentielles [40].

Il faut à ce jour utiliser RSA sur des nombres n composites de taille à peu près 1024 bits pour que les meilleures attaques soient obligées d'effectuer les 2^{80} opérations qui sont recommandées pour considérer un cryptosystème comme "sûr". Le nombre n n'est pas libre, c'est à dire qu'il est fixé par le protocole RSA. Il ne peut donc pas être choisi pour accélerer les calculs modulaires.

Pour être sûr que le protocole RSA soit efficace, l'exponentiation modulo un nombre n composite fixé de taille 1024 bits doit être rapide.

1.3 Les protocoles basés sur les courbes elliptiques

Le protocole de Diffie Hellman [23], ainsi que celui d'ElGamal [25] fonctionnent sur n'importe quel groupe. Le groupe $\mathbb{Z}_p^*$ offre certaines propriétés qui ont pu être utilisées par les cryptanalistes pour attaquer le log discret sur ce groupe de façon sous exponentielle. Il est nécessaire de trouver de nouveaux groupes sur lesquels les cryptanalistes ne peuvent pas utiliser d'autres propriétés que celles définies sur un groupe générique. Les cryptanalistes sont alors obligés d'utiliser l'algorithme rho de Pollard. Celui ci possède une complexité suffisamment grande pour satisfaire de "bons" critères de sécurité. La recherche d'un groupe générique est devenue capitale. Dans cette quête, Koblitz propose en 1985 le groupe des points d'une courbe elliptique [36]. C'est le début de la cryptographie sur les courbes elliptiques, ECC pour "Elliptic Curve Cryptography". Celle ci est depuis très étudiée [20, 65, 37, 30, 64].

Définition 1 *Une courbe elliptique a pour équation générale (forme de Weierstrass) :*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{1.1}$$

Lorsque les coordonnées (x, y) sont définies sur un corps de caractéristique strictement supérieur à 3, alors on peut réécrire l'équation de la courbe sous la forme :

$$y^2 = x^3 + ax + b \tag{1.2}$$

L'intérêt en cryptographie des courbes elliptiques est que si on ajoute le point à l'infini aux points de la courbe, les points d'une courbe elliptique forment un groupe abélien [36, 65, 64].

a) Le point à l'infini est l'élément neutre du groupe.
b) L'opposé d'un point est son symétrique par rapport à l'axe des abscisses.
c) Dans [64], J. H. Silverman démontre qu'il existe une loi de groupe associative qui à deux points associe un troisième point.

Sur la figure 1.1, on propose une exemple de courbe elliptique sur lequel on désire additionner deux points P et Q.

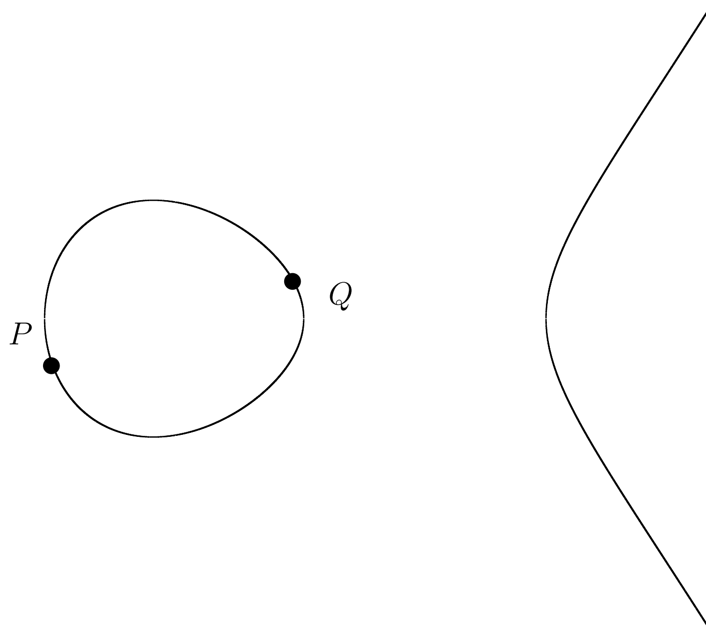

Fig. 1.1 – Exemple d'une courbe elliptique E.

Pour cela, il faut tracer la droite qui passe par ces deux points. Cette droite coupe la courbe en un troisième point, comme nous pouvons le voir sur la figure 1.2.

Il faut alors prendre son point symétrique par rapport à l'axe des abscisses pour obtenir un point R tel que $R = P + Q$ sur le groupe des points de la courbe (voir figure 1.3). Le point symétrique par rapport à l'axe des abscisses est en fait l'opposé d'un point dans le groupe.

Si l'on veut doubler un point, il faut choisir pour droite la tangente de la courbe en ce point et poursuivre le même protocole (figure 1.4).

Les coordonnées sont dans un corps de base. Les corps de base des protocoles de ECC sont soit des corps premiers soit des extensions de corps premiers. Un corps premier est un corps fini $\mathbb{Z}/p\mathbb{Z}$ où p est premier. L'extension de degré k d'un corps premier est noté $(\mathbb{Z}/p\mathbb{Z})^k$. De nombreuse implantations utilisent le corps $(\mathbb{Z}/2\mathbb{Z})^k$. Mais pour plus de clarté, nous utilisons le corps de base $\mathbb{R}$ pour l'exemple de la figure 1.1.

Pour calculer les coordonnées exactes du point R, nous utilisons l'équation de la courbe et celle de la droite. Ainsi, nous pouvons directement calculer les coordonnées de R. Nous évaluons le coefficient directeur λ de la droite.

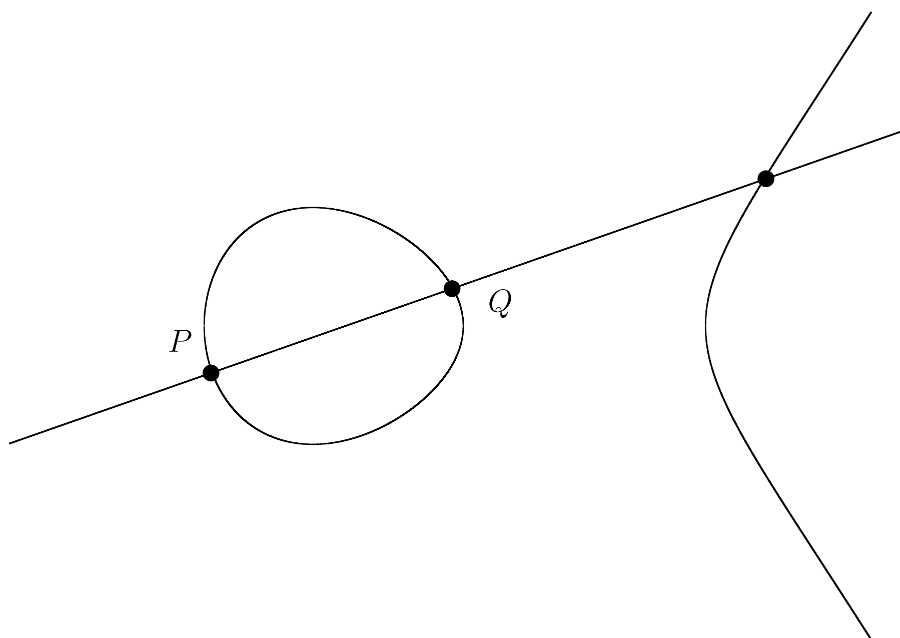

Fig. 1.2 – Droite passant par P et Q.

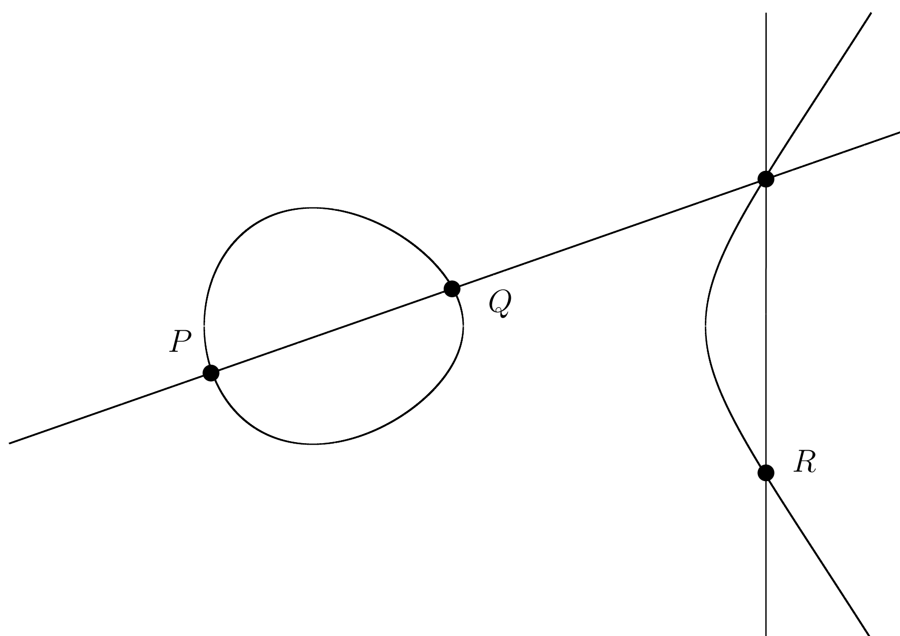

Fig. 1.3 – Calcul du point R de E tel que $P + Q = R$.

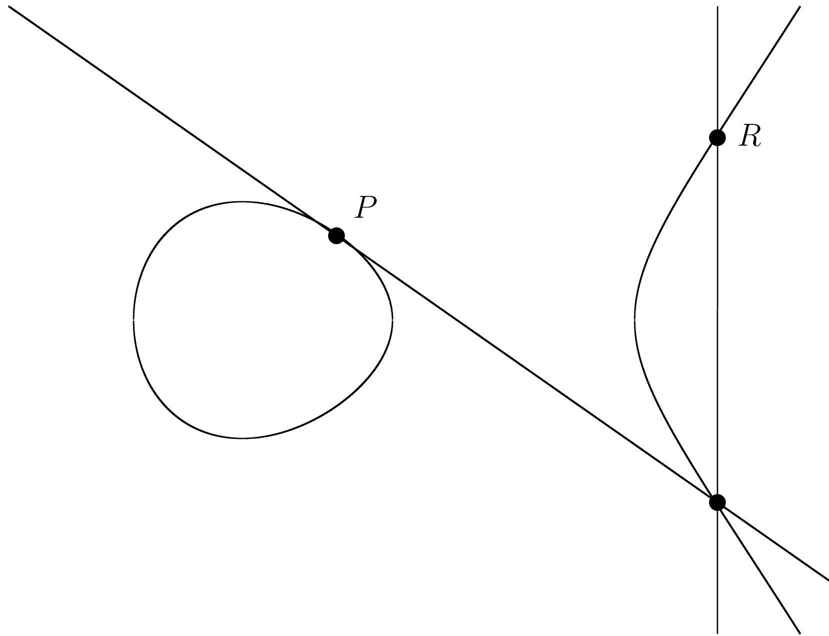

Fig. 1.4 – Calcul du doublement d'un point.

Pour l'addition de deux points P et Q, si $x_P - x_Q = 0$ alors $R = \infty$ (l'élément neutre du groupe) sinon nous avons pour λ :

$$\lambda = \frac{y_P - y_Q}{x_P - x_Q} \tag{1.3}$$

Pour le doublement de P, si $2y_P + a_2 x_P + a_3 = 0$ alors $R = \infty$, sinon nous avons pour λ :

$$\lambda = \frac{3x_P^2 + 2a_2 x_P + a_4 - a_1 y_P}{2y_P + a_2 x_P + a_3} \tag{1.4}$$

En utilisant le coefficient directeur de la droite qui passe par P et Q, nous calculons les coordonnées exactes de R :

$$x_R = -x_P - x_Q - a_2 + \lambda(\lambda + a_1) \tag{1.5}$$

$$y_R = -y_P - a_3 - a_1 x_R + \lambda(x_P - x_R) \tag{1.6}$$

Maintenant que nous possédons un groupe sur les courbes elliptiques, nous appliquons le protocole de Diffie Hellman (ou celui de ElGamal).

Pour Diffie Hellman, l'exponentiation sur $\mathbb{Z}_p^*$ devient une multiplication par un scalaire d'un point P générateur du groupe des points de la courbe elliptique. La multiplication kP d'un point P par un scalaire k se décompose en une suite d'additions et de doublements de points.

Algorithme 5 : Algorithme d'échanges de clé de Diffie Hellman

Input : public : une courbe elliptique E et un point P générateur du groupe de la courbe E

Output : privé : une clé K secrète et commune

begin

Alice choisit un entier a	Bob choisit un entier b
Alice évalue le point $A = aP$	Bob évalue le point $B = bP$
Alice envoie A à Bob	Bob envoie B à Alice
Alice évalue le point $K_a = aB$	Bob calcule $K_b = bA$

end

A la fin du protocole 5, Alice et Bob possèdent bien le même point $K_a = K_b$, c'est à dire le point de E évalué par abP.

L'avantage d'effectuer les opérations sur les courbes elliptiques plutôt que sur $\mathbb{Z}_p^*$ est qu'il n'existe à ce jour pas d'attaque générale du log discret sur le groupe des points d'une courbe elliptique. Certaines courbes ont été attaquées mais seul l'algorithme rho de Pollard (qui fonctionne sur tous les groupes et est optimal) fonctionne sur toutes les courbes. Il garantit donc le niveau de sécurité sur les courbes elliptiques.

Les protocoles cryptographiques basés sur les courbes elliptiques ont des besoins assez spécifiques. Pollard rho donne la sécurité et sa complexité est en $O(q^{1/2})$ où q est le nombre d'éléments du groupe G. Pour connaître la sécurité, il nous faut évaluer maintenant la taille de ce groupe. Or si le corps de base, celui sur lequel sont évaluées les coordonnées des points, est de taille p alors la taille q du groupe engendrée par la courbe elliptique E est d'après le théorème de Hasse [64] tel que

$$|q - (p + 1)| \leq 2\sqrt{p} \tag{1.7}$$

La différence entre q et p est négligeable pour les questions de sécurité. Par contre, il faut préciser que q doit être premier car Pollard rho calcule sa complexité sur le plus grand facteur de la taille du groupe.

Il faut donc choisir un corps suffisamment grand pour le niveau de sécurité désiré.

Nous nous intéressons particulièrement aux opérations sur le corps de base $\mathbb{Z}/p\mathbb{Z}$. Celles ci correspondent à des opérations modulaires. Le choix de p est laissé libre. p peut être choisi pour avoir une arithmétique efficace.

Par contre, nous devons effectuer tous les calculs nécessaires pour l'évaluation des coordonnées des points. Nous devons être capable de fournir une arithmétique modulaire complète, addition, multiplication, inversion, même si certains systèmes de coordonnées permettent de n'avoir qu'une seule inversion à faire sur la totalité du calcul aP.

1.4 Besoins arithmétiques

Dans le tableau 1.4, nous résumons les besoins en arithmétique des principaux protocoles cryptographiques à clés publiques.

Les besoins en arithmétiques sont assez variés. Nous nous plaçons par la suite plutôt dans le contexte des courbes elliptiques qui semble être l'un des cryptosystèmes les plus prometteurs : les besoins sont une arithmétique complète modulo un nombre premier p libre mais de taille supérieure à 160 bits. Dans ce contexte, nous définissons ce qui pour nous représente les *tailles cryptographiques*.

	RSA	Diffie Hellman	
		Classique	ECC
Arithmétique sur	L'anneau $\mathbb{Z}/n\mathbb{Z}$	Le groupe $\mathbb{Z}_p^*$	Le corps de base $\mathbb{Z}/p\mathbb{Z}$
Particularité	n composite fixé	p premier libre	p premier libre
Besoin en arithmétique	multiplication	multiplication	addition multiplication inversion
Sécurité	Taille du modulo		
2^{80}	1024		160
2^{112}	2048		224
2^{128}	3072		256
2^{192}	7680		384
2^{256}	15360		512

Tab. 1.4 – Les protocoles cryptographiques à clé publique.

Les tailles cryptographiques sont des tailles n de nombres premiers p sur lesquels nous désirons faire de la cryptographie et essentiellement les protocoles d'ECC. Les ordinateurs utilisant majoritairement des chiffres de taille 32 bits, les tailles cryptographiques sont des multiples de 32. Enfin, suivant les besoins sécuritaires nous avons comme tailles cryptographiques : 160,192,224,256,288,320,352,384,416,448,480 et 512. Cette liste n'est évidemment pas exhaustive.

L'objectif principal de ce travail de thèse est de pouvoir proposer pour chaque taille cryptographique, un nombre p premier pour lequel il existe une arithmétique modulaire efficace.

Chapitre 2

État de l'art sur l'arithmétique modulaire

Sommaire

2.1 L'addition modulaire 19
2.1.1 L'algorithme basé sur la retenue sortante d'Omura 19
2.1.2 L'algorithme de Takagi en représentation redondante 19
2.2 L'inversion modulaire 21
2.2.1 Via l'algorithme d'Euclide étendu 21
2.2.2 Via le théorème de Fermat 23
2.3 La multiplication modulaire 26
2.3.1 L'algorithme de Taylor avec mémorisation 27
2.3.2 L'algorithme de Blakley 28
2.3.3 L'algorithme de Montgomery 28
2.3.4 L'approximation du quotient par Barrett 31
2.3.5 L'utilisation d'une écriture redondante par Takagi 33
2.3.6 Étude comparative 33
2.4 La famille des nombres de Mersenne 34
2.4.1 Les nombres de Mersenne 35
2.4.2 La classe dense des nombres Pseudo Mersenne 36
2.4.3 Une vision polynomiale : les nombres de Mersenne Généralisés 41
2.4.4 Une extension des nombres de Mersenne Généralisés 44
2.4.5 Étude comparative 44

Dans ce chapitre, nous présentons un état de l'art sur l'arithmétique modulaire. L'efficacité des opérateurs modulaires est primordiale pour la cryptographie à clé publique. Nous analysons les trois opérations nécessaires au protocole ECC : l'addition, l'inversion et la multiplication modulaire. Nous abordons aussi les solutions conseillées par de nombreux standards pour l'implantion de ECC [50, 67]. L'algorithmique généraliste est remplacée par une algorithmique adaptée à un seul modulo. Nous analysons les différents choix de modulo possibles pour l'implantation des protocoles d'ECC.

Les protocoles d'ECC sont des protocoles cryptographiques utilisant des additions, des inversions et des multiplications sur un corps fini. Dans le cadre de corps premiers $\mathbb{Z}/p\mathbb{Z}$, ces opérations deviennent des opérations modulaires modulo le nombre premier p : additions modulaires, multiplications modulaires et inversions modulaires. L'addition modulaire est bien moins coûteuse que la multiplication modulaire. L'inversion peut se calculer par une exponentiation modulaire.

De nombreux autres protocoles, de Diffie-Hellman [23] à RSA [58], en passant par ElGamal [25], effectuent une exponentiation soit sur un corps premier $\mathbb{Z}/p\mathbb{Z}$ soit sur un anneau $\mathbb{Z}/n\mathbb{Z}$. L'exponentiation modulaire peut se décomposer en une suite de multiplications modulaires et de mises au carré modulaires.

La multiplication modulaire est une des opérations les plus utilisées de la cryptographie à clé publique. Toute amélioration de sa complexité a des répercussions positives sur la complexité de l'exponentiation ou de l'inversion modulaire et par conséquent sur les protocoles cryptographiques présentés dans le chapitre 1. La multiplication modulaire étant l'opération centrale de l'arithmétique pour la cryptographie à clé publique, elle a été fortement étudiée [45] et sa complexité largement améliorée.

Dans ce chapitre, nous présentons les quelques algorithmes d'additions modulaires existants. Il existe plusieurs propositions intéressantes malgré la faible marge de manoeuvre dûe à la grande simplicité de cette opération et à sa faible complexité.

A l'opposé, nous présentons une opération fortement complexe : l'inversion modulaire. Son calcul n'a été que difficilement amélioré. Les algorithmes modernes s'inspirent encore fortement d'anciennes méthodes comme celles d'Euclide ou de Fermat. La multiplication modulaire joue un rôle important dans la complexité de l'inversion modulaire à travers entre autre l'exponentiation modulaire et les différentes techniques utilisées pour le calcul de celle-ci.

Nous abordons ensuite les algorithmes de multiplication modulaire. Deux approches sont possibles sur la façon de calculer une multiplication modulaire. Une première catégorie intègre la réduction à la multiplication elle même. Cette technique a pour inconvénient d'imposer un algorithme pour effectuer la multiplication. C'est pour cela que la majorité des algorithmes de multiplications modulaires appartiennent à la seconde catégorie ; la multiplication modulaire y est décomposée en une multiplication suivie d'une réduction modulaire. La réduction modulaire est équivalente à une division euclidienne d'un entier c par une constante p connue (appelé *modulo*) dans laquelle on ne désire récupérer que le reste r :

$$c = r + qp \text{ avec } 0 \leq r < p \tag{2.1}$$

Puisque la multiplication modulaire est une multiplication suivie d'une réduction modulaire, le reste r est tel que

$$r + qp = ab \text{ avec } 0 \leq a, b, r < p \tag{2.2}$$

Le calcul du quotient q n'est pas obligatoire pour la réduction d'une multiplication modulaire. Néanmoins certains algorithmes utilisent le calcul du quotient pour effectuer leur réduction. Ceci permet à de nombreux algorithmes de précalculer avantageusement différentes valeurs pour accélérer le calcul de la réduction.

La multiplication modulaire est traitée dans les deux dernières parties de ce chapitre. Dans la première, nous présentons les algorithmes "généralistes". Un algorithme généraliste de multiplication modulaire est un algorithme qui fonctionne pour tous les moduli. Sa complexité ne tient pas compte du fait que le modulo appartienne ou pas à une classe de nombres spécifiques.

Dans la dernière partie de ce chapitre, nous abordons les classes de moduli particulières et les algorithmes de réduction modulaire qu'elles utilisent. Ces algorithmes ont pour inconvénient de fonctionner sur une classe restreinte de moduli (les nombres de Mersenne par exemple). En utilisant les propriétés spécifiques de ces classes de nombres, ces algorithmes ont l'avantage de faire des réductions modulaires rapides. C'est pour cela que leur utilisation est conseillée par les standards cryptographiques [50, 67].

Dans ce chapitre, nous posons l'entier n comme étant la taille de p en base β : $\beta^{n-1} \leq p < \beta^n$. Si la valeur de β n'est pas précisée, elle est considérée comme égale à 2 ($2^{n-1} \leq p < 2^n$).

2.1 L'addition modulaire

Comme la multiplication modulaire, l'addition modulaire peut être décomposée en une addition suivie d'une réduction. Nous ne présentons pas ici les différents classes de moduli qui peuvent améliorer la complexité de l'addition modulaire. Celles ci sont les mêmes que pour la multiplication modulaire et sont vues dans la section 2.4. La réduction dans ce cas précis correspond à une seule soustraction. La complexité de l'addition modulaire est relativement faible.

Cette complexité peut tout de même être améliorée. J. Omura [54] modifie les deux opérations à effectuer, addition et soustraction. Ainsi, il évite d'avoir à faire une comparaison et fait uniquement des additions.

La méthode proposée par N. Takagi [69] utilise un système redondant de représentation des nombres. C'est à ce jour une des méthodes les plus efficaces.

2.1.1 L'algorithme basé sur la retenue sortante d'Omura

En 1990, J. Omura optimise l'addition modulaire [54]. Il propose l'algorithme 6 qui permet d'accélérer les calculs en transformant une comparaison par la simple analyse de la retenue sortante.

Algorithme 6 : Addition modulaire d'Omura

Input : a, b, p avec $0 \leq a, b < p$ et $c = 2^n - p$
Output : s avec $s = a + b \bmod p$
begin
  $s \leftarrow a + b$
  $t \leftarrow s + c$
  if $t \geq 2^n$ **then** $s \leftarrow t \bmod 2^n$
end

Nous vérifions l'exactitude de l'algorithme 6.

- Si $a + b < p$ alors $a + b + c < p + c = 2^n$ et l'algorithme retourne $s = a + b$ qui est bien inférieur à p.
- Si $a + b \geq p$ alors $a + b + c \geq p + c = 2^n$ et l'algorithme retourne $a + b + c \bmod 2^n = a + b + c - 2^n = a + b - p$ qui est bien inférieur à p puisque $a + b < 2p$ et supérieur à 0 puisque $a + b \geq p$.

Le test de la troisième ligne ne coûte rien : il correspond à l'observation de la retenue sortante du calcul de t. La réduction modulo la puissance de 2 n'est pas comptée non plus dans le coût de l'algorithme 6.

Au final, cet algorithme effectue deux additions de n bits.

$$C_{algo\ 6} = 2Add_n$$

2.1.2 L'algorithme de Takagi en représentation redondante

En 1992, N. Takagi adapte les systèmes d'additions redondantes au modulaire [69]. Il utilise une représentation redondante des nombres : le *borrow save*.

Définition 2 *Le borrow save est un système de représentation où un entier a est représenté par deux entiers a^+ et a^- tels que $a = a^+ - a^-$.*

Il existe des algorithmes (par exemple l'algorithme 7) pour additionner en temps constant des nombres représentés en borrow save.

Algorithme 7 : `ATC` : Addition en Temps Constant

Input : a, b avec $a = a^+ - a^-, b = b^+ - b^-$
Output : s avec $s = a + b$ et $s = s^+ - s^-$
begin
  $c_0^+ \leftarrow 0$
  for $i \leftarrow 1$ ***to*** n **do**
    Soient c_{i+1}^+, c_i^- tels que $2c_{i+1}^+ - c_i^- = a_i^+ + b_i^+ - a_i^-$
  end
  for $i \leftarrow 1$ ***to*** n **do**
    Soient s_{i+1}^-, s_i^+ tels que $2s_{i+1}^- - s_i^+ = b_i^- + c_i^- - c_i^+$
  end
  $s_0^- \leftarrow 0$
  $s_n^+ \leftarrow c_n^+$
end

Les n passages dans chacune des boucles peuvent se faire en parallèle. Les opérations dans la première boucle sont indépendantes les unes des autres, elles peuvent se faire en parallèle. Une fois cette boucle terminée, la seconde boucle se fait dans les mêmes conditions. La complexité de calcul est équivalente en temps à deux additionneurs de type “full adder”. Un Full Adder, noté `FA` reçoit en entrée trois valeurs a, b, c sur un bit et retourne une valeur s sur deux bits telle que $s = a+b+c$. Il est la brique fondamentale des additionneurs.

$$\forall a, b, c \in \{0, 1\}, \quad \texttt{FA}(a, b, c) = a + b + c = s \text{ avec } s = (s_1, s_0)_2$$

L'algorithme d'Omura ne peut pas directement être utilisé pour effectuer une addition modulaire en redondant. Le borrow save est un système de représentation où la comparaison est difficile. Par exemple, $(1, -1, 0)_2 = 1\times2^2 - 1\times2^1 + 0 = 4 - 2 = 2$ est plus petit que $(0, 1, 1)_2 = 0\times2^2 + 1\times2^1 + 1 = 2 + 1 = 3$.

Takagi propose une solution à ce problème en utilisant un bit supplémentaire. Ceci lui permet en regardant uniquement les trois bits de poids fort de savoir si un entier est inférieur à $-p$, supérieur à p ou entre les deux (Voir Algorithme 8).

Algorithme 8 : Addition modulaire de Takagi

Input : a, b avec $a_i, b_i \in \{-1, 0, 1\}$ et $-p < a, b < p$
Output : s avec $s = a + b \pmod p$ et $-p < a, b < p$
begin
  $s \leftarrow \texttt{ATC}(a, b)$
  if $s_{n+1}s_ns_{n-1} < 0$ **then** $s \leftarrow \texttt{ATC}(s, p)$
  else if $s_{n+1}s_ns_{n-1} > 0$ **then** $s \leftarrow \texttt{ATC}(s, -p)$
end

En regardant uniquement les trois bits de poids fort, nous pouvons choisir l'opération à effectuer pour conserver s dans un intervalle acceptable.

Trois cas sont possibles.

1. Si $s_{n+1}s_ns_{n-1} < 0$ alors s est négatif. Le borrow save a la particularité que si le nombre $a_n...a_i$ a un signe (> 0 ou < 0) alors le nombre $a_n...a_0$ a le même signe. Nous avons donc $-2p < s < 0$. Il faut additionner à s le modulo p pour revenir dans l'intervalle initial : $-p < s + p < p$.

2. Si $s_{n+1}s_ns_{n-1} > 0$ alors s est positif. Nous avons donc $0 < s < 2p$. Il faut additionner à s le nombre $-p$ pour revenir dans l'intervalle initial : $-p < s - p < p$.
3. Le dernier cas est $s_{n+1}s_ns_{n-1} = 0$. Dans ce cas nous obtenons que $-2^{n-1} < s < 2^{n-1}$. s est déjà dans l'intervalle désiré : $-p < s < p$.

L'avantage de cette technique est évidemment sa très grande rapidité. Par contre, si l'entrée dans la représentation ne coûte rien ($a^+ = a$ et $a^- = 0$), la sortie correspond à une soustraction classique ($a = a^+ - a^-$). Cette représentation est intéressante lors de nombreuses utilisations de l'addition modulaire redondante. Dans la section 2.3, nous détaillons comment Takagi utilise cette représentation pour effectuer une multiplication modulaire (et par extension une exponentiation modulaire). Il décompose la multiplication modulaire en une suite d'additions modulaires. L'objectif est d'effectuer un maximum d'opérations sans changer de représentation.

2.2 L'inversion modulaire

L'inversion modulaire d'un entier x modulo un nombre premier p est noté $x^{-1} \bmod p$. Elle correspond à trouver l'entier y tel que $xy = 1 \bmod p$. Deux méthodes sont possibles. La première est une extension de l'algorithme proposé par Euclide pour calculer le `PGCD`, *Plus Grand Commun Diviseur*, de deux nombres. La seconde est basée sur le petit théorème de Fermat qui transforme l'inversion en une exponentiation.

2.2.1 Via l'algorithme d'Euclide étendu

L'algorithme 9 dû à Euclide (4$^{\text{ème}}$ et 3$^{\text{ème}}$ siècles avant JC) calcule le `PGCD` de deux entiers a et b (Voir [19]).

Algorithme 9 : `PGCD`

```
Input  : a, b
Output : d, le plus grand diviseur commun à a et b
begin
    u ← a
    v ← b
    while v > 0 do
        q ← ⌊u/v⌋
        t ← v
        v ← u − qv
        u ← t
    end
    d ← u
end
```

Nous vérifions que l'algorithme 9 retourne bien le `PGCD` de a et de b. Le `PGCD` vérifie deux propriétés.

a) Premièrement, le `PGCD` est multiple de tous les diviseurs communs à a et b. Or si un entier divise a et b alors il divise u et v à chaque tour de la boucle : un diviseur de u et v divise aussi $u - qv$. Nous obtenons que d est bien divisible par tous les diviseurs communs à a et b.

b) Deuxièmement, le `PGCD` divise a et b. Nous vérifions que d divise bien a et b en inversant le raisonnement de la première étape. A la dernière étape, nous avons que d divise u (puisque $u = d$) et d divise aussi v (puisque celui ci est égal à 0). Or si d divise $u - qv$ et v alors d divise u. Ainsi nous pouvons remonter jusqu'au fait que d divise a et b.

S'inspirant de ce premier algorithme, Aryabhata, un indien du $5^{\text{ème}}$ siècle, crée l'algorithme d'Euclide étendu (Algorithme 10). Cet algorithme reçoit deux entiers a et b et calcule des coefficients u et v tels qu'ils satisfassent *l'identité de Bezout* (Voir Équation 2.3)

$$au + bv = d \text{ avec } d = \texttt{PGCD}(a, b) \tag{2.3}$$

Algorithme 10 : Euclide étendu

```
Input  : a, b
Output : u, v, d tel que au + bv = d où d = PGCD(a, b)
begin
    (u1, u2, u3) ← (1, 0, a)
    (v1, v2, v3) ← (0, 1, b)
    while v3 ≠ 0 do
        q ← ⌊u3/v3⌋
        (t1, t2, t3) ← (v1, v2, v3)
        (v1, v2, v3) ← (u1, u2, u3) − q(v1, v2, v3)
        (u1, u2, u3) ← (t1, t2, t3)
    end
    (u, v, d) ← (u1, u2, u3)
end
```

Nous vérifions que l'algorithme 10 retourne le bon résultat. Nous avons deux invariants de boucle : $u_1a + u_2b = u_3$ et $v_1a + v_2b = v_3$. Les valeurs de u_3 et de v_3 suivent exactement les mêmes calculs que l'algorithme d'Euclide initial qui calcule le PGCD. Nous obtenons que $u_3 = d$. Grâce à l'invariant de boucle, nous vérifions : $u_1a + u_2b = d$ où $d = \texttt{PGCD}(a, b)$.

L'algorithme 10 est utilisé pour calculer l'inverse dans un corps premier. Dans le cas de l'inversion modulaire, nous posons $a = p$ et $b = x$ pour obtenir $v = x^{-1} \pmod p$. Nous avons alors $up + vx = 1$ car $\texttt{PGCD}(x, p) = 1$. Cette équation donne $vx \equiv 1 \pmod p$.

Exemple 3 *Utilisons l'algorithme 10 pour inverser* $x = 23$ *modulo* $p = 41$. *Observons le tableau 2.1.*

(u_1, u_2, u_3)	$(1, 0, 41)$	$(0, 1, 23)$	$(1, -1, 18)$	$(-1, 2, 5)$	$(4, -7, 3)$	$(-5, 9, 2)$	$(9, -16, 1)$
(v_1, v_2, v_3)	$(0, 1, 23)$	$(1, -1, 18)$	$(-1, 2, 5)$	$(4, -7, 3)$	$(-5, 9, 2)$	$(9, -16, 1)$	$(-23, 41, 0)$
q	1	1	1	3	1	1	2

TAB. 2.1 – Exemple d'utilisation de l'algorithme d'Euclide étendu.

Nous obtenons $x^{-1} \equiv -16 \equiv 25 \pmod p$. *Nous vérifions que* $23 \times 25 \equiv 1 \pmod{43}$.

Avec l'équation 2.4, Knuth [35] donne le nombre moyen de tours dans la boucle que l'algorithme 10 effectue.

$$0.843n + 1.47 \tag{2.4}$$

Chaque tour dans la boucle comporte une division d'éléments de taille n bits et 3 multiplications. Ces trois multiplications peuvent se transformer en une seule puisque la division nous donne déjà v_3 et que l'on n'a pas besoin de v dans le cas particulier de l'inversion modulaire.

$$C_{algo\ 10} \sim (0.843n + 1.47)(Div_n + Mul_n)$$

Le cas le pire correspond à prendre pour u et v deux nombres de Fibonacci consécutifs. Ceux ci sont créés à partir de la suite de Fibonacci régie par $F_0, F_1 = 1$ et $F_i = F_{i-1} + F_{i-2}$. Les premiers nombres de Fibonacci sont $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$. Au final, la complexité de l'algorithme d'Euclide étendu est donnée par Lamé en 1844 (Voir [35]).

Théorème 1 (Lamé) *Si $0 \leq a, b < N$, le nombre de divisions dans l'algorithme d'Euclide appliqué à a et b est au plus de $\lceil \log_\phi \sqrt{5}N \rceil - 2$, où ϕ est le nombre d'or $\phi = \frac{1+\sqrt{5}}{2}$.*

En pratique, la complexité du pire cas est approximativement $1.44n - 0.328$.

$$C_{algo\ 10} < (1.44n - 0.328)(Div_n + Mul_n)$$

La différence de complexité entre le cas moyen et le pire cas fait que l'algorithme d'Euclide étendu est intéressant pour l'implantation logicielle mais beaucoup moins pour l'implantation matérielle. Les circuits sont construits en fonction du cas pire et ne tire aucun avantage d'un "bon" cas moyen.

2.2.2 Via le théorème de Fermat

Au 17$^{\text{ème}}$ siècle, Pierre Simon de Fermat propose ce "petit" théorème (Voir [19]).

Théorème 2 (Fermat) *Soit $0 < x < p$ avec p premier alors $x^{p-1} \equiv 1 \pmod p$*

Ce théorème de Fermat permet de calculer l'inverse de x modulo p. En multipliant l'équation $x^{p-1} \equiv 1 \pmod p$ par $x^{-1} \pmod p$, nous obtenons un calcul de l'inverse.

$$x^{-1} \equiv x^{p-2} \pmod p \tag{2.5}$$

Le calcul de l'inverse sur un corps premier correspond à une exponentiation modulaire.

Pour ce qui est de l'inversion sur un anneau (par exemple pour RSA), au 18$^{\text{ème}}$ siècle, Leonhard Euler généralise le petit théorème de Fermat (Voir Théorème 3). Il utilise *l'indicateur d'Euler*.

Définition 3 *L'indicateur d'Euler, noté $\phi(n)$, est le nombre d'entiers premiers avec n compris entre 1 et $n - 1$.*

L'indicateur d'Euler $\phi(n)$ correspond entre autres au nombre d'entiers inversibles sur l'anneau $\mathbb{Z}/n\mathbb{Z}$.

Théorème 3 (Euler) *Soit $0 < x < n$, deux entiers premiers entre eux, alors $x^{\phi(n)} \equiv 1 \pmod n$ où $\phi(n)$ est l'indicateur d'Euler.*

De même que pour Fermat, cette formule permet de calculer l'inverse de x (si il existe) modulo n.

$$\text{Pour tout } x \text{ tel que } 0 < x < n, \quad \mathtt{PGCD}(x, n) = 1 \Rightarrow x^{-1} \equiv x^{\phi(n)-1} \pmod n \tag{2.6}$$

La formule de Fermat se retrouve à partir de la formule d'Euler. Si p est premier alors $\phi(p) = p - 1$.

Le calcul de l'inversion sur un corps ou un anneau correspond à une exponentiation modulaire. Il existe de nombreuses méthodes pour effectuer une exponentiation modulaire. Nous proposons ici

la méthode considérée comme standard. Elle est basée sur un principe du “square and multiply”, suite de mise au carré et de multiplication :

Algorithme 11 : Exponentiation Modulaire Binaire

```
Input  : 0 < x, e < p avec e = (e_{n-1}, .., e_0)_2
Output : y ≡ x^e (mod p)
begin
    y ← x^{e_{n-1}}
    for i ← n − 2 to 0 do
        y ← y^2 mod p
        if e_i = 1 then y ← xy mod p
    end
end
```

Exemple 4 *En utilisant le petit théorème de Fermat et l'algorithme 11, nous inversons* $x = 23$ *modulo* $p = 41$ *(Voir Tableau 2.2). Notons que* $e = p - 2 = 39 = (1, 0, 0, 1, 1, 1)_2$

i	–	4	3	2	1	0
e_i	1	0	0	1	1	1
y	23	23	37	16	25	25
y^2	–	$23.23 \equiv 37$	$37.37 \equiv 16$	$16.16 \equiv 10$	$25.25 \equiv 10$	$25.25 \equiv 10$
xy	–	–	–	$23.10 \equiv 25$	$23.10 \equiv 25$	$23.10 \equiv 25$

Tab. 2.2 – Exemple d'inversion via Fermat.

Nous obtenons $x^{-1} \equiv x^{39} \equiv 25 \pmod{41}$ *comme pour l'algorithme d'Euclide étendu.*

Si n est la taille binaire de e, nous avons comme complexité :

a) $n - 1$ mises au carré modulaires

b) $n - 1$ multiplications modulaires au pire et $(n - 1)/2$ en moyenne

$$C_{algo\ 11} \sim \frac{3}{2}(n-1)MulMod_n < 2(n-1)MulMod_n$$

Cette complexité peut être diminuée en utilisant différentes techniques [29] (comme celle de la *fenêtre glissante* par exemple). L'exponentiation binaire utilise l'écriture binaire, en base 2, de e. Ce que nous proposons pour améliorer la complexité est d'écrire e en base 2^k. Nous notons $(e_{l-1}, \ldots, e_0)_{2^k}$, l'écriture de e en base 2^k telle que $e = \sum_{i=0}^{l-1} e_i (2^k)^i$.

Il faut commencer par mémoriser toutes les puissances de x ayant pour exposant les entiers entre 1 et $2^k - 1$. La mise au carré modulaire initiale se transforme en une suite de k mises au carré modulaires comme nous pouvons le voir dans l'algorithme 12.

Algorithme 12 : Exponentiation Modulaire en base 2^k

```
Input   : 0 < x, e < p avec e = (e_{l-1}, .., e_0)_{2^k}
Output  : u avec y ≡ x^e (mod p)
begin
    MEM(0) ← 1
    for i ← 1 to 2^k − 1 do
        MEM(i) ← MEM(i − 1) × x mod p
    end
    b ← MEM(e_{l−1})
    for i ← l − 2 to 0 do
        for j ← 1 to k do
            y ← y^2 mod p
        end
        y ← MEM(e_i) × y mod p
    end
end
```

La valeur de k intervient dans la complexité de l'algorithme 12. La longueur l correspond au nombre de chiffres utilisés pour écrire e en base 2^k.

a) $2^k - 3$ multiplications modulaires pour la partie mémorisation

b) $\left\lceil \frac{n}{k} - 1 \right\rceil k \leq n$ mises au carré modulaires

c) $\left\lceil \frac{n}{k} - 1 \right\rceil$ multiplications modulaires

$$C_{algo\ 12} = \left(\left\lceil \frac{n}{k} - 1 \right\rceil (k+1) + 2^k - 3 \right) MulMod_n$$

Les différentes tailles cryptographiques vont de $n = 160$ à $n = 512$. Dans ces conditions, nous choisissons $k = 4$ ou $k = 5$ pour que $\left(\left\lceil \frac{n}{k} - 1 \right\rceil (k+1) + 2^k - 3 \right)$ soit minimale. Par exemple, si $n = 160$ et $k = 4$, l'exponentiation coûte au pire

a) 156 mises au carré modulaires au lieu de 159 pour l'exponentiation binaire.

b) $14 + 39 = 53$ multiplications modulaires alors que le cas moyen de l'exponentiation binaire (Algorithme 11) est déjà de 80.

La multiplication modulaire est d'une importance capitale dans la complexité de l'exponentiation. Ceci est important pour Diffie Hellman, ElGamal ou RSA qui effectuent des exponentiations, mais aussi pour l'inversion de ECC.

Le tableau 2.3 récapitule les complexités dans le cas moyen ($\sim$) et le pire cas ($<$) des différentes méthodes d'inversion. Il faut faire plusieurs remarques pour bien appréhender ce tableau.

- C'est le nombre de tours de boucle qui est approché. Un tour de boucle correspond à une multiplication et une division pour Euclide et à une multiplication modulaire pour Fermat. La multiplication modulaire étant au pire une multiplication suivie d'une division, les tours des algorithmes utilisant le théorème de Fermat doivent être considérés comme moins coûteux que les tours d'Euclide.
- Pour l'algorithme 12, seul $k = 4$ et $k = 5$ sont évalués car ils sont les plus intéressants pour les tailles étudiées.
- Si l'on fait une exponentiation en base 2^k, il n'y a pas vraiment de cas moyen : le cas moyen correspond au pire cas.

	Via Euclide		Via Fermat			
	algo 10		algo 11		algo 12	
	$\sim$	$<$	$\sim$	$<$	$k = 4$	$k = 5$
n	0.843n+1.47	1.44n-0.328	1.5n-1.5	2n-2	1.25n+8	1.2n+23
160	137	231	239	318	208	215
192	164	277	287	382	248	254
224	191	323	335	446	288	292
256	218	369	383	510	328	331
288	245	415	431	574	368	369
320	272	461	479	638	408	407
352	299	507	527	702	448	446
384	326	553	575	766	488	484
416	353	599	623	830	528	523
448	380	645	671	894	568	561
480	407	691	719	958	608	599
512	434	737	767	1022	648	638

Tab. 2.3 – Coût de l'inversion modulaire.

2.3 La multiplication modulaire

Les algorithmes de multiplications modulaires généralistes sont des algorithmes qui fonctionnent pour n'importe quel modulo, c'est-à-dire sans utiliser ni requérir aucune propriété sur le modulo. Les algorithmes que nous donnons dans cette partie sont légèrement modifiés par rapport à leur écriture initiale mais correspondent bien à l'idée originale. De plus, cette partie correspond à une liste non exhaustive d'algorithmes de multiplication modulaire généralistes.

Nous commençons par la décomposition de la multiplication sous forme de carrés par F. J. Taylor [70]. Ceci lui permet d'utiliser des tables mémoires pour calculer la mise au carré et d'ainsi effectuer une multiplication modulaire sans utiliser de multiplieur.

L'algorithme de G. R. Blakley [15] est le premier algorithme de multiplication modulaire intégrée : la réduction est intégrée dans les calculs de la multiplication. C'est en fait une adaptation en modulaire du "double and add".

C'est en 1985 que Peter Montgomery [48] crée un algorithme de multiplication modulaire qui n'est plus une adaptation de la multiplication classique au modulaire. Pour effectuer la réduction modulaire, il n'utilise pas de division euclidienne par le modulo. Il remplace cette opération par une division par une puissance de la base 2^n. La réduction modulaire a alors un coût vraiment intéressant. Par contre, Montgomery utilise une représentation dédiée à son algorithme.

Paul Barrett [14] propose lui d'approcher le calcul du quotient. Cette approximation est suffisamment précise pour avoir un reste à une ou deux soustractions du reste exact. Elle accélère beaucoup les calculs : le coût de son algorithme est très proche de celui de l'algorithme de Montgomery. Il conserve une représentation classique.

Il faut enfin citer l'algorithme de N. Takagi [69] qui utilise son propre système d'addition modulaire en représentation redondante pour proposer une multiplication intégrée très efficace. Celle-ci est une fois de plus sous la forme d'un "double and add".

Nous terminons cette partie par une comparaison de ces différents algorithmes. Cette comparaison montre l'utilité de quasiment tous ces algorithmes mais dans des contextes différents.

2.3.1 L'algorithme de Taylor avec mémorisation

En 1981 [70], F. J. Taylor propose un algorithme de multiplication modulaire utilisant une représentation classique des nombres. Cet algorithme utilise une décomposition de la multiplication en carrés :

$$ab = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4} \tag{2.7}$$

Cette décomposition est adaptable au modulaire à condition que le modulo p soit impair pour pouvoir faire la division par 4.

Cette décomposition est intéressante uniquement si deux mises au carré modulaire ($a^2 \bmod p$) sont moins coûteuses qu'une multiplication modulaire. Taylor utilise une table mémoire, `MEM`, qui effectue cette opération coûteuse. Cette table reçoit en entrée un entier x avec $0 \leq x < p$ et retourne en sortie un entier y avec $0 \leq y < p$ tel que $y \equiv x^2 \pmod{p}$. Pour accélérer encore un peu plus les calculs, nous intègrons aussi la division par 4 modulo p. Ce qui donne pour `MEM` :

$$\text{Pour tout } x \text{ tel que } 0 \leq x < p, \quad \texttt{MEM}(x) \leftarrow 4^{-1}x^2 \pmod{p}$$

Le signe n'a pas d'importance pour une mise au carré ; nous prenons la valeur absolue. Ceci a pour avantage de limiter la taille de $a+b$. En soustrayant p au calcul de $a+b$, nous transformons $0 \leq a+b < 2p$ en $-p < a+b-p < p$. Après avoir enlevé le signe, nous obtenons $0 \leq |a+b-p| < p$.

Algorithme 13 : Multiplication Modulaire de Taylor

Input : a, b, p avec $0 \leq a, b < p$
Data : `MEM` avec $\texttt{MEM}(x) = 4^{-1}x^2 \bmod p$
Output : s avec $s = ab \bmod p$
begin
  $u \leftarrow |a+b-p|$
  $v \leftarrow |a-b|$
  $s \leftarrow \texttt{MEM}(u) - \texttt{MEM}(v)$
  if $s < 0$ **then** $s \leftarrow s + p$
end

Exemple 5 *Calculons $a = 21$ multiplié par $b = 34$ modulo $p = 43$ en utilisant l'algorithme 13.*

Nous calculons $u = |21+34-43| = 12$ et $v = |21-34| = 13$. Nous appelons ensuite MEM(12) qui retourne $\frac{1}{4}12^2 \bmod 43 = 36$. Puis nous appelons MEM(13) qui retourne $\frac{1}{4}13^2 \bmod 43 = 10$. Enfin, nous soustrayons $36 - 10 = 26$.

Nous pouvons vérifier que $21 \times 34 \bmod 43 = 26$.

L'algorithme 13 fonctionne pour tous les moduli impairs. Sa complexité en temps est fortement minimisée. Cette multiplication modulaire ne nécessite que cinq additions et deux appels mémoire.

$$C_{algo\ 13} = 5Add_n + 2\texttt{MEM}$$

Le contrecoup de cet méthode est la taille de la table mémoire. La table `MEM` reçoit des entiers compris entre 0 et $p-1$ et retourne des entiers de la taille de p qui est de n bits. Nous déduisons la taille #`MEM` de la table mémoire `MEM`.

$$\#\texttt{MEM} = pn \text{ bits}$$

2.3.2 L'algorithme de Blakley

En 1983 [15], G. R. Blakley adapte un "double and add" classique en y intégrant des réductions après chaque étape de calcul. Cet algorithme fonctionne pour tous les moduli et utilise une représentation classique des nombres.

Algorithme 14 : Multiplication Modulaire de Blakley

```
Input  : a, b, p avec 0 ≤ a, b < p
Output : s avec s = ab mod p
begin
    a = [a_{n-1}, ..., a_0]_2
    s ← 0
    for i ← n − 1 to 0 do
        s ← 2s
        if s ≥ p then s ← s − p
        s ← s + a_i b mod p
        if s ≥ p then s ← s − p
    end
end
```

L'algorithme 14 a une complexité en temps de $3n$ additions de n bits : le doublement en base 2 n'est pas compté.

$$C_{algo\ 14} = 3nAdd_n$$

Il n'est pas très performant mais sert de plan pour d'autres algorithmes bien plus optimisés (celui de Takagi par exemple) : l'idée d'intégrer la réduction à l'intérieur de la multiplication est utilisée par d'autre algorithmes.

Exemple 6 *Nous reprenons notre exemple :* $21 \times 34 \bmod 43$.
Nous décomposons a *en binaire sur* $n = 6$ *bits car* 43 *est sur six bits.*

$$a = (0, 1, 0, 1, 0, 1)_2$$

Dans le tableau 2.4, nous observons les six étapes de l'algorithme 14.

Exemple d'utilisation de l'algorithme de Blakley				
i	$s \leftarrow 2s$	$s \geq p \Rightarrow s \leftarrow s - p$	$s \leftarrow s + a_i b$	$s \geq p \Rightarrow s \leftarrow s - p$
5	$s \leftarrow 2 \times 0$		$s \leftarrow 0 + 0 \times 34 = 0$	
4	$s \leftarrow 2 \times 0$		$s \leftarrow 0 + 1 \times 34 = 34$	
3	$s \leftarrow 2 \times 34 = 68$	$s \leftarrow 68 - 43 = 25$	$s \leftarrow 25 + 0 \times 34 = 25$	
2	$s \leftarrow 2 \times 25 = 50$	$s \leftarrow 50 - 43 = 7$	$s \leftarrow 7 + 1 \times 34 = 41$	
1	$s \leftarrow 2 \times 41 = 82$	$s \leftarrow 82 - 43 = 39$	$s \leftarrow 39 + 0 \times 34 = 39$	
0	$s \leftarrow 2 \times 39 = 78$	$s \leftarrow 78 - 43 = 35$	$s \leftarrow 35 + 1 \times 34 = 69$	$s \leftarrow 69 - 43 = 26$

TAB. 2.4 – Exemple de multiplication modulaire avec l'algorithme de Blakley.

2.3.3 L'algorithme de Montgomery

En 1985, Peter Montgomery franchit un cap dans la réduction modulaire [48]. Il transforme la réduction modulo p en une division par une puissance de la base $r = 2^n$. En pratique, une division

par une puissance de la base revient à un décalage, qui ne coûte rien.

Pour réduire un entier c modulo p (avec c sur $2n$ bits et p sur n bits), la méthode classique calcule un multiple qp de p tel que la partie haute de qp soit égale à la partie haute de c. Ainsi, en soustrayant qp à c, nous obtenons une valeur équivalente à c modulo p mais dont tous les bits de poids fort sont nuls : $s = c - qp < 2^n$ avec $s = c \bmod p$.

$$s = (0, \ldots, 0, s_{n-1}, \ldots, s_0)_2$$

Pour réduire un entier c modulo p, la méthode de Montgomery est de calculer un multiple qp de p tel que la partie basse de qp soit égale à la partie basse de c. Ainsi, en soustrayant qp à c, nous obtenons une valeur équivalente à c modulo p mais dont tous les bits de poids faible sont nuls :

$$u = (u_{2n-1}, \ldots, u_n, 0, \ldots, 0)_2$$

Le résultat obtenu est ainsi divisible par 2^n. En effectuant cette division, nous obtenons une valeur inférieure à 2^n : $s = \frac{c-qp}{2^n} < 2^n$. Par contre, s n'est pas équivalent à c modulo p. Si l'on pose r' tel que $rr' = 1 \bmod p$ où $r = 2^n$ alors $s = cr' \bmod p$. Pour que la multiplication modulaire de Montgomery soit stable, il faut utiliser une représentation particulière.

Définition 4 *La représentation de Montgomery, notée* mon*, représente l'entier* x *modulo* p *par* $\mathsf{mon}(x) = xr \bmod p$ *où* $r = 2^n$*.*

Algorithme 15 : Multiplication Modulaire de Montgomery

Input : a, b, p avec $0 \leq a, b < p$
Data : p' avec $(-p)p' = 1 \bmod r$
Output : s tel que $s = abr' \bmod p$ avec $rr' = 1 \bmod p$
begin
  $c \leftarrow ab$
  $q \leftarrow cp' \bmod r$
  $s \leftarrow (c + qp)/r$
  if $s \geq p$ **then** $s \leftarrow s - p$
end

L'algorithme 15 de réduction fonctionne pour tous les moduli. Il utilise un précalcul mais surtout une représentation particulière des nombres, mon. L'algorithme 15 retourne $abr' \bmod p$ et non $ab \bmod p$. C'est pour cela que l'on utilise la représentation de Montgomery, mon.

La représentation de Montgomery, mon, est stable par la multiplication modulaire si l'on utilise l'algorithme 15 décrivant la multiplication de Montgomery.

$$\begin{aligned} \texttt{montgomery}(\mathsf{mon}(x), \mathsf{mon}(y)) &= \mathsf{mon}(x)\mathsf{mon}(y)r' \bmod p && (2.8)\\ &= xryrr' \bmod p && (2.9)\\ &= xyr \bmod p && (2.10)\\ &= \mathsf{mon}(xy) && (2.11) \end{aligned}$$

De même, la représentation mon est naturellement stable par l'addition classique.

$$\begin{aligned} \mathsf{mon}(x) + \mathsf{mon}(y) &= xr + yr && (2.12)\\ &= (x + y)r && (2.13)\\ &= \mathsf{mon}(x + y) && (2.14) \end{aligned}$$

Une remarque importante est que la conversion dans et hors de la représentation de Montgomery n'est pas trop coûteuse. Pour convertir, il faut aussi utiliser l'algorithme 15. Celui ci permet d'effectuer une multiplication modulaire. En l'utilisant, avec les bons paramètres, il permet aussi de convertir dans (Voir Équation 2.15) ou hors (Voir Équation 2.16) de la représentation de Montgomery, mon.

$$x \rightarrow \mathsf{mon}(x) : \begin{cases} \texttt{montgomery}(x, r^2) & = & xr^2r' \\ & = & xr \\ & = & \mathsf{mon}(x) \end{cases} \tag{2.15}$$

$$\mathsf{mon}(x) \rightarrow x : \begin{cases} \texttt{montgomery}(\mathsf{mon}(x), 1) & = & xrr' \\ & = & x \end{cases} \tag{2.16}$$

Le surcoût amené par ces conversions est minime sur le coût total d'une exponentiation modulaire. La complexité de l'algorithme 15 reste l'une des plus intéressante à ce jour. Il effectue trois multiplications. Mais étant donné que seule la partie basse de la deuxième multiplication est utilisée, sa complexité est en pratique inférieure.

$$C_{algo\ 15} = 2Mul_n + MulBas_n + 2Add_n$$

Exemple 7 *Nous reprenons notre exemple :* $21 \times 34 \bmod 43$.

Pour plus de clarté, nous utilisons la base $\beta = 10$. *La puissance supérieure de la base est* $r = 10^2 = 100$. *Il faut précalculer* $p' = 93$ *car* $(-43) \times 93 = 1 \bmod 100$. *Pour entrer dans la représentation* mon*, il faut aussi précalculer* $r^2 \bmod p = 24$.

Nous utilisons l'algorithme 15 pour entrer dans la représentation (Tableau 2.5), faire la multiplication (Tableau 2.6) et sortir de la représentation de mon *de Montgomery (Tableau 2.7).*

$x \rightarrow \mathsf{mon}(x) : \texttt{montgomery}(x, r^2)$		
x	21	34
$c \leftarrow xr^2$	$c \leftarrow 21 \times 24 = 504$	$c \leftarrow 34 \times 24 = 816$
$q \leftarrow cp' \bmod r$	$q \leftarrow 504 \times 93 \bmod 100 = 72$	$q \leftarrow 816 \times 93 \bmod 100 = 88$
$s \leftarrow (c + qp)/r$	$s \leftarrow (504 + 72 \times 43)/100 = 36$	$s \leftarrow (816 + 88 \times 43)/100 = 46$
$\mathsf{mon}(x)$	36	46

TAB. 2.5 – Exemple de conversion dans la représentation de Montgomery.

Les deux nombres que nous désirons multiplier sont maintenant dans la représentation de Montgomery. Nous les multiplions en utilisant l'algorithme de Montgomery (Tableau 2.6).

$\texttt{montgomery}(a, b)$	
$c \leftarrow ab$	$c \leftarrow 36 \times 46 = 1656$
$q \leftarrow cp' \bmod r$	$q \leftarrow 1656 \times 93 \bmod 100 = 8$
$s \leftarrow (c + qp)/r$	$s \leftarrow (1656 + 8 \times 43)/100 = 20$
$\mathsf{mon}(x)$	20

TAB. 2.6 – Exemple de multiplication modulaire avec l'algorithme de Montgomery.

Pour terminer et récupérer le résultat exact, il faut convertir ce résultat hors de la représentation de Montgomery (Tableau 2.7).

$\mathsf{mon}(x) \to x : \mathtt{montgomery}(\mathsf{mon}(x), 1)$	
$\mathsf{mon}(x)$	20
$c \leftarrow \mathsf{mon}(x) \times 1$ $q \leftarrow cp' \bmod r$ $s \leftarrow (c + qp)/r$	$c \leftarrow 20 \times 1 = 20$ $q \leftarrow 20 \times 93 \bmod 100 = 60$ $s \leftarrow (20 + 60 \times 43)/100 = 26$
x	26

Tab. 2.7 – Exemple de conversion hors de la représentation de Montgomery.

L'exemple 2.6 montre que l'algorithme 15 n'est pas pertinent pour une seule multiplication modulaire. Il faut $(i + 2)$ exécutions de l'algorithme 15 pour effectuer i multiplications modulaires. Sur une exponentiation modulaire qui effectue entre n et $2n$ multiplications modulaires, le surcoût des conversions devient négligeable.

$$MulMod \sim (1 + \frac{1}{n})C_{algo\ 15}$$

Lorsque l'on utilise des opérateurs de taille k fixée (opérateurs sur 32 ou 64 bits par exemple), l'algorithme de Montgomery est exécutable bloc par bloc. C'est à dire qu'un nombre a est multiplié par le premier bloc de b et réduit, puis multiplié par le second bloc de b, réduit ... Ceci permet d'effectuer des multiplications modulaires sur des grands nombres $(160, \dots, 1024$ bits) en utilisant des opérateurs sur 32 bits par exemple (Mul_{32} et Add_{32}). L'algorithme 16 utilise des mots de taille k : $\beta = 2^k$. La version par bloc utilise $r = \beta$.

Algorithme 16 : Multiplication Modulaire de Montgomery par bloc

Input : a, b, p avec $0 \leq a, b < p$
Data : p' avec $(-p)p' = 1 \bmod r$
Output : s tel que $s = abr' \bmod p$ avec $rr' = 1 \bmod p$
begin
  $c \leftarrow 0$
  for $i \leftarrow 0$ ***to*** $n - 1$ **do**
    $q_i \leftarrow (s_0 + a_i b_0)p' \bmod r$
    $s \leftarrow (s + a_i b + q_i p)/r$
  end
  if $s \geq p$ **then** $s \leftarrow s - p$
end

La complexité de l'algorithme 16 fait apparaître plus clairement l'intérêt de l'idée de Montgomery : sa complexité est plus proche de deux multiplications $(2n^2)$ que de trois $(3n^2)$.

$$C_{algo\ 16} = (2n^2 + n)Mul_k + (4n^2 + 4n + 2)Add_k$$

2.3.4 L'approximation du quotient par Barrett

En 1986 [14], P. Barrett propose un algorithme qui n'utilise pas de représentation particulière : les entiers sont simplement représentés en base β avec $\beta = 2^k$. Avec l'équation 2.17, il approche le

quotient $q = \left\lfloor \frac{ab}{p} \right\rfloor$ par un calcul sans division.

$$
\begin{aligned}
q &= \left\lfloor \frac{ab}{p} \right\rfloor \\
&= \left\lfloor \frac{\frac{ab}{\beta^{n-1}} \times \frac{\beta^{2n}}{p}}{\beta^{n+1}} \right\rfloor \\
&\simeq \left\lfloor \frac{\left\lfloor \frac{ab}{\beta^{n-1}} \right\rfloor \times \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor}{\beta^{n+1}} \right\rfloor
\end{aligned}
\tag{2.17}
$$

où β est la base utilisée (la division par β est un décalage) et n la taille de p en base β.

Dans [14], Barrett évalue précisément l'erreur sur le quotient.

$$
q - 2 \leq \left\lfloor \frac{\left\lfloor \frac{ab}{\beta^{n-1}} \right\rfloor \times \left\lfloor \frac{\beta^{2n}}{p} \right\rfloor}{\beta^{n+1}} \right\rfloor \leq q \tag{2.18}
$$

Deux soustractions au plus permettent de corriger le résultat final. La division par p est précalculée. Le reste de l'algorithme 17 ne contient que des multiplications ou des décalages, et supprime les divisions coûteuses.

Algorithme 17 : Multiplication Modulaire de Barrett

Input : a, b, p avec $0 \leq a, b < p$
Data : $v = \lfloor \beta^{2n}/p \rfloor$
Output : s avec $s = ab \bmod p$
begin
  $c \leftarrow ab$
  $u \leftarrow \lfloor c/\beta^{n-1} \rfloor$
  $w \leftarrow uv$
  $q \leftarrow \lfloor w/\beta^{n+1} \rfloor$
  $s \leftarrow c - qp \bmod \beta^{n+1}$
  if $s \geq p$ **then** $s \leftarrow s - p$
  if $s \geq p$ **then** $s \leftarrow s - p$
end

Il faut éclaircir l'utilisation du modβ^{n+1} de la cinquième ligne de calcul. D'après l'approximation 2.18 de q, nous avons $0 \leq s < 3p$. C'est pour cela qu'il est inutile de calculer la partie haute de qp. La complexité de l'algorithme 17 est de trois multiplications. Comme pour l'algorithme 15 de Montgomery, nous affinons la complexité de l'algorithme 17. En effectuant l'algorithme 17 bloc par bloc, la complexité s'approche une nouvelle fois des deux multiplications ($2n^2$).

$$C_{algo\ 17} = (2n^2 + 4n)Mul_k$$

Exemple 8 *Nous précalculons* $v = \lfloor \beta^{2n}/p \rfloor = \lfloor 10^4/43 \rfloor = 232$.

Dans le tableau 2.8, nous utilisons l'algorithme 17 pour effectuer la multiplication modulaire de $a = 21$ *par* $b = 34$ *modulo* $p = 43$.

L'algorithme 17 est l'un des plus efficaces lorsque l'utilisateur n'a que quelques réductions à faire car il n'utilise pas de représentations particulières, à la différence de la méthode de Montgomery.

$c \leftarrow ab$	$c = 12.34 = 408$
$u \leftarrow \lfloor c/\beta^{n-1} \rfloor$	$u = \lfloor 408/10 \rfloor = 40$
$w \leftarrow uv$	$w = 40.232 = 9280$
$q \leftarrow \lfloor w/\beta^{n+1} \rfloor$	$q = \lfloor 9280/10^3 \rfloor = 9$
$s \leftarrow c - qp$	$s \leftarrow 408 - 9.43 = 21$
s	21

Tab. 2.8 – Exemple de réduction avec l'algorithme de Barrett.

2.3.5 L'utilisation d'une écriture redondante par Takagi

En 1992, N. Takagi [69] crée un algorithme d'addition modulaire en représentation redondante en utilisant le borrow save (Voir Section 2.1). L'algorithme 18 de multiplication modulaire est stable pour le borrow save.

Cette méthode est inspirée de l'algorithme de Blakley. La totalité des opérations est effectuée en utilisant un système redondant. L'addition de deux nombres redondants, noté ATC, se fait en temps constant (Voir Section 2.1).

Algorithme 18 : Multiplication Modulaire de Takagi

Input : a, b, p avec $0 \leq a, b < p$
Output : s avec $s = ab \bmod p$
begin
  $s \leftarrow 0$
  for $i \leftarrow n$ ***to*** 0 **do**
    $s \leftarrow 2s$
    if $s_{n+1}s_ns_{n-1} < 0$ **then** $s \leftarrow \texttt{ATC}(s, p)$
    else if $s_{n+1}s_ns_{n-1} > 0$ **then** $s \leftarrow \texttt{ATC}(s, -p)$
    $s \leftarrow \texttt{ATC}(s, a_ib)$
    if $s_{n+1}s_ns_{n-1} < 0$ **then** $s \leftarrow \texttt{ATC}(s, p)$
    else if $s_{n+1}s_ns_{n-1} > 0$ **then** $s \leftarrow \texttt{ATC}(s, -p)$
  end
end

L'algorithme 18 est un "double and add" qui réduit à chaque tour pour maintenir s entre $-p$ et p. Comme pour l'addition modulaire, il faut regarder les trois bits de poids fort pour choisir la bonne opération à effectuer.

La complexité de l'algorithme 18 se déduit

$$C_{algo\ 18} = 3(n+1)\texttt{ATC}$$

2.3.6 Étude comparative

Lorsque le modulo est imposé, nous devons utiliser des algorithmes généralistes (qui fonctionnent pour tous les moduli).

Ces algorithmes ne sont pas tous en concurrence ; nous pourrions nous arrêter sur l'algorithme de Taylor qui en 5 additions et deux appels mémoire parait être le plus rapide. En réalité, chaque algorithme possède des avantages et des inconvénients qui les rendent tous intéressants mais dans des contextes différents.

Il existe une différence majeure entre les algorithmes de réduction qui commencent par faire la

multiplication $c = ab$ puis qui réduisent le résultat de cette multiplication et les algorithmes qui réduisent leurs calculs au fur et à mesure de la multiplication. Les algorithmes de cette seconde classe sont dits *intégrés*.

L'intérêt des premiers est que la multiplication, laissée libre, peut être optimisée par des algorithmes classiques de multiplication [34, 35, 62]. Ils sont plutôt conseillés lorsque le modulo devient grand. Les algorithmes généralistes de réduction sont les algorithmes de Montgomery et Barrett (Algorithmes 15 et 17).

A l'inverse, les algorithmes à réduction intégrée sont plus efficaces pour des petits moduli. Les algorithmes à réduction intégrée sont les algorithmes de Blakley, Taylor et Takagi (Algorithmes 13, 14, 18).

L'algorithme de Barrett et celui de Montgomery sont en concurrence direct : ce sont tous deux des algorithmes de réduction généralistes avec précalculs. L'algorithme de Montgomery est certes plus rapide mais il nécessite l'utilisation d'une représentation particulière. Il parait judicieux de conseiller l'algorithme de Barrett dans le cadre d'une simple réduction et de proposer l'algorithme de Montgomery, plus rapide, dans le cadre de calcul plus nombreux (exponentiation modulaire par exemple). Une comparaison plus détaillée existe [16].

Le surcoût d'un précalcul est moins important que le surcoût d'une représentation. Dans la cas d'une exponentiation modulaire par exemple, la représentation donne un surcoût à chaque exponentiation alors que le précalcul ne coûte qu'une seule fois pour toutes les exponentiations sur le même modulo.

Dans le tableau 2.9, nous résumons les différentes caractéristiques de ces algorithmes.

Algorithme	Type	Précalcul	Représentation	Coût
Taylor 13	Intégré	MEM	-	$5Add_n + 2\texttt{MEM}$
Blakley 14	Intégré	-	-	$3nAdd_n$
Montgomery 16	Réduction	Inversion	$\mathsf{mon}(x) = x2^k \bmod p$	$(2n^2 + n)Mul_k$
Barrett 17	Réduction	Division	-	$(2n^2 + 4n)Mul_k$
Takagi 18	Intégré	-	borrow save	$3n\texttt{ATC}$

TAB. 2.9 – Les algorithmes de multiplication modulaire généralistes.

2.4 La famille des nombres de Mersenne

Les besoins arithmétiques de la cryptographie à clé publique sont plus spécifiques que les solutions proposées par l'arithmétique modulaire généraliste. L'implantation des protocoles d'ECC requiert une arithmétique modulaire efficace sur un corps premier. Par contre, ces protocoles imposent uniquement la taille du corps et non pas le type de modulo à utiliser.

Cette possibilité de choix du modulo est utilisée pour accélérer les implantations des protocoles d'ECC. Il existe de nombreux standards proposant des moduli pour différentes tailles cryptographiques [50, 67]. Toutes ces propositions de modulo s'inspirent d'une classe de nombres particulièrement adaptée à l'arithmétique modulaire : les *nombres de Mersenne*. Les classes de moduli proposées sont des extensions ou des généralisations de la classe des nombres de Mersenne. Cette famille de classes de nombres est appelée *la famille des nombres de Mersenne*.

Dans cette partie, nous commençons par présenter les nombres de Mersenne. L'efficacité de ces nombres est très grande. Nous montrons en quoi cette classe de nombre est insuffisante. Ceci est dû à sa trop faible densité. La solution trouvée à ce problème de densité est la classe des nombres

Pseudo Mersenne. Cette classe dense a pour contrepartie d'avoir une arithmétique moins rapide que celle des nombres de Mersenne. Pour concilier la densité et la rapidité, un compromis existe à travers la classe des nombres *Mersenne Généralisés*. D'autres généralisations existent encore.

2.4.1 Les nombres de Mersenne

L'algorithme présenté ici fonctionne pour une classe très particulière de nombres proposés par Knuth en 1981 [35] : les *nombres de Mersenne*.

Définition 5 *Un nombre de Mersenne est un entier p de la forme $p = 2^n - 1$ avec $n \in \mathbb{N}^*$.*

L'intérêt des nombres de Mersenne vient de leur forme si particulière qui offre une équivalence intéressante pour la réduction :

$$2^n \equiv 1 \pmod{p} \tag{2.19}$$

Grâce à cette propriété, la réduction est énormément simplifiée. Puisque la représentation en binaire est la représentation standard en informatique, la division euclidienne par 2^n est équivalente à un simple décalage.

Algorithme 19 : Multiplication Modulaire sur les nombres de Mersenne

---

Input : a, b avec $0 \leq a, b < p$
Data : $p = 2^n - 1$
Output : s avec $s = ab \bmod p$
begin
  $c \leftarrow ab$
  $c = c_1 2^n + c_0$
  $s \leftarrow c_1 + c_0$
  if $s \geq p$ **then** $s \leftarrow s - p$
end

L'algorithme 19 bénéficie de la propriété particulière des moduli sur lesquels il opère. La deuxième ligne de l'algorithme 19 qui devrait coûter une division euclidienne ne coûte en fait rien. Le coût de cet algorithme est d'une multiplication et de deux additions sur n bits.

$$C_{algo\ 19} = Mul_n + 2Add_n$$

Exemple 9 *Le nombre* 31 *est un nombre de Mersenne :* $31 = 2^5 - 1$. *Nous multiplions* $a = 23$ *par* $b = 10$ *modulo* $p = 31$.

Exemple de réduction par un nombre de Mersenne	
$c \leftarrow ab$	$c \leftarrow 230$
$c = c_1 2^n + c_0$	$c = 7.32 + 6$
$s \leftarrow c_1 + c_0$	$s \leftarrow 7 + 6 = 13$

TAB. 2.10 – Exemple de réduction par un nombre de Mersenne.

Les nombres de Mersenne sont une des classes de moduli les plus intéressantes que nous puissions utiliser. La contrepartie de leur très faible coût est la faible densité de cette classe. Il existe au maximum un seul nombre de Mersenne de n bits. Théoriquement, cela suffit. Mais le protocole ECC utilise des nombres premiers. Ce second besoin raréfie les nombres de Mersenne possibles comme nous pouvons le voir avec les propriétés 1 et 2. La propriété 1 montre que les nombres de Mersenne sont les seuls nombres premiers de la forme $\beta^n - 1$.

Propriété 1 *Soit* $p = \beta^n - 1$ *un nombre premier, si* $n \geq 2$ *alors* $\beta = 2$.

Preuve
Un nombres p de forme $\beta^n - 1$ est décomposable :

$$\beta^n - 1 = (\beta - 1)(1 + \beta + \beta^2 + ... + \beta^{n-1})$$

Pour que p soit premier, il est nécessaire que le facteur $(\beta - 1)$ soit égal à 1. Autrement dit nous devons avoir $\beta = 2$. □

Cette première remarque en amène une seconde.

Propriété 2 *Si* $p = 2^n - 1$ *est premier alors* n *est premier.*

Preuve
Supposons que n ne soit pas premier alors il existe $u \geq v \geq 2$ tel que $n = uv$. Nous obtenons que $p = (2^u)^v - 1$. Or avec la propriété 1, nous savons que si p est premier alors $u = 1$. □

En plus d'imposer une faible densité aux nombres de Mersenne, cette seconde propriété prouve qu'il n'existe pas de nombres de Mersenne premiers pour les tailles cryptographiques sur lesquelles nous désirons opérer. Les tailles cryptographiques sont des multiples de 32.

Le tableau 2.11 montre les nombres de Mersenne premiers avec $n \leq 1000$.

Malgré le fait que les nombres de Mersenne ne puissent pas avoir des tailles multiples de 32, le nombre de Mersenne $p_{521} = 2^{521} - 1$ est un des nombres conseillés par le *NIST, National Institute of Standards and Technology* [50], et par le *SEC, Standard for Efficient Cryptography* [67], pour opérer sur les courbes elliptiques (Voir Annexe A.1 et A.2). La rapidité de sa réduction prime sur le fait que sa représentation en base 32 soit potentiellement redondante.

L'unique défaut des nombres de Mersenne tient dans le fait que les nombres de Mersenne premiers soient trop rares.

2.4.2 La classe dense des nombres Pseudo Mersenne

Les nombres de Mersenne ne sont pas en nombre suffisant. Crandall [22] élargit la classe des nombres de Mersenne et propose en 1992 les nombres *Pseudo Mersenne.*

Définition 6 *Un nombre Pseudo Mersenne est un entier* p *de la forme* $p = 2^n - c$ *avec* $n \in \mathbb{N}^*$ *et* $c < 2^{\frac{n}{2}}$.

Comme pour les nombres de Mersenne, nous obtenons une propriété intéressante de la forme particulière des nombres Pseudo Mersenne (Voir Équation 2.20).

$$2^n \equiv c \pmod{p} \tag{2.20}$$

n	Les nombres de Mersenne premiers : $p = 2^n - 1$ avec p premier
2	3
3	7
5	31
7	127
13	8191
17	131071
19	524287
31	2147483647
61	2305843009213693951
89	618970019642690137449562111
107	162259276829213363391578010288127
127	170141183460469231731687303715884105727
521	686479766013060971498190079908139321726943530014330540939446345918554318339765605212255964066145455497729631139148085803712198799971664381257402829111505715 1
607	531137992816767098689588206552468627329593117727031923199444138200403559860852242739162502265229285668889329486246501015346579337652707239409519978766587351943831270835393219031728127

Tab. 2.11 – Les nombres de Mersenne premiers de moins de 1000 bits.

La taille $k = \lfloor \log_2(c) + 1 \rfloor$ de c est capitale. L'algorithme 20 effectue la réduction modulaire d'un entier s sur $2n$ bits (comme le produit d'une multiplication) en un entier r sur n bits.

Algorithme 20 : Réduction Modulaire sur les nombres Pseudo Mersenne

---

Input : s, p avec $0 \leq s < p^2$
Data : $p = 2^n - c$ avec $c < 2^{\frac{n}{2}}$
Output : r tel que $r = s \bmod p$
begin
  Soient s_0 et s_1 tels que $s = s_1 2^{\frac{3n}{2}} + s_0$
  $t \leftarrow s_1 c 2^{\frac{n}{2}} + s_0$
  Soient t_0 et t_1 tels que $t = t_1 2^n + t_0$
  $r \leftarrow t_1 c + t_0$
  if $r \geq p$ **then** $r \leftarrow r - p$
end

Le coût de l'algorithme 20 est donné par l'équation 2.21.

$$\begin{aligned} C_{algo\ 20} &= 2Mul_{\frac{n}{2},\frac{n}{2}} + 3Add_n \\ &= 2Mul_{k,\frac{n}{2}} + 3Add_n \\ &\simeq Mul_{k,n} + 3Add_n \end{aligned} \quad (2.21)$$

En 2000, A. A. Hiasat [31] propose un algorithme pour améliorer la complexité de l'algorithme 20. Celui ci opère sur la classe des nombres Pseudo Mersenne. Cet algorithme part du principe que seule une partie de la réduction prend beaucoup de temps : la réduction de la partie des bits de poids fort. L'idée de Hiasat est d'utiliser une table mémoire pour effectuer la partie la plus lourde de la réduction. Cette partie correspond à la réduction des $k+1$ bits de poids fort du résultat de la multiplication.

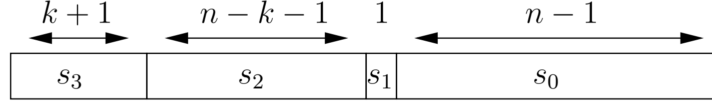

FIG. 2.1 – La décomposition de s pour la réduction de Hiasat.

Hiasat effectue normalement la multiplication $s = ab < 2^{2n}$. Il décompose ensuite la valeur à réduire en quatre parties suivant l'équation 2.22. Cette équation utilise la taille n de p et la taille k de c avec $c = 2^n - p$ (Voir Figure 2.1).

$$s = s_3 2^{2n-k-1} + s_2 2^n + s_1 2^{n-1} + s_0 \quad \text{avec} \begin{cases} s_3 & < & 2^{k+1} \\ s_2 & < & 2^{n-k-1} \\ s_1 & < & 2 \\ s_0 & < & 2^{n-1} \end{cases} \tag{2.22}$$

Nous utilisons l'équation 2.22 pour obtenir plusieurs équivalences modulo p :

$$\begin{array}{rcl} s & \equiv & s_3 2^{2n-k-1} + s_2 2^n + s_1 2^{n-1} + s_0 \pmod p \\ & \equiv & (s_2 c + s_0) + s_3 2^{2n-k-1} + s_1 2^{n-1} \pmod p \end{array} \tag{2.23}$$

L'idée de Hiasat est d'utiliser une table mémoire `MEM` pour effectuer la partie la plus lourde de la réduction. Nous définissons la table mémoire `MEM` selon l'équation 2.24.

$$\mathtt{MEM}(s_3, s_1) = s_3 2^{2n-k-1} + s_1 2^{n-1} \bmod p \tag{2.24}$$

Chaque entrée de cette table mémoire est indicée par $(k + 1) + 1 = k + 2$ bits et contient n bits. Nous obtenons une taille pour `MEM` de $n2^{k+2}$bits. Avec cette table mémoire, nous terminons l'équation d'équivalence de s.

$$\begin{array}{rcl} s & \equiv & s_3 2^{2n-k-1} + s_2 2^n + s_1 2^{n-1} + s_0 \pmod p \\ & \equiv & (s_2 c + s_0) + s_3 2^{2n-k-1} + s_1 2^{n-1} \pmod p \\ & \equiv & (s_2 c + s_0) + \mathtt{MEM}(s_3, s1) \pmod p \end{array} \tag{2.25}$$

Hiasat utilise l'équation 2.25 et propose l'algorithme 21.

Algorithme 21 : Réduction Modulaire de Hiasat

Input : s, p avec $0 \leq s < p^2$
Data : $p = 2^n - c$ et $k = \lfloor \log_2(c) + 1 \rfloor$
Output : r tel que $r = s \bmod p$
begin
  Soient s_0, s_1, s_2 et s_3 tels que $s = s_3 2^{2n-k-1} + s_2 2^n + s_1 2^{n-1} + s_0$
  $r \leftarrow s_2 c + s_0 + \mathtt{MEM}(s_3, s1)$
  if $r \geq p$ **then** $r \leftarrow r - p$
end

L'algorithme 21 utilise l'équation 2.25. Si besoin, il effectue une soustraction finale. Cette soustraction peut être nécessaire pour obtenir un résultat sur n bits :

$$\left.\begin{array}{r} \left.\begin{array}{r} \left.\begin{array}{r} c < 2^k \\ s_2 < 2^{n-k-1} \end{array}\right\} \Rightarrow s_2 c < 2^{n-1} \\ s_0 < 2^{n-1} \end{array}\right\} \Rightarrow s_2 c + s_0 < 2^n \\ \mathtt{MEM}(x, y) < p \end{array}\right\} \Rightarrow s_2 c + s_0 + \mathtt{MEM}(s_3, s1) < 2^n + p \tag{2.26}$$

La complexité de l'algorithme 21 est meilleure que celle de l'algorithme 20.

$$C_{algo\ 21} = Mul_{k,n-k-1} + 3Add_n + \texttt{MEM}$$

Sa contrepartie est la taille #MEM de la table mémoire MEM.

$$\#\texttt{MEM} = n2^{k+2} \text{ bits.}$$

L'algorithme 21 fonctionne pour tous les moduli mais devient réellement efficace lorsque k devient petit. Il correspond bien à un algorithme pour la classe des nombres Pseudo Mersenne.

L'avantage des nombres de Pseudo Mersenne est leur forte densité. Il existe une approximation du nombre $p(x)$ de nombres premiers inférieurs à x [35] :

$$p(x) \sim \frac{x}{\log(x)}$$

Ceci nous permet d'approcher le nombre $N(n)$ de nombres Pseudo Mersenne premiers existant pour une taille n fixée :

$$N(n) = p(2^n) - p(2^n - 2^{\frac{n}{2}})$$

$$N(n) \sim \frac{2^n}{\log(2^n)} - \frac{2^n - 2^{\frac{n}{2}}}{\log(2^n - 2^{\frac{n}{2}})}$$

Sur la figure 2.2, nous voyons que cette approximation est très légèrement inférieure au nombre exact de nombres Pseudo Mersenne premier.

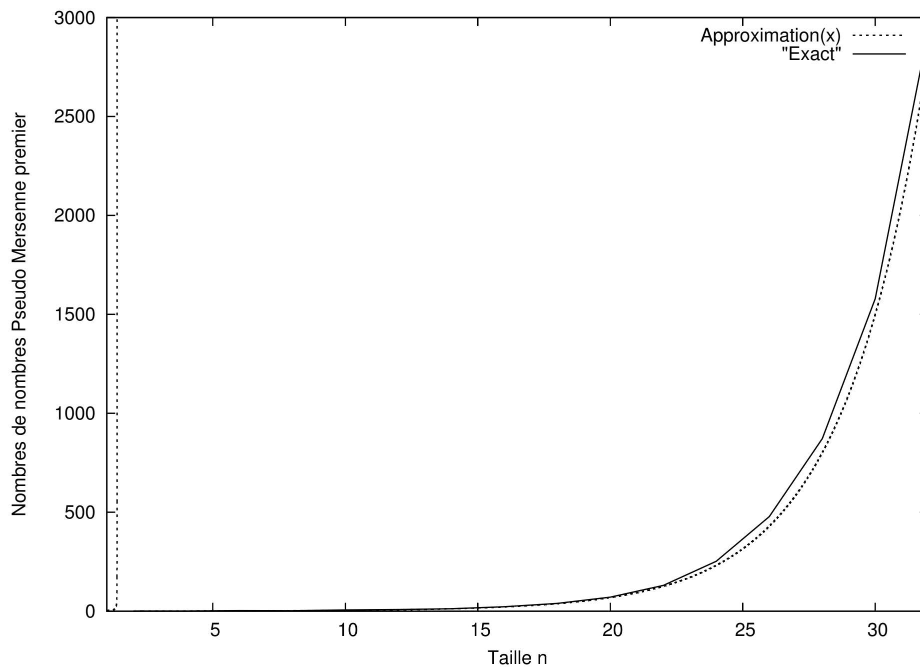

FIG. 2.2 – Le nombre de nombres Pseudo Mersenne premiers.

Pour $n = 50$, le nombre approché de nombres de Pseudo Mersenne premiers est déjà de 960000. Cette approximation nous assure de pouvoir fournir pour toute taille cryptographique souhaitée un grand nombre de nombres Pseudo Mersenne premiers sur lesquels nous pouvons réduire avec deux multiplications de $n/2$ par $n/2$ bits. Ce coût est inférieur au coût de la majorité des algorithmes de réductions modulaires généralistes.

Les nombres Pseudo Mersenne évitent le principal défaut des nombres de Mersenne qui ne sont pas assez nombreux pour pouvoir fournir un nombre de Mersenne premiers pour tout n.

Nous montrons dans le tableau 2.12, pour quelques tailles cryptographiques n, la taille k de c lorsque $2^n - c$ est le nombre Pseudo Mersenne premier le plus proche de 2^n. Il y apparaît des nombres Pseudo Mersenne premiers avec des c de taille suffisamment petite pour permettre l'utilisation d'une réduction modulaire efficace.

n	160	192	224	256	288	320	352	384	416	448	480	512
k du plus petit c	6	8	6	8	8	8	10	9	9	8	6	10
$\#p$ avec $k \leq 8$	4	1	1	1	1	1	0	0	0	2	1	0
$\#p$ avec $k \leq 10$	10	8	6	6	3	7	4	1	5	3	4	4
$\#p$ avec $k \leq 12$	35	33	30	16	16	13	25	13	12	18	13	14
$\#p$ avec $k \leq 16$	527	499	430	365	326	313	260	223	205	215	202	199

Tab. 2.12 – La taille k de c pour différentes tailles cryptographiques.

Les nombres Pseudo Mersenne premiers proposés par le SEC [67] sont donnés dans le tableau 2.13 (Voir Annexe A.2).

Modulo	Valeur
$secp_{160a}$	$2^{160} - (2^{32} + 2^{14} + 2^{12} + 2^9 + 2^8 + 2^7 + 2^3 + 2^2 + 1)$
$secp_{192a}$	$2^{192} - (2^{32} + 2^{12} + 2^8 + 2^7 + 2^6 + 2^3 + 1)$
$secp_{224a}$	$2^{224} - (2^{32} + 2^{12} + 2^{11} + 2^9 + 2^7 + 2^4 + 2 + 1)$
$secp_{256a}$	$2^{256} - (2^{32} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 1)$

Tab. 2.13 – Les nombres Pseudo Mersenne proposés par le SEC [67].

Dans le tableau 2.14, nous montrons le coût de la réduction modulaire pour les nombres Pseudo Mersenne. Nous comptabilisons le nombre d'additions et de multiplications. Pour pouvoir effectuer une comparaison plus précise, nous évaluons aussi le coût des ces algorithmes uniquement par le nombre d'additions. Pour cela, nous évaluons le nombre d'additions nécéssaires à chaque multiplication utilisée par l'algorithme de réduction. Ce nombre est calculable à partir de l'écriture binaire de c : ces multiplications sont des multiplications par des constantes (par c).

	Coût de la réduction			
	Avec Multiplication		Sans Multiplication	
Modulo	Mul_{32}	Add_{32}	Mul_{32}	Add_{32}
$secp_{160a}$	6	7	-	41
$secp_{192a}$	7	8	-	37
$secp_{224a}$	8	9	-	57
$secp_{256a}$	9	10	-	41

Tab. 2.14 – Complexité des réductions par les nombres Pseudo Mersenne du SEC.

2.4.3 Une vision polynomiale : les nombres de Mersenne Généralisés

Les nombres Pseudo Mersenne ajoutent à la multiplication initiale obligatoire plusieurs autres multiplications. Ces multiplications ont un coût inférieur à la multiplication initiale. Elles représentent néanmoins un surcoût non négligeable. Pour éviter ce défaut, Solinas introduit en 1999 [66] une autre classe de moduli : les *nombres de Mersenne Généralisés*.

Définition 7 *Un nombre de Mersenne Généralisé est un entier p de la forme $p = P(2^k)$ où P est un polynôme unitaire de degré n tel que ses coefficients P_i appartiennent à $\{-1, 0, 1\}$ et que son second terme de plus haut degré soit de degré inférieur ou égal à $\frac{n}{2}$.*

La définition 7 se résume par l'équation 2.27.

$$p = P(2^k) \text{ où } P(X) = X^n - C(X) \text{ avec } \left\{ \begin{array}{l} \|C\|_\infty = 1 \\ \deg C \leq \frac{n}{2} \end{array} \right. \tag{2.27}$$

Exemple 10 *Avec le polynôme $P(X) = X^3 - (X + 1)$ et $k = 3$, nous obtenons le nombre premier $p = P(2^3) = 503$.*

Les nombres de Mersenne Généralisés sont particulièrement intéressants car le degré d du second terme de plus haut degré est inférieur à $\frac{n}{2}$. Dans l'exemple 10, le polynôme C est $X + 1$ et son degré est $\deg(C) = 1$.

Tout l'intérêt de l'algorithme 22 tient dans la représentation polynomiale. Les entiers sont représentés sous forme polynomiale. Le polynôme A correspond à l'entier a si $A(\beta) = a$ où $\beta = 2^k$. La conversion dans la représentation ne coûte pas de temps car $\beta = 2^k$ (décalage).

Algorithme 22 : Réduction Modulaire sur les nombres de Mersenne Généralisés

Input : s, p avec $0 \leq s < p^2$
Data : $p = P(2^k)$ où $P = X^n - C(X)$ avec $\deg C \leq \frac{n}{2}$
Output : r tel que $r = s \bmod p$
begin
  Soit $S(\beta)$ la représentation polynomiale de s
  Soient U et L tels que $S = U.X^{\frac{3n}{2}} + L$
  $T \leftarrow UCX^{\frac{n}{2}} + L$
  Soient U', L' tels que $T = U'.X^n + L'$
  $R \leftarrow U'C + L'$
  $r \leftarrow R(2^k)$
  if $r \geq p$ **then** $r \leftarrow r - p$
end

L'algorithme 22 est une adaptation de l'algorithme 20 pour les nombres Pseudo Mersenne. Les nombres de Mersenne Généralisés sont la vision polynomiale des nombres Pseudo Mersenne. La similitude est évidente quand on choisit $k = 1$ et donc $p = P(2)$. Nous obtenons alors l'équivalence des définitions et des algorithmes de réduction. Les nombres de Mersenne Généralisés sont plutôt utilisés avec $k = 32$ qui est une taille de chiffre standard.

Plusieurs nombres de Mersenne Généralisés sont conseillés par le NIST [50] pour l'implantation des protocoles sur les courbes elliptiques (Voir Annexe A.1). Nous détaillons plus précisément leur utilisation et leur coût.

Le nombre de Mersenne Généralisé premier p_{192} est conseillé pour travailler sur un corps dont la taille est de 192 bits.

$$p_{192} = 2^{192} - 2^{64} - 1 \quad (2.28)$$

On obtient $P = X^3 - X - 1$ avec $P(2^{64}) = p_{192}$. L'algorithme 23 est dédié à ce nombre.

Algorithme 23 : Réduction Modulaire sur $p_{192} = 2^{192} - 2^{64} - 1$

Input : s, p avec $0 \leq s < p^2$
Output : r tel que $r = s \bmod p$
begin
- $s = S(2^{64})$
- $A \leftarrow (S_2, S_1, S_0)_X$
- $B \leftarrow (0, S_3, S_3)_X$
- $C \leftarrow (S_4, S_4, 0)_X$
- $D \leftarrow (S_5, S_5, S_5)_X$
- $R \leftarrow A + B + C + D$
- $r \leftarrow R(2^{64}) \bmod p$

end

Nous pouvons vérifier que $S \equiv A + B + C + D \bmod P$.

$$\begin{aligned} S = &\quad S_5\beta^5 + S_4\beta^4 + S_3\beta^3 + S_2\beta^2 + S_1\beta + S_0 \\ S = &\quad S_5(\beta^2 + \beta + 1) + S_4\beta^4(\beta^2 + \beta) + S_3(\beta + 1) + S_2\beta^2 + S_1\beta + S_0 \bmod P \\ S = &\quad D + C + B + A \bmod P \end{aligned}$$

Le nombre de Mersenne Généralisé premier p_{224} est conseillé pour travailler sur un corps dont la taille est de 224 bits.

$$p_{224} = 2^{224} - 2^{96} + 1 \quad (2.29)$$

On obtient $P = X^7 - X^3 - 1$ avec $P(2^{32}) = p_{224}$. L'algorithme 24 est dédié à ce nombre.

Algorithme 24 : Réduction Modulaire sur $p_{224} = 2^{224} - 2^{96} + 1$

Input : s, p avec $0 \leq s < p^2$
Output : r tel que $r = s \bmod p$
begin
- $s = S(2^{32})$
- $A \leftarrow (S_6, S_5, S_4, S_3, S_2, S_1, S_0)_X$
- $B \leftarrow (S_{10}, S_9, S_8, S_7, 0, 0, 0)_X$
- $C \leftarrow (0, S_{13}, S_{12}, S_{11}, 0, 0, 0)_X$
- $D \leftarrow (S_{13}, S_{12}, S_{11}, S_{10}, S_9, S_8, S_7)_X$
- $E \leftarrow (0, 0, 0, 0, S_{13}, S_{12}, S_{11})_X$
- $R \leftarrow A + B + C - D - E$
- $r \leftarrow R(2^{32}) \bmod p$

end

Le nombre de Mersenne Généralisé premier p_{256} est conseillé pour travailler sur un corps dont la taille est de 256 bits.

$$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \quad (2.30)$$

On obtient $P = X^8 - X^7 + X^6 + X^3 - 1$ avec $P(2^{32}) = p_{256}$. L'algorithme 25 est dédié à ce nombre.

Algorithme 25 : Réduction Modulaire sur $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

Input : s, p avec $0 \leq s < p^2$
Output : r tel que $r = s \bmod p$
begin
$s = S(2^{32})$
$A \leftarrow (S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0)_X$
$B \leftarrow (S_{15}, S_{14}, S_{13}, S_{12}, S_{11}, 0, 0, 0)_X$
$C \leftarrow (0, S_{15}, S_{14}, S_{13}, S_{12}, 0, 0, 0)_X$
$D \leftarrow (S_{15}, S_{14}, 0, 0, 0, S_{10}, S_9, S_8)_X$
$E \leftarrow (S_8, S_{13}, S_{15}, S_{14}, S_{13}, S_{11}, S_{10}, S_9)_X$
$F \leftarrow (S_{10}, S_8, 0, 0, 0, S_{13}, S_{12}, S_{11})_X$
$G \leftarrow (S_{11}, S_9, 0, 0, S_{15}, S_{14}, S_{13}, S_{12})_X$
$H \leftarrow (S_{12}, 0, S_{10}, S_9, S_8, S_{15}, S_{14}, S_{13})_X$
$I \leftarrow (S_{13}, 0, S_{11}, S_{10}, S_9, 0, S_{15}, S_{14})_X$
$R \leftarrow A + B + C + D + E - F - G - H - I$
$r \leftarrow R(2^{32}) \bmod p$
end

Le nombre de Mersenne Généralisé premier p_{384} est conseillé pour travailler sur un corps dont la taille est de 384 bits.

$$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \tag{2.31}$$

On obtient $P = X^{12} - X^4 - X^3 + X - 1$ avec $P(2^{32}) = p_{384}$. L'algorithme 26 est dédié à ce nombre.

Algorithme 26 : Réduction Modulaire sur $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

Input : s, p avec $0 \leq s < p^2$
Output : r tel que $r = s \bmod p$
begin
$s = S(2^{32})$
$A \leftarrow (S_{11}, S_{10}, S_9, S_8, S_7, S_6, S_5, S_4, S_3, S_2, S_1, S_0)_X$
$B \leftarrow (0, 0, 0, 0, 0, S_{23}, S_{22}, S_{21}, 0, 0, 0, 0)_X$
$C \leftarrow (S_{23}, S_{22}, S_{21}, S_{20}, S_{19}, S_{18}, S_{17}, S_{16}, S_{15}, S_{14}, S_{13}, S_{12})_X$
$D \leftarrow (S_{20}, S_{19}, S_{18}, S_{17}, S_{16}, S_{15}, S_{14}, S_{13}, S_{12}, S_{23}, S_{23}, S_{21})_X$
$E \leftarrow (S_{19}, S_{18}, S_{17}, S_{16}, S_{15}, S_{14}, S_{13}, S_{12}, S_{20}, 0, S_{23}, 0)_X$
$F \leftarrow (0, 0, 0, 0, S_{23}, S_{22}, S_{21}, 0, 0, 0, 0, 0)_X$
$G \leftarrow (0, 0, 0, 0, 0, 0, S_{23}, S_{22}, S_{21}, 0, 0, S_{20})_X$
$H \leftarrow (S_{22}, S_{21}, S_{20}, S_{19}, S_{18}, S_{17}, S_{16}, S_{15}, S_{14}, S_{13}, S_{12}, S_{23})_X$
$I \leftarrow (0, 0, 0, 0, 0, 0, 0, S_{23}, S_{22}, S_{21}, S_{20}, 0)_X$
$J \leftarrow (0, 0, 0, 0, 0, 0, 0, S_{23}, S_{23}, 0, 0, 0)_X$
$R \leftarrow A + 2B + C + D + E + F + G - H - I - J$
$r \leftarrow R(2^{32}) \bmod p$
end

Dans le tableau 2.15, nous évaluons les complexités des algorithmes proposés par le NIST [50] (Voir Annexe A.1). Pour évaluer la complexité de ces algorithmes, nous comptons le nombre d'additions à faire dans la partie réduction. Nous ne comptons pas le nombre d'additions qui se cachent derrière la reconstruction $s \leftarrow S(2^{32}) \bmod p$.

Modulo	Valeur	Coût en Add_{32}
p_{192}	$2^{192} - 2^{64} - 1$	14
p_{224}	$2^{224} - 2^{96} + 1$	17
p_{256}	$2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	46
p_{384}	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	62

TAB. 2.15 – Coût des réductions par les nombres de Mersenne Généralisés proposés par le NIST.

Grâce à leur forme polynomiale qui ne nécessite pas de multiplication supplémentaire, les nombres de Mersenne Généralisés semblent plus efficaces que les nombres Pseudo Mersenne sur les tailles cryptographiques. Leur complexité en nombre d'additions est relativement faible.

Nous voyons sur le tableau 2.16 que le nombre de Mersenne Généralisés premiers est faible mais suffisant pour certaines tailles cryptographiques. La contrainte $\deg(C) \leq \frac{n}{2}$ peut être relachée. En permettant l'utilisation de polynôme C de plus fort degré, la densité augmente mais le coût de la réduction aussi.

Taille n	160	192	224	256	288	320	352	384	416	448	480	512
$d \leq \frac{n}{2}$	0	1	1	0	0	1	3	2	8	8	13	22
$d > \frac{n}{2}$	1	1	12	20	61	175	502	1288	3733	10368	28713	80380

TAB. 2.16 – Nombres de Mersenne Généralisés premiers avec $k = 32$.

2.4.4 Une extension des nombres de Mersenne Généralisés

En 2003, Chung et Hasan [18] proposent de fusionner les deux classes concurrentes, les nombres Pseudo Mersenne et les nombres de Mersenne Généralisés. Puisque les nombres Pseudo Mersenne semblent plus adaptés pour un travail sur les mots et que les nombres de Mersenne Généralisés semblent plus adaptés au travail sur les grands nombres, Chung et Hasan décident d'utiliser les deux types de propriétés pour proposer une classe encore plus étendue : les *nombres de Mersenne Plus Généralisés*.

Définition 8 *Un nombre de Mersenne Plus Généralisé est un entier p de la forme $p = P(2^k - c)$ avec $c < 2^{\frac{k}{2}}$ où P est un polynôme unitaire de degré n tel que ces coefficients P_i appartiennent à $\{-1, 0, 1\}$ et que son second terme de plus au degré soit de degré $\leq \frac{n}{2}$.*

La définition 8 correspond à la définition des nombres de Mersenne Généralisés mais où le polynôme est évalué en un nombre Pseudo Mersenne.

Exemple 11 *Le polynôme $P(X) = X^4 - X^3 - 1$ évalué en $\beta = 2^4 - 2$ donne $p = P(\beta) = 35671$, qui est un nombre premier.*

2.4.5 Étude comparative

L'utilisation des classes de moduli particuliers est intéressante si l'on a la possibilité de choisir le modulo sur lequel on désire opérer. C'est la cas pour l'utilisation des protocole d'ECC.

Les nombres Pseudo Mersenne et les nombres de Mersenne Généralisés sont tous les deux pertinents mais dans des situations différentes. Les nombres de Mersenne Généralisés paraissent plus adaptés avec leurs forme polynomiale au calcul sur les grands nombres qui sont souvent représentés sous forme polynomiale. Les nombres Pseudo Mersenne sont plutôt conseillés pour opérer sur des chiffres, c'est à dire sur de petits moduli.

Les algorithmes des classes de moduli particuliers sont hiérarchisés en coût mais aussi en densité de la classe concernée. Plus la classe est petite plus elle est rapide. Sur la figure 2.3, nous voyons que moins l'on a de contraintes, plus on monte dans la hiérarchie et plus l'on accélère l'arithmétique modulaire utilisable sur la classe concernée.

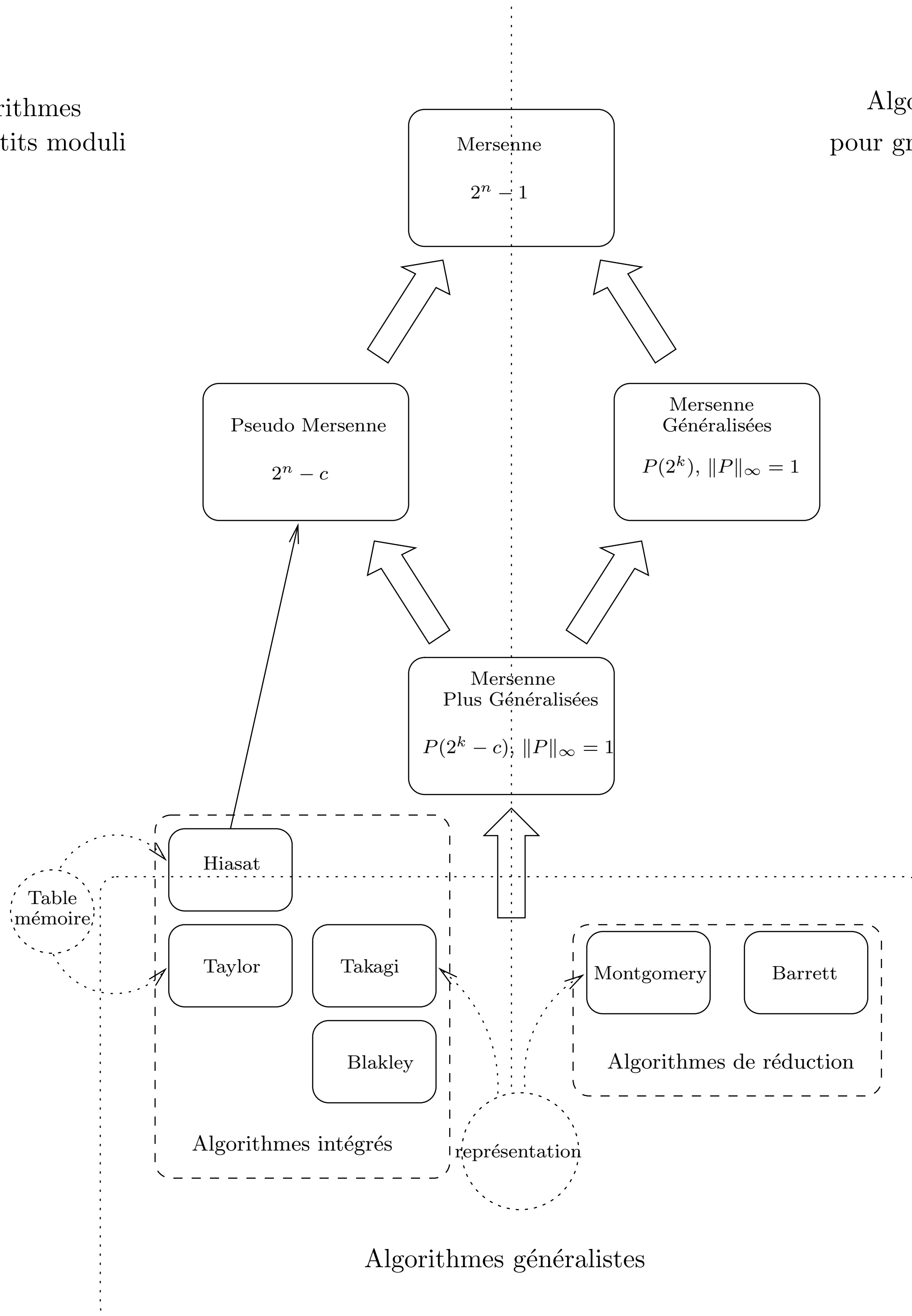

FIG. 2.3 – La hiérarchie des moduli.

Chapitre 3

Systèmes de représentation adaptés à l'arithmétique modulaire

Sommaire

3.1 Définition et propriétés des systèmes de représentation adaptés 48
- 3.1.1 Les systèmes de numération simple de position 48
- 3.1.2 Les systèmes de représentation modulaire 49
- 3.1.3 Les systèmes de représentation adaptés 52

3.2 Algorithmique sur les systèmes de représentation adaptés 53
- 3.2.1 La multiplication 53
- 3.2.2 L'addition 55
- 3.2.3 Les conversions 56
- 3.2.4 Principe de réduction des coefficients 57

3.3 Une classe de moduli optimisée 59
- 3.3.1 Principe de réduction interne 59
- 3.3.2 Minimisation de la matrice de réduction 63
- 3.3.3 Création d'un système de représentation adapté optimisé 64
- 3.3.4 Densité de la classe de moduli 65
- 3.3.5 Exemple dans le système de représentation adapté $\mathcal{B}_{256}$ 67

3.4 Généralisation des systèmes de représentation adaptés 73
- 3.4.1 Les réseaux euclidiens 73
- 3.4.2 Théorème fondamental des systèmes de représentation adaptés 76
- 3.4.3 Utilisation de tables mémoire 82
- 3.4.4 Transcription d'algorithmes connus dans les systèmes de représentation adaptés 85

Les représentations des nombres utilisées jusqu'à présent en arithmétique modulaire sont des systèmes classiques de représentation des nombres. Les algorithmes de calculs modulaires sur ces systèmes utilisent différentes méthodes classiques d'arithmétique en calcul modulaire (Voir Section 2.3). L'autre solution utilisée est de trouver des moduli adaptés à la représentation de position : c'est la "famille des Mersenne" (Voir Section 2.4). Nous proposons non plus de chercher des moduli adaptés au système de représentation utilisé mais de chercher des systèmes de représentation adaptés au modulo choisi. Ces systèmes permettent de créer au choix une classe de moduli pour ECC ou une arithmétique généraliste pour RSA.

Dans ce chapitre, nous commençons par définir le système de représentation que nous allons utiliser ainsi que tous les systèmes de représentation qui sont nécessaires à sa compréhension. Nous présentons aussi les premières propriétés fondamentales que nous avons sur ce nouveau système de représentation.

Cette représentation introduite, nous proposons les algorithmes nécessaires pour calculer : nous montrons comment additionner ou multiplier dans ce système mais aussi comment passer d'une représentation classique de position à cette représentation et inversement. Les algorithmes que nous proposons font appel à une opération primordiale des systèmes de représentation adaptés : *la réduction des coefficients*. Les deux parties qui terminent ce chapitre proposent différentes solutions pour effectuer cette opération de réduction des coefficients.

Pour réduire le surcoût dû à la réduction des coefficients, une première idée inspirée des besoins réels de la cryptographie à clé publique est de proposer une classe de moduli pour laquelle nous pouvons le borner. Cette classe est composée de moduli sur lesquels l'arithmétique proposée concurrence la "famille des Mersenne" : Mersenne, Pseudo Mersenne, Mersenne Généralisés, ... (Voir Section 2.4). Ce travail a été publié dans [10].

Il existe des protocoles (RSA par exemple) pour lesquels le modulo sur lequel ils opèrent ne peut être choisi librement. Dans une dernière partie, nous proposons une arithmétique généraliste sur les systèmes de représentation adaptés ; celle ci est efficace pour tous les moduli. La solution proposée utilise la théorie des réseaux Euclidiens. Celle ci permet dans un premier temps de borner notre représentation à des tailles raisonnables. Nous terminons par une proposition de plusieurs types d'algorithmes généralistes de réduction des coefficients. Ce travail a été publié dans [11].

3.1 Définition et propriétés des systèmes de représentation adaptés

Avant de définir les systèmes de représentation adaptés, nous rappelons ce qu'est la représentation par système de numération simple de position. Ensuite, nous introduisons les systèmes de représentation modulaires ainsi que différents critères permettant de les analyser. Ce nouveau système modulaire sert de base pour pouvoir créer un système de représentation adapté. Nous terminons par la définition du système de représentation adapté.

3.1.1 Les systèmes de numération simple de position

Le système de numération simple de position est définie par un entier β appelé base et éventuellement une taille n. On peut représenter tout entier x compris entre 0 et $\beta^n - 1$ par un vecteur $(x_{n-1}, \dots, x_0)_\beta$ (Voir Équation 3.1)

$$x = \sum_{i=0}^{n-1} x_i \beta^i \text{ avec } 0 \leq x_i < \beta \tag{3.1}$$

Les x_i sont appelés *chiffres*. Par exemple, $(1, 1, 0)_2$ correspond en binaire (base 2) à l'entier 6.

Cette représentation peut devenir *redondante* en élargissant le choix des chiffres.

Définition 9 *Un système de représentation est redondant si au moins deux de ces représentations correspondent à la même valeur.*

Si l'on utilise un système où les chiffres respectent $0 \leq x_i < \beta$ pour représenter un nombre modulo p alors ce système devient redondant dès que $p < \beta^n$. Par exemple, si l'on choisit $p = 13$

alors $(3,2)_4 \pmod{13}$ et $(0,1)_4 \pmod{13}$ sont deux représentations différentes d'une même valeur : 1 mod 13.

3.1.2 Les systèmes de représentation modulaire

Nous utilisons le fait qu'un nombre est représenté modulo p. Nous introduisons la notion de *système de représentation modulaire.*

Définition 10 *Un système de représentation modulaire est défini par un quadruplet (p, n, γ, ρ) tel que, pour tout entier $0 \leq x < p$, il existe n entiers x_i satisfaisant*

$$x \equiv \sum_{i=0}^{n-1} x_i \gamma^i \pmod{p}, \quad \textit{avec } 0 \leq x_i < \rho \tag{3.2}$$

Remarquons que si l'on fixe p et n alors tous les couples γ, ρ ne permettent pas de construire un système de représentation modulaire. Par la suite, nous rechercherons les couples γ, ρ qui possède des propriétés intéressantes pour l'arithmétique modulaire.

Exemple 12 *Nous choisissons $p = 53$ et $n = 3$ et nous désirons représenter tous les entiers entre 0 et 52 avec trois chiffres. Observons sur le tableau 3.1 que certains couples (γ, ρ) donnent un nombre de représentations suffisantes. Pourtant, ils ne représentent pas tous les nombres compris entre 0 et 52.*

γ, ρ	$29, 4$	$31, 5$	$30, 6$
Nombre de représentations : ρ^n	64	125	216
Nombre d'entiers représentables	47	53	48
Entiers non représentables	4, 12, 19, 26, 43, 50	-	20, 21, 22, 23, 24

TAB. 3.1 – Quelques propositions de couples γ, ρ pour $p = 53, n = 3$.

Dans le tableau 3.1, $\mathcal{B} = (p = 53, n = 3, \gamma = 31, \rho = 5)$ est un système de représentation modulaire. Tout entier compris entre 0 et 52 est représentable en utilisant l'équation 3.2.Les entiers représentables et leur codage sont présentés dans le tableau 3.2.

On peut aisément vérifier que $(3, 2, 4)_{\mathcal{B}}$ représente bien 34 :

$$4 + 2.31 + 3.31^2 \equiv 34 \pmod{53} \tag{3.3}$$

Nous voyons que $(2, 4, 2)_{\mathcal{B}}$ représente aussi 34 :

$$2 + 4.31 + 2.31^2 \equiv 34 \pmod{53} \tag{3.4}$$

Cette écriture est redondante.

Définition 11 *La relation d'équivalence notée $u \overset{\mathcal{B}}{\equiv} v$ est telle que u et v représentent le même élément de $\mathbb{Z}/p\mathbb{Z}$ dans le système de représentation modulaire $\mathcal{B} = (p, n, \gamma, \rho)$.*

$$u \overset{\mathcal{B}}{\equiv} v \iff \sum_{i=0}^{n-1} u_i \gamma^i \equiv \sum_{i=0}^{n-1} v_i \gamma^i \pmod{p} \tag{3.5}$$

0	1	2	3	4	5	6	7	8	9
(0, 0, 0)	(0, 0, 1)	(0, 0, 2)	(0, 0, 3)	(0, 0, 4)	(2, 3, 4)	(4, 1, 0)	(1, 0, 0)	(1, 0, 1)	(0, 2, 0)
(3, 1, 1)	(2, 3, 0)	(2, 3, 1)	(2, 3, 2)	(2, 3, 4)			(4, 1, 1)	(3, 3, 0)	(1, 0, 2)
	(3, 1, 2)	(3, 1, 3)	(3, 1, 4)					(4, 1, 2)	(3, 3, 1)
									(4, 1, 3)

10	11	12	13	14	15	16	17	18	19
(0, 2, 1)	(0, 2, 2)	(0, 2, 3)	(0, 2, 4)	(2, 0, 0)	(2, 0, 1)	(1, 2, 0)	(1, 2, 1)	(0, 4, 0)	(0, 4, 1)
(1, 0, 3)	(1, 0, 4)	(3, 3, 4)			(4, 3, 0)	(2, 0, 2)	(2, 0, 3)	(1, 2, 2)	(1, 2, 3)
(3, 3, 2)	(3, 3, 3)					(4, 3, 1)	(4, 3, 2)	(2, 0, 4)	(4, 3, 4)
(4, 1, 4)								(4, 3, 3)	

20	21	22	23	24	25	26	27	28	29
(0, 4, 2)	(0, 4, 3)	(0, 4, 4)	(2, 2, 0)	(2, 2, 1)	(1, 4, 0)	(1, 4, 1)	(1, 4, 2)	(1, 4, 3)	(1, 4, 4)
(1, 2, 4)	(3, 0, 0)	(3, 0, 1)	(3, 0, 2)	(3, 0, 3)	(2, 2, 2)	(2, 2, 3)	(2, 2, 4)	(4, 0, 0)	(4, 0, 1)
	(3, 3, 3)				(3, 0, 4)				

30	31	32	33	34	35	36	37	38	39
(3, 2, 0)	(0, 1, 0)	(0, 1, 1)	(0, 1, 2)	(0, 1, 3)	(0, 1, 4)	(2, 4, 4)	(4, 2, 0)	(1, 1, 0)	(1, 1, 1)
(4, 0, 2)	(3, 2, 1)	(2, 4, 0)	(2, 4, 1)	(2, 4, 2)	(2, 4, 3)			(4, 2, 1)	(3, 4, 0)
	(4, 0, 3)	(3, 2, 2)	(3, 2, 3)	(3, 2, 4)					(4, 2, 2)
		(4, 0, 4)							

40	41	42	43	44	45	46	47	48	49
(0, 3, 0)	(0, 3, 1)	(0, 3, 2)	(0, 3, 3)	(0, 3, 4)	(2, 1, 0)	(2, 1, 1)	(1, 3, 0)	(1, 3, 1)	(1, 3, 2)
(1, 1, 2)	(1, 1, 3)	(1, 1, 4)	(3, 4, 4)			(4, 4, 0)	(2, 1, 2)	(2, 1, 3)	(2, 1, 4)
(3, 4, 1)	(3, 4, 2)	(3, 4, 3)					(4, 4, 1)	(4, 4, 2)	(4, 4, 3)
(4, 2, 3)	(4, 2, 4)								

50	51	52
(1, 3, 3)	(1, 3, 4)	(3, 1, 0)
(4, 4, 4)		

Tab. 3.2 – L'ensemble des éléments du système de représentation modulaire $\mathcal{B} = (p = 53, n = 3, \gamma = 31, \rho = 5)$.

Si nous reprenons notre exemple où $\mathcal{B} = (53, 3, 31, 5)$ alors nous avons $(3, 2, 4) \overset{\mathcal{B}}{\equiv} (2, 4, 2)$.

Nous remarquons que si $\mathcal{B} = (p, n, \gamma, \rho)$ est un système de représentation modulaire alors $\mathcal{B}' = (p, n, \gamma, \rho + 1)$ l'est aussi. Une fois p, n, γ fixés, le plus intéressant est de prendre comme valeur pour ρ le plus petit des entiers tel que (p, n, γ, ρ) soit un système de représentation modulaire. Nous notons cet entier ρ_{min}.

Exemple 13 *Nous reprenons l'exemple déjà vu, $p = 53$ et $n = 3$. La figure 3.1 montre les valeurs de ρ_{min} en fonction de γ. Le fait que $p - \gamma$ soit l'opposé de γ rend cette figure symétrique.*

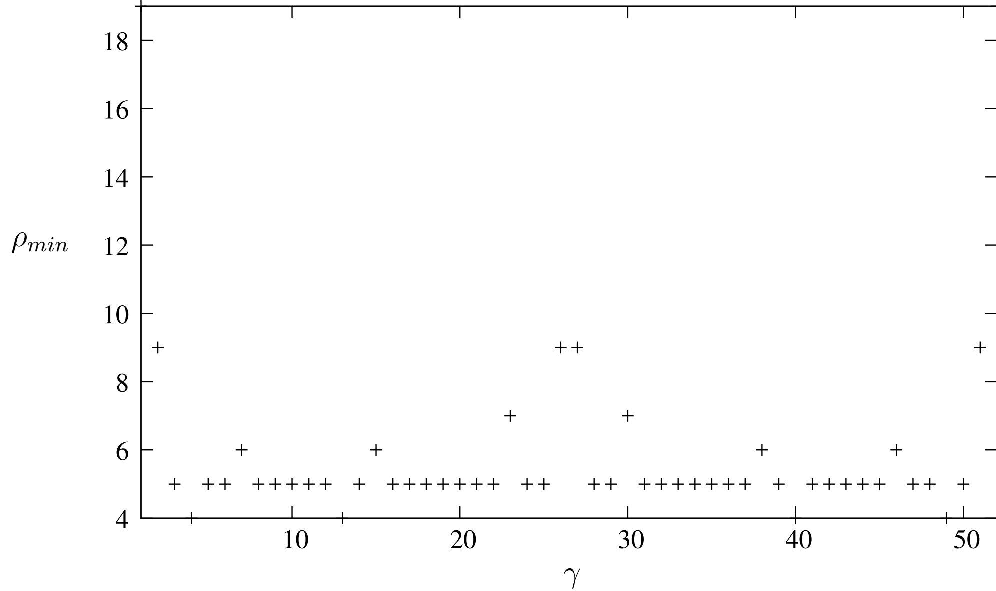

FIG. 3.1 – Valeurs de ρ_{min} tel que $\mathcal{B} = (53, 3, \gamma, \rho_{min})$ soit un système de représentation modulaire.

Propriété 3 *Soit $\mathcal{B} = (p, n, \gamma, \rho)$ un système de représentation modulaire ; alors $\lceil p^{1/n} \rceil \leq \rho$.*

<u>*Preuve*</u>
Nous désirons représenter les entiers compris entre 0 et $p - 1$, soit p valeurs. Un système de représentation modulaire $\mathcal{B} = (p, n, \gamma, \rho)$ permet de créer ρ^n représentations. Il permet donc de représenter au maximum ρ^n valeurs. Nous en déduisons que $p \leq \rho^n$. □

L'évaluation ρ_{min} est une question cruciale pour l'utilisation des systèmes de représentation modulaires. Ce système de représentation ne doit pas être trop fortement redondant : la taille nécessaire à sa représentation ne doit pas être prohibitive. Nous définissons un critère de qualité : le *rapport de redondance*.

Définition 12 *Le rapport de redondance, noté π, est donné par l'équation 3.6.*

$$\pi = \log_2 \frac{\rho}{\lceil p^{1/n} \rceil} \tag{3.6}$$

Ce rapport permet d'évaluer la redondance d'un système de représentation. Si $\pi = 0$, le système de représentation n'est pas redondant. Si $\pi > 0$ le surcoût de la représentation est de π bits par chiffre. Par la suite, l'évaluation de π sera donc récurrente.

3.1.3 Les systèmes de représentation adaptés

Le système de représentation modulaire ne possède pas initialement de propriété intéressante pour l'arithmétique modulaire. Par contre, il sert de base pour définir la notion de *système de représentation adapté.*

Définition 13 *Un système de représentation adapté* $(p, n, \gamma, \rho)_c$ *est un système de représentation modulaire* (p, n, γ, ρ) *vérifiant*

$$\gamma^n \equiv c \pmod p \tag{3.7}$$

où c *est un "petit" entier.*

Comme pour la classe des Pseudo Mersenne, le fait que c soit "petit" accélère la réduction.

Pour offrir encore plus de flexibilité aux systèmes de représentation modulaires, nous proposons de donner une généralisation de la définition des systèmes de représentation adaptés. Comme pour la classe des Mersenne généralisés, nous définissons le système de représentation adapté généralisé.

Définition 14 *Un système de représentation adapté généralisé* $(p, n, \gamma, \rho)_E$ *est un système de représentation modulaire* (p, n, γ, ρ) *caractérisé par un polynôme* E *satisfaisant les trois critères suivants :*

a) γ *est une racine du polynôme* E *modulo* p*.*

b) le degré du polynôme E *est égal à* n*.*

c) le polynôme E *est un polynôme unitaire "intéressant" pour la réduction (avec ces coefficients petits ou nuls).*

Nous retrouvons le système de représentation adapté comme étant un système de représentation adapté généralisé où $E(X) = X^n - c$ est le polynôme qui caractérise le système de représentation adapté. Comme première généralisation, nous pouvons considérer le polynôme $E(X) = X^n - (aX + b)$.

Dans un système de représentation adapté (p, n, γ, ρ), nous représentons l'entier a par un polynôme $A \in \mathbb{Z}(X)$ tel que $A(\gamma) \equiv a \pmod p$ et $\deg A < n$. Ses coefficients A_i sont tels que $-\rho < A_i < \rho$: nous autorisons les chiffres signés.

Définition 15 *L'ensemble des représentations, noté* $a_{\mathcal{B}}$*, de l'entier* a *dans un système de représentation adapté* $\mathcal{B} = (p, n, \gamma, \rho)$ *est défini par l'équation 3.8.*

$$A \in a_{\mathcal{B}} \iff \begin{cases} A(\gamma) \equiv a \pmod p \\ \deg A < n \\ \|A\|_\infty < \rho \end{cases} \tag{3.8}$$

où $\|.\|_\infty$ *est la norme infinie. Dans la suite, si* A *est une représentation de* a *dans le système* $\mathcal{B}$*, nous le notons* $A \equiv a_{\mathcal{B}}$*.*

La norme infinie d'un polynôme A, notée $\|A\|_\infty$, est le maximum des valeurs absolues des coefficients du polynôme A.

$$\|A\|_\infty = \max_{0 \le i \le \deg A} A_i$$

Par la suite, nous noterons k la taille des coefficients : $2^{k-1} < \rho \le 2^k$.

3.2 Algorithmique sur les systèmes de représentation adaptés

Dans cette partie, nous présentons les algorithmes nécessaires à une arithmétique sur le système de représentation que nous proposons. Nous développons les deux opérations de base d'une arithmétique : l'addition et la multiplication. Pour que ce système soit utilisable, nous devons aussi fournir les moyens de convertir un élément représenté dans un système de position classique dans ce système et inversement.

La multiplication modulaire étant l'opération la plus utilisée de l'arithmétique modulaire pour la cryptographie, nous exposons dans un premier temps comment cette opération s'effectue sur les systèmes de représentation adaptés. Nous effectuons la multiplication sous forme polynomiale. Puis nous effectuons deux types de réduction, pour obtenir un polynôme qui appartient à notre représentation, autrement dit pour que cette représentation soit stable par la multiplication. La première est la réduction du degré du polynôme, appelée *réduction externe*. Dans les systèmes de représentation adaptés, celle-ci est fortement minimisée par l'aspect "Pseudo Mersenne" du système de représentation adapté. La seconde est la réduction des coefficients qui correspond à une propagation de retenue dans une représentation classique. Pour l'arithmétique dans les systèmes de représentation adaptés, cette opération devient cruciale. Elle est aussi utilisée lors de l'addition modulaire dans les systèmes de représentation adaptés et pour les conversions des représentations de ce système.

Nous exposons une méthode générale de la réduction des coefficients. Celle-ci n'est explicitée que dans les sections 3.3 et 3.4.

3.2.1 La multiplication

Nous proposons l'algorithme suivant de multiplication modulaire sur les systèmes de représentation adaptés (Voir Algorithme 27).

Algorithme 27 : `MulMod`(A,B,$\mathcal{B}$), Multiplication sur un système de représentation adapté

```
Input  : A, B ≡ a_B, b_B avec B = (p, n, γ, ρ)_E
Output : S ≡ s_B avec s ≡ ab (mod p)
begin
    U ← MulPol(A, B)
    V ← RedExt(U, E)
    S ← RedCoef(V, B)
end
```

Observons plus en détail les trois étapes de l'algorithme 27.

1. `MulPol` est une multiplication polynomiale. Nous avons $U(\gamma) \equiv A(\gamma)B(\gamma) \equiv ab \equiv s \pmod{p}$. Le polynôme U respecte donc la première ligne de l'équation 3.8. Toutefois, après la multiplication polynomiale (`MulPol`), U est un polynôme de degré $\deg(U) < 2n - 1$. Ses coefficients peuvent être supérieurs à ρ. Mais nous avons $\|U\|_\infty < n\rho^2$.

   Au delà de l'algorithme "scolaire" [35], de nombreux algorithmes de multiplication polynomiale sont proposés dans la littérature. Ils se décomposent généralement en trois étapes : l'évaluation des polynômes A et B en différents points, la multiplication des valeurs interpolées et la reconstruction du polynôme quotient U. Cette méthode est initialement proposée par Karatsuba [34]. Toom et Cook [35], puis Schönhage et Strassen [62] améliorent l'idée initiale en prenant d'autre choix de points. Dans le tableau 3.3, nous donnons les complexités de ces différents algorithmes ainsi que les valeurs de n pour lesquelles le choix des algorithmes concernés devient intéressant. Ces valeurs sont tirées de la thèse de P. Giorgi [26] après analyse

des choix d'implantations faits pour la librairie de calcul GMP, pour "GNU Multiprecision Package".

`MulPol`	Nombre d'opérations	Utilisation
Standard [35]	$O(n^2)$	n < 10
Karatsuba [34]	$O(n^{\log_2 3}) \sim O(n^{1.585})$	$10 \leq$ n < 300
Toom-Cook [35]	$O(n^{\log_3 5}) \sim O(n^{1.465})$	$300 \leq$ n < 1000
Schönhage-Strassen [62]	$O(n \log n \log \log n)$	$1000 \leq$ n

Tab. 3.3 – Complexités des différentes méthodes de multiplication polynomiale.

La complexité de l'algorithme scolaire de multiplication est de

$$C_{\texttt{MulPol}} = n^2 Mul_k + (n-1)^2 Add_{2k}$$

Puisque nous avons $\deg A, \deg B < n$ et $\|A\|_\infty, \|B\|_\infty < \rho$, le polynôme U est tel que $\deg U < 2n-1$ et $\|U\|_\infty < n\rho^2$.

2. `RedExt` est ce que nous appelons par la suite la *réduction externe*. Elle correspond à une réduction polynomiale du polynôme U par E ; $V = U \pmod E$. Elle permet de réduire le degré du polynôme. Cette réduction correspond à la deuxième condition donnée dans l'équation 3.8 pour le système de représentation adapté $\mathcal{B} = (p, n, \gamma, \rho)_E$. Comme vu dans l'équation 3.8, il faut que $\deg S < n$ pour que $S \equiv s_{\mathcal{B}}$. Or le résultat de la première ligne retourne un polynôme U avec $\deg(U) < 2n-1$. La réduction externe coûte peu vu la forme très particulière du polynôme $E(X) = X^n - c$ qui donne une équivalence utile : $X^n \equiv c \pmod E$.

Algorithme 28 : `RedExt`(U,E), Réduction Externe

```
Input  : U ≡ Z[X] avec deg(U) < 2n − 1 et E(X) = X^n − c
Output : V ≡ Z[X] avec V = U mod E
begin
    for i ← 0 to n − 2 do
        V_i ← U_i + cU_{i+n}
    end
    V_{n−1} ← U_{n−1}
end
```

La complexité de l'algorithme 28 dépend de l'élément c. Nous décidons donc de restreindre c à des valeurs pour lesquelles la multiplication ne coûte rien : $c \in \{1, -1, 2, -2, 4, \ldots\}$. C'est à dire que nous utilisons comme valeurs pour c, de petites puissances de 2 ($\pm 2^t$) pour lesquelles la multiplication se réduit à un décalage. Dans ces conditions, nous évaluons la complexité de l'algorithme 28.

$$C_{\texttt{RedExt}} = (n-1) Add_{2k}$$

Après cette réduction polynomiale, nous avons $V \equiv AB \pmod E$. C'est à dire que $V = AB + QE$. Nous avons donc $V(\gamma) = A(\gamma)B(\gamma) + Q(\gamma)E(\gamma)$. Or $E(\gamma) \equiv 0 \pmod p$. Nous avons $V(\gamma) \equiv A(\gamma)B(\gamma) \pmod p$. Nous avons aussi $\deg V < n$. Le polynôme V respecte les deux premières lignes de l'équation 3.8. La deuxième ligne de l'équation 3.8 est aussi respectée.

Après l'utilisation de `MulPol` et de `RedExt`, les coefficients de V peuvent être supérieurs à ρ. Nous avons $\|V\|_\infty < cn\rho^2$. Nous pouvons retrouver ce résultat avec l'équation 3.9.

$$\text{Pour tout } i \text{ de } [0, n-1], V_i = \sum_{j=0}^{i} A_{i-j}B_j + \sum_{j=1}^{n-i-1} cA_{i+j}B_{n-j} \tag{3.9}$$

A et B sont des représentations du système de représentation adapté $\mathcal{B} = (p, n, \gamma, \rho)$, leurs coefficients sont tels que $\|A\|_\infty, \|B\|_\infty < \rho$. L'équation 3.9 montre que chaque coefficient du polynôme V est la somme de n produits. Ces produits sont majorés par $c\rho^2$. Nos obtenons que $\|V\|_\infty < nc\rho^2$.

Le calcul de la multiplication et de la réduction polynomiale de l'algorithme 27 peuvent se faire en une seule étape. De plus, chaque coefficient du polynôme V peut être calculé indépendamment des autres : nous pouvons paralléliser cette étape.

3. `RedCoef` est ce que nous appelons la *réduction des coefficients*. `RedCoef` permet de construire un polynôme S équivalent au polynôme V ($S \overset{\mathcal{B}}{\equiv} V$) tel que $\|S\|_\infty < \rho$. Cette réduction permet de satisfaire la dernière condition de l'équation 3.8, c'est à dire la majoration que doivent respecter les coefficients de S, $\|S\|_\infty < \rho$. Après l'utilisation de `MulPol` et de `RedExt`, nous avons $\|V\|_\infty < cn\rho^2$. Il faut réduire les coefficients de V pour avoir un polynôme $S \overset{\mathcal{B}}{\equiv} V$ tel que $S \equiv s_\mathcal{B}$ avec $s = ab$.

   L'algorithme de réduction des coefficients `RedCoef` est primordial pour l'algorithmique des systèmes de représentation adaptés. Nous ne le définissons que dans la dernière partie de cette section. Il existe de nombreuses possibilités pour réduire les coefficients.

Nous donnons les algorithmes des opérations de base de l'arithmétique modulaire sur un système de représentation adapté.

3.2.2 L'addition

Comme pour le produit, l'addition se décompose en une opération polynomiale suivie d'une réduction des coefficients. Ici, comme lors de l'addition polynomiale, les degrés des polynômes n'augmente pas, la réduction externe n'est pas nécessaire

Algorithme 29 : `AddMod`(A,B,$\mathcal{B}$), Addition sur un système de représentation adapté

Input : $A, B \equiv a_\mathcal{B}, b_\mathcal{B}$ et $\mathcal{B} = (p, n, \gamma, \rho)_E$
Output : $S \equiv s_\mathcal{B}$ avec $s \equiv a + b \pmod p$
begin
  $U \leftarrow A + B$
  $S \leftarrow$ `RedCoef`$(U, \mathcal{B})$
end

Il faut vérifier que S est la représentation de $s = a + b \bmod p$. Or comme $U = A + B$, on a $s = S(\gamma) \equiv U(\gamma) \equiv A(\gamma) + B(\gamma) \equiv a + b \pmod p$. La réduction externe est inutile. Puisque $A, B \equiv a_\mathcal{B}, b_\mathcal{B}$ alors $\deg(A), \deg(B) < n$ et avec $U \leftarrow A + B$ on obtient $\deg(U) < n$. Les coefficients sont ensuite réduits par l'algorithme `RedCoef`. Dans ce cas précis, les coefficients de U sont tels que $\|U\|_\infty < 2\rho$. La réduction des coefficients sera simplifiée par ce faible débordement. La complexité de l'addition est, comme pour la multiplication, fortement dépendante de celle de l'algorithme `RedCoef`.

3.2.3 Les conversions

Suivant l'utilisation, il peut être nécessaire de convertir de ou vers un système classique de représentation. Nous devons donc transformer un élément représenté dans un système de numération simple de position en son représentant dans un système de représentation adapté et inversement.

Nous proposons deux algorithmes pour trouver à partir de l'entier a (représenté dans un système de représentation classique) un polynôme $A \equiv a_{\mathcal{B}}$ avec $\mathcal{B} = (p, n, \gamma, \rho)_E$.

Le polynôme A dont tous les coefficients sont nuls sauf $A_0 = a$, respecte les deux premiers critères de l'équation 3.8 d'appartenance à un système de représentation adapté : $A(\gamma) = a$ et $\deg(A) = 0 < n$. Pour obtenir une représentation du système de représentation adapté, il suffit alors de retourner $A \leftarrow \texttt{RedCoef}(A, \mathcal{B})$ alors nous obtenons $A \equiv a_{\mathcal{B}}$. Le polynôme A est tel que $\|A\|_\infty < p$. On a donc des coefficients très grands à réduire, ce qui sera plus long pour `RedCoef`. Cette première méthode ne semble pas optimale.

Nous supposons que nous recevons a dans un système de numération simple de position. Nous représentons a en base β tel que $p < \beta^n$. Nous considérons comme connus (précalculés) les représentations $P_i(X)$ des puissances de la base β. Nous avons $P_i \equiv (\beta^i)_{\mathcal{B}}$. Ces représentations sont calculables avec `RedCoef` (Voir Équation 3.10).

$$\text{Pour tout } i \text{ tel que } 0 \leq i < n, \quad P_i \leftarrow \texttt{RedCoef}(\beta^i) \tag{3.10}$$

Algorithme 30 : Conversion dans un système de représentation adapté

Input : a et $\mathcal{B} = (p, n, \gamma, \rho)_E$
Data : Pour tout i tel que $0 \leq i < n, P_i \equiv (\beta^i)_{\mathcal{B}}$
Output : $A \equiv a_{\mathcal{B}}$
begin
  $a = (a_{n-1}, \ldots, a_0)_\beta$
  $U \leftarrow \sum_{i=0}^{n-1} a_i.P_i(X)$
  $A \leftarrow \texttt{RedCoef}(U, \mathcal{B})$
end

Puisque les polynômes $P_i(X) \equiv (\beta^i)_{\mathcal{B}}$, leurs coefficients sont inférieurs à ρ et puisque $0 \leq a_i < \beta$, la somme $\sum_{i=0}^{n-1} a_i.P_i(X)$ donne un polynôme U tel que $\|U\|_\infty < n\beta\rho$. Si l'on choisit $\beta = 2^k \simeq \rho$, alors les coefficients de U sont du même ordre de grandeur que ceux du polynôme V à réduire lors de la multiplication modulaire dans un système de représentation adapté dont les coefficients sont bornés par $cn\rho^2$. Cette remarque est importante, car moyennant quelques mémorisations $\sim n - 2$ (la mémorisation de $P_0 \equiv 1_{\mathcal{B}}$ et de $P_1 \equiv \beta_{\mathcal{B}}$ est inutile), le coût de conversion dans la représentation est équivalent à un produit dans un système de représentation adapté. Ce coût est très faible (voire négligeable) dans le contexte d'une arithmétique modulaire pour la cryptographie où le nombre de produits est conséquent.

Pour convertir l'élément d'un système de représentation adapté dans un système de numération simple de position, nous utilisons la définition du système de représentation adapté. Puisque $A(\gamma) \equiv a \pmod p$, il suffit d'évaluer le polynôme A en γ et de le réduire modulo p. Pour accélérer ce calcul, nous utiliserons la représentation d'Horner.

$$A(\gamma) = A_0 + \gamma(A_1 + \gamma(A_2 + ... + \gamma(A_{n-2} + \gamma A_{n-1}))...) \tag{3.11}$$

Algorithme 31 : Conversion hors d'un système de représentation adapté

Input : $\mathcal{B} = (p, n, \gamma, \rho)$ et $A \equiv a_{\mathcal{B}}$
Output : a tel que $0 \leq a < p$
begin
  $a \leftarrow 0$
  for $i \leftarrow n-1$ ***to*** 0 **do**
    $a \leftarrow (a\gamma + A_i) \bmod p$
  end
end

Le coût de cet algorithme est de $n-1$ multiplications modulaires.

Pour limiter le coût, nous nous proposons de précalculer une partie des opérations. Nous reprenons la forme polynomiale $A(\gamma) \equiv \sum_{i=0}^{n-1} A_i\gamma^i \pmod{p}$. Les valeurs à précalculer sont les $\gamma^i \bmod p$: nous mémorisions les n entiers g_i tel que $g_i = \gamma^i \bmod p$.

Algorithme 32 : Conversion hors d'un système de représentation adapté

Input : $\mathcal{B} = (p, n, \gamma, \rho)_E$ et $A \equiv a_{\mathcal{B}}$
Data : Pour tout i tel que $0 \leq i < n, g_i = \gamma^i \bmod p$
Output : a tel que $0 \leq a < p$
begin
  $a \leftarrow \sum_{i=0}^{n-1} A_i g_i \bmod p$
end

La complexité est cette fois ci beaucoup plus intéressante. Elle se compose de $n-1$ multiplications d'éléments $A_i < \rho$ et d'éléments $0 \leq g_i < p$. La réduction finale est encore moins coûteuse qu'une réduction modulaire sur p. On a en effet $|\sum_{i=0}^{n-1} A_i g_i| < n\rho p$. Ce nombre est bien inférieur à p^2. Globalement, la sortie avec mémorisation coûte moins cher qu'une multiplication modulaire classique. La mémorisation est de $n-1$ nombres de la taille de p.

Les conversions d'un système de représentation adapté ont un coût très faible pour ne pas dire négligeable.

3.2.4 Principe de réduction des coefficients

Il importe maintenant que le coût de l'arithmétique que nous venons de définir soit intéressant. Il dépend principalement du coût de la réduction des coefficients qui est l'opération centrale des systèmes de représentation adaptés.

Dans cette partie, nous allons décrire un algorithme général pour la réduction des coefficients.

Tous les polynômes A manipulés sont désormais d'un degré inférieur à n. Nous notons $\boldsymbol{a}$ le vecteur formé des n coefficients A_i du polynôme A. Par la suite, nous n'utilisons plus que la notation vectorielle. Elle conserve les notations des éléments du système de représentation adapté : $\boldsymbol{u} \overset{\mathcal{B}}{\equiv} \boldsymbol{v}$ est l'équivalence dans le système de représentation adapté $\mathcal{B}$. C'est à dire que les polynômes U et V correspondant sont tels que $U(\gamma) \equiv V(\gamma) \pmod{p}$. De même, $\boldsymbol{u} \equiv u_{\mathcal{B}}$ correspond à $U(\gamma) \equiv u \pmod{p}$.

Nous proposons l'algorithme `RedCoef`. Celui ci fait appel à un second algorithme `RedInt`. `RedInt` est ce que nous appelons la *réduction interne*.

Définition 16 *La réduction interne,* `RedInt` *est une réduction partielle des coefficients. L'algorithme* `RedInt` *réduit un vecteur* $\boldsymbol{x}$ *dont chaque élément est sur* k_e *bits en un vecteur* $\boldsymbol{y}$ *équivalent*

($\boldsymbol{x}\overset{\mathcal{B}}{\equiv}\boldsymbol{y}$) mais plus petit pour la norme max : les chiffres de $\boldsymbol{y}$ sont sur k_s bits avec $k_s < k_e$ (Voir Équation 3.12).

$$\mathtt{RedInt}(\boldsymbol{x}) = \boldsymbol{y} \text{ avec } \boldsymbol{x}\overset{\mathcal{B}}{\equiv}\boldsymbol{y} \text{ et } \begin{cases} \|\boldsymbol{x}\|_\infty & < & 2^{k_e} \\ \|\boldsymbol{y}\|_\infty & < & 2^{k_s} \\ k_s & < & k_e \end{cases} \tag{3.12}$$

`RedInt` est un algorithme générique qui est implantable de nombreuses façons. Les valeurs de k_e et k_s entre autres dépendent des choix d'implantation de `RedInt`. Les différentes méthodes possibles varient en fonction des contraintes : mémorisation, rapidité ...

L'algorithme `RedInt` ne réduit que partiellement les coefficients. Les algorithmes de multiplication, d'addition et de conversion du système de représentation adapté nécessitent une réduction totale des coefficients : quelle que soit la taille des chiffres d'un vecteur $\boldsymbol{x}$, `RedCoef` doit le réduire en un vecteur $\boldsymbol{y}$ équivalent, même si $\|\boldsymbol{x}\|_\infty \geq 2^{k_e}$. L'algorithme `RedInt` n'est pas suffisant. Nous proposons un algorithme `RedCoef` de réduction complète des coefficients qui utilise `RedInt` (Voir Algorithme 33).

Algorithme 33 : `RedCoef`

Input : $\mathcal{B} = (p, n, \gamma, \rho)$ et $\boldsymbol{a} \in \mathbb{Z}^n$
Data : $\mathtt{RedInt}(\boldsymbol{x}) = \boldsymbol{y}$ avec $\boldsymbol{x}\overset{\mathcal{B}}{\equiv}\boldsymbol{y}$ et $\|\boldsymbol{x}\|_\infty < 2^{k_e}, \|\boldsymbol{y}\|_\infty < 2^{k_s}$ et $k_s < k_e$
Output : s, tel que $\boldsymbol{s}\overset{\mathcal{B}}{\equiv}\boldsymbol{a}$ avec $\|\boldsymbol{s}\|_\infty < \rho$
begin
  $\boldsymbol{s} \leftarrow \boldsymbol{a}$
  while $\|\boldsymbol{s}\|_\infty \geq \rho$ **do**
    $l \leftarrow \mathtt{max}(\lfloor \log_2(\|\boldsymbol{s}\|_\infty) \rfloor + 1, k_e)$
    soient u, v tels que $\boldsymbol{s} = \boldsymbol{u}2^{l-k_e} + \boldsymbol{v}$ avec $\|\boldsymbol{v}\|_\infty < 2^{l-k_e}$
    $\boldsymbol{s} \leftarrow \mathtt{RedInt}(\boldsymbol{u})2^{l-k_e} + \boldsymbol{v}$
  end
end

L'algorithme 33 réduit complètement un vecteur $\boldsymbol{x}$. Pour cela, `RedCoef` considère le vecteur $\boldsymbol{u}$ correspondant aux k_s bits de poids fort de chaque élément de $\boldsymbol{x}$, réduit les coefficients de ce vecteur avec l'algorithme `RedInt` et recompose le vecteur $\boldsymbol{s}$ équivalent. `RedCoef` réitère ces trois étapes jusqu'à ce que chaque coefficient de $\boldsymbol{s}$ soit inférieur à ρ. Pour avoir la certitude que la boucle se termine, nous devons avoir $\rho \leq 2^{k_s}$.

Le coût de `RedCoef` dépend complètement du coût de `RedInt`. Nous pouvons voir que si les coefficients de $\boldsymbol{u}$ sont réduits de k_e bits en k_s bits alors les coefficients de $\boldsymbol{s}$ sont réduits de l bits en $l - k_e + k_s$ bits. Nous pouvons en déduire le nombre d'appels à `RedInt` pour réduire totalement $\boldsymbol{a}$ dont les éléments sont sur l_a bits en un vecteur dont les éléments sont sur k_s bits ($< \rho$). Ce nombre est donné par l'équation 3.13.

$$C_{\mathtt{RedCoef}} = \left\lceil \frac{l_a - k_s}{k_e - k_s} \right\rceil C_{\mathtt{RedInt}} \qquad \text{avec } l_a = \lfloor \log_2(\|\boldsymbol{a}\|_\infty) + 1 \rfloor \tag{3.13}$$

Dans les deux sections suivantes (Sections 3.3 et 3.4), nous proposons différentes méthodes pour l'algorithme de réduction interne `RedInt` correspondant à différents cas et différentes contraintes.

3.3 Une classe de moduli optimisée

Les protocoles cryptographiques de ECC font partie des protocoles pour lesquels nous pouvons choisir librement le modulo p sur lequel nous effectuons les opérations arithmétiques de ces protocoles. Pour les protocoles de ECC, les seules conditions sur le choix du modulo sont sa primalité et sa taille : p doit être premier et de taille 160 bits (Voir Section 1). Nous proposons un algorithme de réduction interne, `RedInt`, de très faible coût, qui fonctionne sur une classe de moduli que nous définissons.

Nous commençons par exposer l'algorithme `RedInt` que nous allons utiliser pour effectuer la réduction interne. Cet algorithme est une version vectorielle de l'algorithme de réduction par des Pseudo Mersenne (Algorithme 20). Nous faisons apparaître les besoins de l'algorithme qui définissent les systèmes de représentation adaptés exploitables par cette méthode. Cet algorithme utilise une matrice dont la valeur des coefficients agit directement sur la coût de la réduction interne par `RedInt` : *la matrice de réduction.* C'est l'existence de ce type de matrices qui définit la classe des moduli p des systèmes de représentation adaptés utilisée ainsi que la qualité de la réduction interne.

La famille des matrices de réduction permet de définir une classe de moduli associée. Dans un deuxième temps, nous présentons comment générer les systèmes de représentation adaptés qui conviennent à ces moduli. Nous donnons une méthode pour générer des systèmes de représentation adaptés sur lesquels l'arithmétique a un coût inférieur au coût sur les systèmes connus de représentation.

Nous terminons par un exemple de création d'un système de représentation adapté et du fonctionnement de son arithmétique. Cet exemple est une des propositions de système de représentation adapté pour ECC qui a pour objectif de concurrencer les différentes propositions [50, 67] de moduli tirées de la famille des Mersenne (Voir Section 2.4). La taille cryptographique concernée est de 256 bits. La totalité des autres propositions pour les différentes tailles cryptographiques est donnée dans l'annexe B.1.

3.3.1 Principe de réduction interne

L'algorithme `RedInt` que nous proposons ici se base sur l'existence d'une matrice carrée $\boldsymbol{\mathcal{M}}$ composée de n vecteurs lignes $\boldsymbol{\mathcal{M}}_i$ de taille n. Chaque vecteur $\boldsymbol{\mathcal{M}}_i$ représente l'entier $2^k\gamma^i$ dans $\mathcal{B} = (p, n, \gamma, \rho)$ où k est la taille des blocs sur lesquels nous désirons opérer : $p < 2^{kn}$.

$$\text{Pour tout } i \text{ tel que } 0 \leq i < n, \quad \boldsymbol{\mathcal{M}}_i \equiv (2^k\gamma^i)_{\mathcal{B}} \tag{3.14}$$

Il existe déja un vecteur représentant $2^k\gamma^i$ dans $\mathcal{B} = (p, n, \gamma, \rho)$. La matrice de $2^k Id$ est composée de n vecteurs lignes. Chacun de ces vecteurs lignes représente $2^k\gamma^i$ dans $\mathcal{B} = (p, n, \gamma, \rho)$. Nous avons donc une équivalence entre deux représentations dans $\mathcal{B}$: pour tout i, $\boldsymbol{\mathcal{M}}_i \overset{\mathcal{B}}{\equiv} (0, \ldots, 0, 2^k, 0, \ldots, 0)$.

Du point de vue matriciel, l'équation 3.14 se retranscrit en :

$$\boldsymbol{\mathcal{M}} \overset{\mathcal{B}}{\equiv} \begin{pmatrix} 2^k & 0 & \cdots & 0 & 0 \\ 0 & 2^k & \cdots & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 2^k & 0 \\ 0 & 0 & \cdots & 0 & 2^k \end{pmatrix} \tag{3.15}$$

Pour réduire, un vecteur $\boldsymbol{v}$ dont les éléments sont supérieur à 2^k, nous lui retirons le vecteur $(0, \ldots, 0, 2^k, 0, \ldots, 0)$ et nous lui ajoutons le vecteur $\boldsymbol{\mathcal{M}}_i$ correspondant. Par exemple, si $(1, 2, 1)$

représente la même valeur que $(0,8,0)$ dans un système de représentation $\mathcal{B}$ alors nous pouvons réduire $(5,10,3)$ en un vecteur équivalent de norme infinie inférieure $(5,10,3)\overset{\mathcal{B}}{\equiv}(5,10,3)-(0,8,0)+(1,2,1)=(6,4,4)$.

Pour que l'on puisse réduire par cette méthode, il faut que les vecteurs $\boldsymbol{\mathcal{M}}_i$ ne fassent pas trop augmenter la valeurs des éléments d'un vecteur lors d'une addition vectorielle. Pour évaluer cette augmentation, nous définissons un premier critère de qualité d'une matrice de réduction : le *poids*.

Définition 17 *Le poids ω d'une matrice $\boldsymbol{\mathcal{M}}$ est le maximum des sommes des valeurs absolues des éléments de ces colonnes. Nous pouvons le définir par l'équation 3.16.*

$$\omega = \max_{0\le j\le n-1} \sum_{i=0}^{n-1} |\boldsymbol{\mathcal{M}}_{i,j}| \tag{3.16}$$

Le poids est la norme matricielle compatible avec la norme infinie pour les vecteurs :

$$\|\boldsymbol{u}\boldsymbol{\mathcal{M}}\|_\infty \le \|\boldsymbol{u}\|_\infty \omega(\boldsymbol{\mathcal{M}})$$

Exemple 14 *La matrice $\boldsymbol{\mathcal{M}}_{256}$ suivante a pour poids $\omega = 3$.*

$$\boldsymbol{\mathcal{M}}_{256} = \begin{pmatrix} 1&0&0&2&0&0&0&0\\ 0&1&0&0&2&0&0&0\\ 0&0&1&0&0&2&0&0\\ 0&0&0&1&0&0&2&0\\ 0&0&0&0&1&0&0&2\\ 1&0&0&0&0&1&0&0\\ 0&1&0&0&0&0&1&0\\ 0&0&1&0&0&0&0&1 \end{pmatrix}$$

L'algorithme 34 réduit des vecteurs avec des chiffres sur $k_e = 2k - \lceil \log_2 \omega \rceil$ bits en un vecteur avec des chiffres sur $k_s = k+1$ bits. Ces valeurs imposent une première condition préliminaire sur la matrice de réduction $\boldsymbol{\mathcal{M}}$ (Voir Équation 3.17).

$$\left.\begin{array}{l} k_e = 2k - \lceil \log_2 \omega \rceil \\ k_s = k+1 \end{array}\right\} \text{ et } k_s < k_e \Rightarrow \lceil \log_2 \omega \rceil < k-1 \tag{3.17}$$

Algorithme 34 : `RedInt` avec Matrice

Input : $\boldsymbol{a} \in \mathbb{Z}^n$ avec $\|\boldsymbol{a}\|_\infty < 2^{k_e = 2k - \lceil \log_2 \omega \rceil}$
Data : $\mathcal{B} = (p, n, \gamma, \rho)$ et $\boldsymbol{\mathcal{M}} \overset{\mathcal{B}}{\equiv} 2^k.Id$ avec $\lceil \log_2 \omega \rceil < k-1$
Output : $\boldsymbol{s}$ tel que $\boldsymbol{s} \overset{\mathcal{B}}{\equiv} \boldsymbol{a}$ avec $\|\boldsymbol{s}\|_\infty < 2^{k_s = k+1}$
begin
  Soient $\boldsymbol{u}, \boldsymbol{v}$ tels que $\boldsymbol{a} = \boldsymbol{u}2^k + \boldsymbol{v}$
  $\boldsymbol{s} \leftarrow \boldsymbol{u}\boldsymbol{\mathcal{M}} + \boldsymbol{v}$
end

Nous vérifions que l'algorithme `RedInt` retourne un vecteur $\boldsymbol{s}$ tel que $\|\boldsymbol{s}\|_\infty < 2^{k_s}$. Nous savons que $\|\boldsymbol{v}\|_\infty < 2^k$ et $\|\boldsymbol{u}\|_\infty < 2^{k - \lceil \log_2 \omega \rceil}$. Or après la multiplication matricielle, on obtient

$$\begin{array}{rcl} \|\boldsymbol{u}\boldsymbol{\mathcal{M}}\|_\infty & \le & \|\boldsymbol{u}\|_\infty \omega \\ & < & 2^{k - \lceil \log_2 \omega \rceil} \omega \\ & < & 2^k \end{array} \tag{3.18}$$

Nous obtenons

$$\|\boldsymbol{s}\|_\infty = \|\boldsymbol{u}\boldsymbol{\mathcal{M}} + \boldsymbol{v}\|_\infty \leq \|\boldsymbol{u}\boldsymbol{\mathcal{M}}\|_\infty + \|\boldsymbol{v}\|_\infty < 2^k + 2^k = 2^{k+1}$$

Nous calculons maintenant la complexité de l'algorithme `RedInt`. Pour cela, nous définissons un second critère de qualité pour la matrice de réduction $\boldsymbol{\mathcal{M}}$: la *densité*.

Définition 18 *La densité ϕ d'une matrice $\boldsymbol{\mathcal{M}}$ est le nombre d'éléments non nuls d'une matrice. Nous pouvons la définir par l'équation 3.19.*

$$\phi = \sum_{i,j=0}^{n-1} \begin{cases} 1 \text{ si } \boldsymbol{\mathcal{M}}_{i,j} \neq 0 \\ 0 \text{ si } \boldsymbol{\mathcal{M}}_{i,j} = 0 \end{cases} \tag{3.19}$$

Exemple 15 *La matrice $\boldsymbol{\mathcal{M}}_{256}$ a pour densité* 16.

La densité est importante pour l'évaluation réelle de la multiplication du vecteur par la matrice. Si les éléments de la matrice de réduction $\boldsymbol{\mathcal{M}}$ sont de petites puissances de 2 $(1, 2, 4\dots)$, la densité est le nombre d'additions nécessaires (pas de multiplication, uniquement des décalages) pour effectuer la multiplication matricielle et l'addition vectorielle. Les multiplications par de petites puissances de 2 ne coûtent rien : ce sont de simples décalages. La qualité de la matrice de réduction est capitale pour la complexité de `RedInt`. Si elle n'est composée que de puissances de 2, nous avons comme complexité pour `RedInt` :

$$C_{\texttt{RedInt}} = \phi_{\boldsymbol{\mathcal{M}}} Add_k$$

Nous remplaçons les différentes valeurs des paramètres $k_e = 2k - \lceil \log_2 \omega \rceil$, $k_s = k + 1$ et $C_{\texttt{RedInt}} = \phi_{\boldsymbol{\mathcal{M}}} Add_k$ dans l'équation 3.13 pour calculer le coût de la réduction des coefficients d'un vecteur $\boldsymbol{a}$:

$$C_{\texttt{RedCoef}} = \left\lceil \frac{l_a - k_s}{k_e - k_s} \right\rceil C_{\texttt{RedInt}} \quad \text{avec } l_a = \lfloor \log_2(\|\boldsymbol{a}\|_\infty) + 1 \rfloor$$

$$C_{\texttt{RedCoef}} = \left\lceil \frac{l_a - k - 1}{2k - \lceil \log_2(\omega) \rceil - k - 1} \right\rceil C_{\texttt{RedInt}} \quad \text{avec } l_a = \lfloor \log_2(\|\boldsymbol{a}\|_\infty) + 1 \rfloor$$

$$C_{\texttt{RedCoef}} = \left\lceil \frac{l_a - (k+1)}{k - (\lceil \log_2(\omega) \rceil + 1)} \right\rceil C_{\texttt{RedInt}} \quad \text{avec } l_a = \lfloor \log_2(\|\boldsymbol{a}\|_\infty) + 1 \rfloor$$

$$C_{\texttt{RedCoef}} = \left\lceil \frac{l_a - (k+1)}{k - (\lceil \log_2(\omega) \rceil + 1)} \right\rceil \phi_{\boldsymbol{\mathcal{M}}} Add_k \quad \text{avec } l_a = \lfloor \log_2(\|\boldsymbol{a}\|_\infty) + 1 \rfloor$$

Le signe des éléments de la matrice $\boldsymbol{\mathcal{M}}$ est important. Nous pouvons utiliser des éléments négatifs. Si l'on n'utilise que des éléments positifs, nous avons moins de redondance (Voir Propriété 4).

Définition 19 *Un système de représentation modulaire positif est un système de représentation modulaire $\mathcal{B} = (p, n, \gamma, \rho)$ tel que pour tout entier $0 \leq a < p$ a une représentation A dont tous les éléments sont positifs.*

$$\textit{Pour tout } a \textit{ tel que } 0 \leq a < p, \quad \exists A \equiv a_{\mathcal{B}} \textit{ tel que } \forall i, A_i \geq 0$$

Propriété 4 *Soit $\mathcal{B} = (p, n, \gamma, \rho)$, si tous les éléments de la matrice $\boldsymbol{\mathcal{M}}_{\mathcal{B}}$ sont positifs alors $\mathcal{B} = (p, n, \gamma, \rho)$ est un système de représentation modulaire positif.*

$$\forall i, j \boldsymbol{\mathcal{M}}_{i,j} \geq 0 \Rightarrow \forall 0 \leq a < p, \exists A \equiv a_{\mathcal{B}} \text{ tel que } \forall i, A_i \geq 0$$

Preuve

Pour avoir une représentation de a dans le système de représentation adapté, il faut poser $\boldsymbol{a} = a$ puis réduire les coefficients du vecteur $\boldsymbol{a}$ avec `RedCoef`. `RedInt` effectue un décalage, une multiplication matricielle et une addition vectorielle. $\boldsymbol{\mathcal{M}}$ étant une matrice positive, si l'algorithme `RedInt` reçoit un vecteur positif en entrée, il retournera un vecteur positif. `RedCoef` effectue des décalages et des appels à `RedInt` ; s'il reçoit un vecteur positif en entrée, il retournera aussi un vecteur positif. Or initialement, le vecteur $\boldsymbol{a} = a$ est un vecteur positif. □

La réduction des coefficients `RedCoef` se fait efficacement si l'on utilise $\rho = 2^{k+1}$. Le système de représentation possède alors une redondance : $\pi = 1$. Il est parfois nécessaire de réduire complètement un vecteur pour obtenir sa représentation "minimale" : $\rho = 2^k$. Dans la propriété 5, nous voyons que sous certaines conditions, `RedCoef` permet de réduire un vecteur a avec $\|a\|_\infty < 2^{k+1}$ en un vecteur $< 2^k$.

Propriété 5 *Soient un système de représentation adapté $\mathcal{B} = (p, n, \gamma, \rho = 2^k)$, sa matrice de réduction $\boldsymbol{\mathcal{M}}$ de poids ω et un vecteur a avec $\|a\|_\infty < 2^{k+1}$, si $\omega \leq \frac{2^{k-1}}{n}$ alors `RedCoef` réduit $\boldsymbol{a}$ en au maximum $2n$ appels à `RedInt`.*

Preuve

Nous avons besoin de la norme L_1 d'un vecteur $\boldsymbol{u}$, notée $\|u\|_1$. Elle est la somme des valeurs absolues des coefficients de $\boldsymbol{u}$.

$$\|\boldsymbol{u}\|_1 = \sum_{i=0}^{n-1} |\boldsymbol{u}_i| \tag{3.20}$$

L'algorithme `RedCoef` reçoit en entrée un vecteur $\boldsymbol{a}$ tel que $\|\boldsymbol{a}\|_\infty < 2^{k+1}$. Ce qui correspond pour la norme L_1 à $\|\boldsymbol{a}\|_1 < n2^{k+1}$.

Chaque fois que `RedCoef` fait appel à `RedInt`, c'est que $\|\boldsymbol{a}\|_\infty \geq 2^k$. Dans ces conditions, lors du déroulement de `RedInt`, le vecteur $\boldsymbol{u}$ ne sera pas nul : $\|\boldsymbol{u}\|_1 \geq 1$.

Nous nous intéressons à $\boldsymbol{v}$ défini par $\boldsymbol{a} = 2^k\boldsymbol{u} + \boldsymbol{v}$.

$$\|\boldsymbol{v}\|_1 = \|\boldsymbol{a}\|_1 - 2^k\|\boldsymbol{u}\|_1$$

Nous évaluons maintenant $\boldsymbol{s}$ défini par $\boldsymbol{s} = \boldsymbol{u}.\boldsymbol{\mathcal{M}} + \boldsymbol{v}$.

$$\|\boldsymbol{s}\|_1 \leq \|\boldsymbol{u}.\boldsymbol{\mathcal{M}}\|_1 + \|\boldsymbol{v}\|_1$$

$$\|\boldsymbol{s}\|_1 \leq \|\boldsymbol{u}\|_1\omega + \|\boldsymbol{v}\|_1$$

$$\|\boldsymbol{s}\|_1 \leq \|\boldsymbol{u}\|_1\omega + \|\boldsymbol{a}\|_1 - 2^k|\boldsymbol{u}\|_1$$

Nous obtenons au final.

$$\|\boldsymbol{a}\|_1 - \|\boldsymbol{s}\|_1 \geq \|\boldsymbol{u}\|_1(2^k - \omega)$$

Nous avons $\|\boldsymbol{u}\|_1 \geq 1$. A chaque appel à `RedInt`, la norme L_1 du vecteur $\boldsymbol{a}$ diminue au minimum de $2^k - \omega$. Initialement, nous avions pour le vecteur $\boldsymbol{a}$:

$$\|\boldsymbol{a}\|_1 < n2^{k+1}$$

Mais après $2n$ appels à `RedInt`, nous avons

$$\|\boldsymbol{a}\|_1 < n2^{k+1} - 2n(2^k - \omega)$$

$$\|\boldsymbol{a}\|_1 < 2n\omega$$

Donc si $2n\omega \leq 2^k$, nous avons $\|\boldsymbol{a}\|_1 < 2^k$. Dans ces conditions (après $2n$ appels), nous sommes sûrs que $\|\boldsymbol{a}\|_\infty < 2^k$ puisque nous avons même $\|\boldsymbol{a}\|_1 < 2^k$. □

3.3.2 Minimisation de la matrice de réduction

La minimisation du poids et de la densité de la matrice de réduction apparaît comme étant capitale pour l'optimisation de la réduction interne `RedInt` et donc par répercussion sur la totalité de l'arithmétique sur les systèmes de représentation adaptés. Pour minimiser le poids et la densité de la matrice de réduction $\boldsymbol{\mathcal{M}}$, il s'avère que l'on peut utiliser la propriété fondamentale du système de représentation adapté : $\gamma^n \equiv c \pmod{p}$ avec c "petit".

A partir de cette propriété, nous allons chercher les systèmes de représentation adaptés particuliers qui possèdent une représentation de 2^k, notée ξ, $\xi \equiv (2^k)_{\mathcal{B}}$ avec ses coefficients ξ_i nuls ou "petits". Si l'on trouve un système de représentation adapté possédant cette particularité, nous pourrons alors définir une matrice de réduction $\boldsymbol{\mathcal{M}}$ donnant un faible coût à la réduction.

Nous disposons de n polynômes $\boldsymbol{\mathcal{M}}_i$ équivalents aux polynômes $X^i 2^k$ dans le système de représentation adapté $\mathcal{B}$. Nous connaissons le premier $\boldsymbol{\mathcal{M}}_0$ puisque $\xi \equiv (2^k)_{\mathcal{B}}$. Nous posons $\boldsymbol{\mathcal{M}}_0 = \boldsymbol{\xi}$ avec $\boldsymbol{\mathcal{M}}_0 \overset{\mathcal{B}}{\equiv} X^0 2^k$. La construction des $n-1$ autres vecteurs est donnée par :

$$\boldsymbol{\mathcal{M}}_i = X^i \xi(X) \bmod (X^n - c) \tag{3.21}$$

Les vecteurs ainsi construits vérifient

$$\boldsymbol{\mathcal{M}}_i(X) = X^i \xi(X) + K(X)(X^n - c)$$

$$\boldsymbol{\mathcal{M}}_i(\gamma) = \gamma^i \xi(\gamma) + K(\gamma)(\gamma^n - c)$$

$$\boldsymbol{\mathcal{M}}_i(\gamma) \equiv \gamma^i 2^k \pmod{p}$$

$$\boldsymbol{\mathcal{M}}_i \overset{\mathcal{B}}{\equiv} X^i 2^k$$

Il est clair que si c et $\boldsymbol{\xi}_i$ sont petits alors la matrice $\boldsymbol{\mathcal{M}}$ a des coefficients petits (faible poids). De même, si $\boldsymbol{\xi}$ a une faible densité alors $\boldsymbol{\mathcal{M}}$ a aussi une faible densité : pour un système de représentation adapté, nous avons $\phi_{\boldsymbol{\mathcal{M}}} = n\phi_{\boldsymbol{\xi}}$.

$$\boldsymbol{\mathcal{M}} = \begin{pmatrix} \xi_0 & c\xi_{n-1} & \cdots & c\xi_2 & c\xi_1 \\ \xi_1 & \xi_0 & \cdots & c\xi_3 & c\xi_2 \\ \vdots & & \ddots & & \vdots \\ \xi_{n-2} & \xi_{n-3} & \cdots & \xi_0 & c\xi_{n-1} \\ \xi_{n-1} & \xi_{n-2} & \cdots & \xi_1 & \xi_0 \end{pmatrix} \overset{\mathcal{B}}{\equiv} \begin{pmatrix} 2^k & 0 & \cdots & 0 & 0 \\ 0 & 2^k & \cdots & 0 & 0 \\ \vdots & & \cdots & & \vdots \\ 0 & 0 & \cdots & 2^k & 0 \\ 0 & 0 & \cdots & 0 & 2^k \end{pmatrix} \tag{3.22}$$

3.3.3 Création d'un système de représentation adapté optimisé

Nous pouvons maintenant construire les systèmes de représentation adaptés qui peuvent utiliser l'algorithme de réduction interne `RedInt`. Plus généralement, il nous faut définir les quatre éléments qui composent ces systèmes de représentation adaptés particuliers.

Nous désirons donc créer un système de représentation adapté possédant les bonnes caractéristiques de réduction que nous avons définies. Pour ceci, nous allons inverser le raisonnement. Nous allons partir des polynômes ξ possibles et recréer le système de représentation adapté correspondant.

Un système de représentation adapté est défini par deux polynômes :
- Le polynôme de réduction externe $E(X) = X^n - c$ qui fait partie de la définition même du système de représentation adapté. Celui-ci sert pour l'algorithmique de réduction externe.
- Le polynôme de réduction interne $I(X) = 2^k - \xi(X)$ utilisé dans l'algorithmique de réduction interne.

Nous allons voir maintenant comment construire les quatre éléments (p, n, γ, ρ) d'un système de représentation adapté à partir d'un couple E, I.

1. **Le modulo p :** Il faut commencer par rappeler que l'équation 3.22, nous permet de construire la matrice de réduction $\boldsymbol{\mathcal{M}}$ qui est équivalente à la matrice $2^k Id$. Nous obtenons une matrice $2^k Id - \boldsymbol{\mathcal{M}}$ qui est équivalente ($\overset{\mathcal{B}}{\equiv}$) à 0 (modulo p). Nous avons $2^k Id - \boldsymbol{\mathcal{M}} \overset{\mathcal{B}}{\equiv} 0$. Nous pouvons en déduire que $\det(2^k Id - \boldsymbol{\mathcal{M}}) \equiv 0 \pmod{p}$.

   Ainsi, le modulo p est un diviseur du déterminant de la matrice $2^k Id - \boldsymbol{\mathcal{M}}$. Nous pouvons choisir au choix le $\det(2^k - \boldsymbol{\mathcal{M}})$ si l'on désire travailler sur un anneau ou un de ses facteurs premiers si l'on désire travailler sur un corps premier (comme pour ECC par exemple).

2. **Le nombre de chiffres n :** Celui-ci est directement déterminé par le degré du polynôme E. Les polynômes utilisés pour la représentation appartiennent à $\mathbb{Z}[X]/(E)$. Leur degré est donc inférieur strictement au degré du polynôme E.

   Le nombre de chiffres n est donné par $n = \deg E$.

3. **La base γ :** Nous savons que par définition $E(\gamma) \equiv 0 \pmod{p}$, que $\xi \equiv (2^k)_{\mathcal{B}}$. $\xi \equiv (2^k)_{\mathcal{B}}$ se traduit par $\xi(\gamma) \equiv 2^k \pmod{p}$, c'est à dire que $I(\gamma) \equiv 0 \pmod{p}$. γ est à la fois une racine de E et de I (modulo p), le *pgcd* de E et I aura aussi γ comme racine (modulo p).

   γ est donc définie comme une racine de $pgcd(E, I) \bmod p$.

4. **Le majorant des coefficients ρ :** Le majorant des coefficients ρ est complètement défini par le polynôme $I(X) = 2^k - \xi(X)$. En fait, nous avons un choix à effectuer. La première possibilité est d'optimiser le coût en mémoire en prenant $\rho = 2^k$ comme l'algorithme nous le permet. La seconde est d'optimiser le coût en temps en prenant $\rho = 2^{k+1}$ pour limiter le nombre d'appels à `RedInt`.

Nous pouvons résumer la création d'un base optimisée par l'équation 3.23.

$$\mathcal{B} = (p, n, \gamma, \rho) \text{ avec } \begin{cases} p \mid \det(2^k - \boldsymbol{\mathcal{M}}) \\ n = \deg(E) \\ \gamma = \text{ racine } pgcd(E, I) \bmod p \\ \rho = 2^{k \text{ ou } k+1} \end{cases} \tag{3.23}$$

Finalement, nous proposons l'algorithme 35 pour trouver des systèmes de représentation adaptés pour la cryptographie.

Algorithme 35 : Création d'un système de représentation adapté

```
Output : B = (p, n, γ, ρ)
begin
    Choisir la taille des chiffres k.
    Choisir le nombre de chiffres n pour avoir la taille cryptographique nk telle que p < 2^{nk}
    while B ne convient pas do
        Choisir le polynôme unitaire E de degré n.
        Choisir ξ tel que ses coefficients ξ_i soient petits ou nuls.
        Construire M.
        p ← det(2^k Id − M)
        if p premier then
            γ ← racine pgcd(E, 2^k − ξ)
            Choisir ρ = 2^k ou ρ = 2^{k+1}
        end
    end
end
```

Grâce à cet algorithme, nous proposons pour différentes tailles cryptographiques un ensemble de systèmes de représentation aux caractéristiques intéressantes sur lesquels l'arithmétique sera compétitive par rapport aux différents moduli proposés par les standards [50, 67]. Dans l'annexe B.1, nous proposons les systèmes de représentation adaptés pour ECC. Nous donnons les éléments qui les composent ainsi que les éléments nécessaires à leur arithmétique : les polynômes de réduction interne et externe, et la matrice de réduction.

3.3.4 Densité de la classe de moduli

L'algorithme 35 permet de construire des systèmes de représentation adaptés opérant sur des moduli qui n'appartiennent pas aux classes de la “famille de nombres de Mersenne”. Cette algorithme permet aussi de créer des systèmes de représentation adaptés pour opérer sur les moduli de chacune des classes de la “famille de nombres de Mersenne” (Voir Exemples 16, 17, 18 et 19). En incluant les autres classes de moduli proposées pour l'arithmétique modulaire, cette nouvelle classe de moduli devient la plus dense d'entre elles.

Dans les exemples suivants, nous reconstruisons ces classes où seul le nombre de chiffres n est laissé libre.

Exemple 16 *Nous allons reconstruire la classe des nombres de Mersenne :* $p = 2^n - 1$.

$$\left.\begin{array}{l} k=1 \\ E=X^n-1 \\ \xi=X \end{array}\right\} \Rightarrow 2^k Id-\boldsymbol{\mathcal{M}}=\begin{pmatrix} 2 & 0 & 0 & \cdots & 0 & 0 & -1 \\ -1 & 2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \cdots & 2 & 0 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2 & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \Rightarrow p=2^n-1 \tag{3.24}$$

Exemple 17 *Nous allons reconstruire la classe des Pseudo Mersenne : $p = 2^n - c$.*

$$\left.\begin{array}{l} k=1 \\ E=X^n-c \\ \xi=X \end{array}\right\} \Rightarrow 2^k Id-\boldsymbol{\mathcal{M}}=\begin{pmatrix} 2 & 0 & 0 & \cdots & 0 & 0 & -c \\ -1 & 2 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2 & \cdots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \cdots & 2 & 0 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2 & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 2 \end{pmatrix} \Rightarrow p=2^n-c \tag{3.25}$$

Exemple 18 *Nous allons reconstruire la classe des Mersenne Généralisés : $p = P(2^k)$ avec $P = X^n - C(X)$.*

$$\left.\begin{array}{l} k \text{ défini} \\ E=X^n-C(X) \\ \xi=X \end{array}\right\} \Rightarrow 2^k Id-\boldsymbol{\mathcal{M}}=\begin{pmatrix} 2^k & 0 & 0 & \cdots & -C_2 & -C_1 & -C_0 \\ -1 & 2^k & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2^k & \cdots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \cdots & 2^k & 0 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2^k & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 2^k \end{pmatrix} \Rightarrow p=P(2^k) \tag{3.26}$$

Exemple 19 *Nous allons reconstruire la classe des Mersenne Plus Généralisés : $p = P(2^k - c)$ avec $P = X^n - C(X)$.*

$$\left.\begin{array}{l} k \text{ défini} \\ E=X^n-C(X) \\ \xi=X+c \end{array}\right\} \Rightarrow 2^k Id-\boldsymbol{\mathcal{M}}=\begin{pmatrix} 2^k-c & 0 & 0 & \cdots & -C_2 & -C_1 & -C_0 \\ -1 & 2^k-c & 0 & \cdots & 0 & 0 & 0 \\ 0 & -1 & 2^k-c & \cdots & 0 & 0 & 0 \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \cdots & 2^k-c & 0 & 0 \\ 0 & 0 & 0 & \cdots & -1 & 2^k-c & 0 \\ 0 & 0 & 0 & \cdots & 0 & -1 & 2^k-c \end{pmatrix} \Rightarrow p=P(2^k-c) \tag{3.27}$$

Ces quatre exemples permettent de reconstruire les classes de moduli de la famille des Mersenne. Ils font aussi apparaître les nombreuses autres possibilités pour choisir le couple E, I qui permet de construire les éléments de la classe de moduli concernée. C'est grâce au polynôme I et aux possibilités sur les coefficients de ξ que nous pouvons agrandir la classe des moduli intéressants bien au delà des classes classiques de moduli.

3.3.5 Exemple dans le système de représentation adapté $\mathcal{B}_{256}$

Dans cette partie, nous proposons de détailler complètement la création et l'utilisation d'un système de représentation adapté.

Création du système de représentation adapté $\mathcal{B}_{256}$

Dans cet exemple, nous désirons créer un système de représentation pour effectuer des calculs sur un corps premier $\mathbb{Z}/p\mathbb{Z}$ où p est sur 256 bits. Nous allons suivre pas à pas l'algorithme 35.

1. Il faut tout d'abord choisir la taille des chiffres. Nous prenons par exemple $k = 32$ bits, ce qui correspond aux processeurs courants.
2. Il faut ensuite calculer n tel que $nk \geq 256$. Nous pouvons prendre $n = 8$.
3. Nous choisissons le polynôme de réduction externe E. Pour avoir une bonne complexité, nous prenons $X^8 - 2$.
4. Nous effectuons plusieurs essais pour ξ. Nous commençons par les polynômes n'ayant qu'un seul coefficient à 1 : $\xi = X^i$. Les moduli construits ne nous conviennent pas : ils ne sont pas premiers. Nous essayons les polynômes ayant deux coefficients à 1 : $\xi = X^i + X^j$. Nous nous arrêtons sur le polynôme de réduction interne $\xi(X) = X^5 + 1$.
5. Nous construisons à partir de $\xi(X)$, la matrice de réduction $\boldsymbol{\mathcal{M}}_{256}$:

$$\boldsymbol{\mathcal{M}}_{256} = \begin{pmatrix} 1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Notons que son poids $\omega = 3$ et sa densité $\phi = 16$ sont très petits. Notons aussi que si l'on désire avoir p premier, alors $\phi \geq n$ (Voir Propriété 6).

Propriété 6 *Si la densité* $\phi_{\boldsymbol{\mathcal{M}}} < n$ *alors le modulo* $\det(2^k Id - \boldsymbol{\mathcal{M}})$ *n'est pas premier.*

<u>*Preuve*</u>
La matrice $\boldsymbol{\mathcal{M}}$ est une matrice carrée de n par n. Si $\phi_{\boldsymbol{\mathcal{M}}} < n$ alors il existe au moins une ligne vide (dont tout les éléments sont nuls).

$$\phi_{\boldsymbol{\mathcal{M}}} < n \Rightarrow \exists i, \forall j \boldsymbol{\mathcal{M}}_{i,j} = 0$$

La matrice $2^k Id - \boldsymbol{\mathcal{M}}$ possède donc une ligne dont tous les éléments sont nuls sauf un, 2^k. D'après les règles de calculs du déterminant nous avons 2^k qui divise le déterminant de $2^k Id - \boldsymbol{\mathcal{M}}$.

$$\phi_{\boldsymbol{\mathcal{M}}} < n \Rightarrow 2^k | \det(2^k Id - \boldsymbol{\mathcal{M}})$$

□

Notons encore que puisque la matrice $\boldsymbol{\mathcal{M}}$ est entièrement positive, la propriété 4 permet que le système de représentation adapté ne manipule que des polynômes à coefficients positifs. Il n'y aura pas de surcoût pour le signe lors de l'évaluation du rapport de redondance π.

6. Le déterminant de la matrice $2^k Id - \boldsymbol{\mathcal{M}}$ nous donne le modulo p.

$$p = \det \begin{pmatrix} 2^{32}-1 & 0 & 0 & -2 & 0 & 0 & 0 & 0 \\ 0 & 2^{32}-1 & 0 & 0 & -2 & 0 & 0 & 0 \\ 0 & 0 & 2^{32}-1 & 0 & 0 & -2 & 0 & 0 \\ 0 & 0 & 0 & 2^{32}-1 & 0 & 0 & -2 & 0 \\ 0 & 0 & 0 & 0 & 2^{32}-1 & 0 & 0 & -2 \\ -1 & 0 & 0 & 0 & 0 & 2^{32}-1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 2^{32}-1 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 2^{32}-1 \end{pmatrix}$$

$$p = 115792089021636622262124715160334756877804245386980633020041035952359812890593$$

p est premier. Sa taille est $\lfloor \log_2(p) + 1 \rfloor = 256$ bits.

7. Enfin pour déterminer γ, nous calculons le $pgcd(E, 2^k - \xi(X)) \bmod p$

$$pgcd(X^8 - 2, 2^{32} - X^5 - 1) \bmod p$$

$$= X + 101318077893932044479359125765110224563625028328059144624991208218600222490597$$

On en déduit pour γ.

$$\gamma + 101318077893932044479359125765110224563625028328059144624991208218600222490597 \equiv 0$$

$$\gamma = p - 101318077893932044479359125765110224563625028328059144624991208218600222490597$$

$$\gamma = 14474011127704577782765589395224532314179217058921488395049827733759590399996$$

8. Pour finir vue la grande qualité de la matrice de réduction (poids et densité), nous prenons $\rho = 2^{32}$ pour ainsi obtenir un bon rapport de redondance $\pi = \log_2 \frac{\rho}{\lceil p^{1/n} \rceil}$.

$$\pi = \log_2 \frac{2^{32}}{\lceil p^{1/n} \rceil} \simeq \log_2 1.0000000002328306437 \simeq 0$$

Ce système de représentation adapté n'est donc pas redondant.

Le système de représentation adapté $\mathcal{B}_{256}$ fait aussi partie d'un classe très particulière. Nous considérons que les systèmes de représentation adaptés les plus performants sont de la forme : $E = X^n - 2$ et $\xi = X^i + 1$. Cette classe possède une réduction efficace. Le coût de la réduction externe `RedExt` est de $2(n-1)$ additions et celui de la réduction interne `RedInt` de $2n$ additions. Il existe plusieurs systèmes de représentation de cette forme pour les chiffres de 32 bits. Nous donnons dans l'annexe B.2 les tailles de chiffres pour lesquelles il existe de tels systèmes.

Passage dans la représentation de $\mathcal{B}_{256}$

Nous nous proposons de représenter deux entiers dans le système de représentation adapté $\mathcal{B}_{256}$.

$$a = 111111111111111111111111111111111111111122222222222222222222222222222222222222$$

$$b = 111111111111112222222222222222222222222222333333333333333444444444444444444444444$$

Nous utilisons l'algorithme 30. Pour cela, nous précalculons les $P_i = \texttt{RedCoef}(\beta^i)$ avec $\beta = 2^{32}$. (Voir Tableau 3.4).

$$\begin{aligned}
P0 &= (0,0,0,0,0,0,0,1)\\
P1 &= (0,0,1,0,0,0,0,1)\\
P2 &= (0,0,2,0,0,2,0,1)\\
P3 &= (2,0,3,0,0,6,0,1)\\
P4 &= (8,0,4,4,0,12,0,1)\\
P5 &= (20,0,5,20,0,20,8,1)\\
P6 &= (40,8,6,60,0,30,48,1)\\
P7 &= (70,56,7,140,16,42,168,1)
\end{aligned}$$

TAB. 3.4 – Précalculs nécessaires à l'entrée dans le système de représentation $\mathcal{B}_{256}$.

Nous pouvons maintenant écrire l'entier a en base $\beta = 2^{32}$.

$$a = (4121340167, 1848846868, 1378226265, 2758018487, 1888719273, 3684936490, 17783075, 2386092942)_\beta$$

Il faut maintenant sommer les vecteurs correspondants : $A \overset{\mathcal{B}}{\equiv} \sum_{i=0}^{n-1} a_i.P_i(X)$.

$$A \overset{\mathcal{B}}{\equiv} (415853798152, 245585824296, 70919481524, 726515034708,$$

$$65941442672, 307924628816, 792155607840, 18083963567)$$

Au final, il faut appliquer l'algorithme `RedCoef` : $A \leftarrow \texttt{RedInt}(\sum_{i=0}^{n-1} a_i.P_i(X))$.

$$A \leftarrow (3536937903, 772688665, 2200004808, 665562045, 1516933361, 2981950903, 1881625898, 904094417)_\mathcal{B}$$

Nous n'avons eu besoin que d'un seul appel à `RedInt` pour réduire A tel que $A \equiv a_\mathcal{B}$.
Nous ferons la même chose pour obtenir $B \equiv b_\mathcal{B}$ avec exactement le même coût.

$$B \leftarrow (3504275317, 774104747, 4000229047, 981186486, 1516933361, 1341263555, 596411246, 864793859)_\mathcal{B}$$

Multiplication dans le système de représentation $\mathcal{B}_{256}$

Nous utilisons ici l'algorithme 27 pour multiplier les deux éléments A, B de $\mathcal{B}_{256}$.

1. La première étape est d'effectuer la multiplication polynomiale de A et B.

$$S \leftarrow \texttt{MulPol}(A, B)$$

$$S \leftarrow 12394404191244640251X^{14} + 5445674237041707346X^{13} + 22456126246501185332X^{12} + 10596674125786597954X^{11} + 20754940786274399201X^{10} + 22360940992452949951X^{9} + 22106308291801217945X^{8} + 25521614349647301834X^{7} + 16326753238062758862X^{6} + 14320338225314510507X^{5} + 9221258475851931466X^{4} + 6985510932446778164X^{3} + 4913624566887026020X^{2} + 2166430599270373964X^{1} + 781855299777785203$$

2. La deuxième étape est la réduction externe avec le polynôme $X^8 - 2$

$$S \leftarrow \texttt{RedExt}(S)$$

$$S \leftarrow 25521614349647301834X^{7} + 41115561620552039364X^{6} + 25211686699397925199X^{5} + 54133510968854302130X^{4} + 28178859184019974072X^{3} + 46423506139435824422X^{2} + 46888312584176273866X^{1} + 44994471883380221093$$

3. La troisième et dernière étape est la réduction des coefficients avec la matrice $\boldsymbol{\mathcal{M}}$. Il faudra en fait exécuter deux appels directs à `RedInt` : un premier qui réduira la partie haute mais dont le résultat aura encore des chiffres sur 35 bits ; un second appel suffira pour passer en dessous de la borne des 32 bits par chiffre.

$$S \leftarrow \texttt{RedCoef}(S)$$

Premier passage :

$$S \leftarrow (17241585329, 21793393942, 19870263953, 25013036332, 26768225729, 23039455479, 39636468860, 25630946480)$$

Second passage :

$$S \leftarrow (61716154, 318557476, 2690394778, 3538199865, 998421969, 1564619012, 981763215, 4156110017)_{\mathcal{B}}$$

Sortie du système de représentation $\mathcal{B}_{256}$

Pour sortir du système de représentation $\mathcal{B}_{256}$, nous nous contentons d'évaluer le polynôme S en γ modulo p.

$$s \leftarrow S(\gamma) \bmod p$$

$$s \leftarrow 23393354378786891104295916794436707166757217337138716566226264741841707945546$$

Nous pouvons vérifier que $s = ab \bmod p$.

Comparaison avec le Mersenne Généralisé p_{256}

Pour terminer cette partie, nous allons comparer l'efficacité de l'arithmétique dans le système de représentation $\mathcal{B}_{256}$ avec l'arithmétique modulo le Mersenne Généralisé p_{256} qui est conseillé par le NIST et le SEC [50, 67].

Nous nous proposons de comparer la multiplication modulaire modulo p_{256} utilisant l'algorithme 25 avec la multiplication sur $\mathcal{B}_{256}$ utilisant l'algorithme 27.

Dans le tableau 3.5, nous faisons apparaître les éléments de comparaison des deux algorithmes en comptant le nombre de multiplications 32 bits Mul_{32} et le nombre d'additions 32 bits Add_{32}.

Modulo p_{256}	Dans $\mathcal{B}_{256}$	Gain
Multiplication Classique	Multiplication polynomiale	
$n^2 Mul_k + (2n^2 - 2n)Add_k$	$n^2 Mul_k + (2n^2 - 2(2n-1))Add_k$	$2(n-1)Add_k$
$64Mul_{32} + 112Add_{32}$	$64Mul_{32} + 98Add_{32}$	$+14Add_{32}$
Réduction modulaire	Réduction externe	
–	$2(n-1)$	–
$46Add_{32}$	$14Add_{32}$	$+32Add_{32}$
–	Réduction interne	
–	ϕAdd_k	$-2nAdd_k$
–	16	$-16Add_{32}$
$64Mul_{32} + 158Add_{32}$	$64Mul_{32} + 128Add_{32}$	$+30Add_{32}$

Tab. 3.5 – Comparaison de la multiplication sur un corps de 256 bits.

Il faut tout d'abord noter que nous n'avons pas comptabilisé dans le tableau 3.5 les réductions finales nécessaires, ni celles modulo p, $s \leftarrow S(2^{32}) \bmod p$, ni celles du système de représentation $\mathcal{B}$ correspondant au `RedInt` finaux. Ces étapes finales sont équivalentes et correspondent à des propagations de retenues et non des additions 32 bits avec 32 bits. De plus si les utilisateurs utilisent une redondance avec $\rho = 2^{k+1}$ alors cette étape s'effectue en une seule fois au pire dans un système de représentation adapté.

Une seconde remarque importante est que la différence entre ces deux multiplications vient du fait que la multiplication classique effectue une propagation de retenue (qui sont ici des chiffres 32 bits). Celui-ci correspond en fait à une réduction interne. De même, la réduction modulaire peut évidement être considérée comme une réduction externe. La seule différence serait donc dans l'ordre des actions. La réduction modulo p commence par faire une réduction interne (peu coûteuse) puis effectue la réduction externe (plus chère). A l'inverse, dans le système de représentation $\mathcal{B}$, il faut d'abord effectuer la réduction externe (la moins coûteuse) puis finir par la réduction interne (plus coûteuse).

Notons que certains systèmes de représentation adaptés peuvent avoir une réduction interne moins coûteuse que leur réduction externe. Dans ce cas, il faut changer l'ordre de la réduction externe et interne. Pour cela, il faudra réduire les coefficients du polynôme résultant de la multiplication AB en deux fois : la partie haute puis la partie basse.

$$\left.\begin{array}{r} U \leftarrow \texttt{MulPol}(A,B) \\ V \leftarrow \texttt{RedExt}(U,E) \\ S \leftarrow \texttt{RedCoef}(U,\mathcal{B}) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} UX^n + L \leftarrow \texttt{MulPol}(A,B) \\ V \leftarrow \texttt{RedCoef}(U,\mathcal{B})X^n + \texttt{RedCoef}(L,\mathcal{B}) \\ S \leftarrow \texttt{RedExt}(V,E) \end{array}\right. \quad (3.28)$$

Pour terminer cette partie, nous évaluons le coût de la réduction avec les systèmes de représentation adaptés proposés dans l'annexe B.1 et nous le comparons aux coûts de la réduction

Moduli				Critères			Coût des réductions			
$\mathcal{B}$	$\lvert p\rvert$	n	k	$\sim\pi$	ω	ϕ	`RedExt`	`RedInt`	Total	Gain
$secp_{128r1}$	128	4	32	-	-	-	13	6	19	-
$\mathcal{B}_{128}$	128	4	32	0	3	8	6	8	14	-26%
$secp_{160r1}$	160	5	32	-	-	-	~ 12	8	20	-
$\mathcal{B}_{160}$	160	5	32	0	5	10	8	10	18	-10%
$secp_{192r1}$	192	6	32	-	-	-	14	10	24	-
$\mathcal{B}_{192_a}$	192	6	32	0	9	12	10	$12+6$	28	$+17\%$
$\mathcal{B}_{192_b}$	192	6	32	0	4097	12	10	12	22	-8%
$secp_{224r1}$	224	7	32	-	-	-	17	12	29	-
$\mathcal{B}_{224}$	224	7	32	0	17	14	12	14	26	$-11,5\%$
$secp_{256r1}$	256	8	32	-	-	-	46	14	60	-
$\mathcal{B}_{256}$	256	8	32	0	3	16	14	16	30	-50%
$\mathcal{B}_{288_a}$	288	9	32	0	2	14	36	14	50	
$\mathcal{B}_{288_b}$	288	9	32	0	1025	18	16	18	34	
$\mathcal{B}_{320_a}$	320	10	32	1	2	11	36	11	47	
$\mathcal{B}_{320_b}$	320	10	32	0	33	20	18	20	38	
$\mathcal{B}_{352}$	352	11	32	0	3	22	20	22	42	
$secp_{384r1}$	384	12	32	-	-	-	62	22	84	-
$\mathcal{B}_{384_a}$	384	12	32	0	3	19	54	19	73	-13%
$\mathcal{B}_{384_b}$	384	12	32	1	3	24	22	24	46	-45%
$\mathcal{B}_{384_c}$	384	12	32	0	9	24	22	24	46	-45%
$\mathcal{B}_{416}$	416	13	32	0	3	26	24	26	50	
$\mathcal{B}_{448_a}$	448	14	32	1	9	28	26	28	54	
$\mathcal{B}_{448_b}$	448	14	32	0	33	28	26	28	54	
$\mathcal{B}_{480_a}$	480	15	32	1	3	30	28	30	58	
$\mathcal{B}_{480_b}$	480	15	32	0	2	16	56	16	78	
$\mathcal{B}_{512}$	512	16	32	0	17	32	30	32	62	

TAB. 3.6 – Comparaison des moduli pour différentes tailles cryptographiques.

avec les nombres de Mersenne Généralisés proposés dans les standards. Le tableau 3.6 liste ces systèmes de représentation avec leurs caractéristiques et leurs coûts. Ce tableau montre l'utilité de la nouvelle classe de moduli que nous proposons à travers deux points :

1. La forte densité de cette classe permet de proposer des moduli à faible coût de réduction pour des tailles cryptographiques pour lesquelles il n'existe pas de nombre de Mersenne Généralisé intéressant : dans les standards [50, 67], il n'existe pas de proposition de nombres de Mersenne généralisés pour les tailles de 288,320,352,416,448,480 et 512 bits.

2. Pour les autres tailles cryptographiques, la technique de réduction utilisée permet de proposer des moduli pour lesquels la réduction est moins coûteuse que la réduction par les nombres de Mersenne Généralisés proposés par les standards.

3. Un dernier avantage qui n'apparaît pas dans le tableau 3.6, est la possibilité de parallélisation de la réduction par matrice sur les systèmes de représentation adaptés. Cette aspect n'a pas encore été réellement étudié.

3.4 Généralisation des systèmes de représentation adaptés

Nous avons vu comment trouver des moduli pour lesquels il existe un système de représentation adapté efficace. Maintenant, nous proposons de trouver pour un modulo donné un système de représentation adapté performant.

Nous commençons par faire une description d'un outil mathématique qui est capital dans la compréhension et la généralisation des systèmes de représentation adaptés : les *réseaux euclidiens* introduit par Minkowski en 1896 [47]. *La géométrie des nombres* permet d'obtenir une approche géométrique (plus court vecteur, existence d'un point dans un volume...) de problème arithmétiques [52, 55, 21, 53, 17, 42, 46]. Nous introduisons les principales connaissances (définitions, propriétés, problématiques, algorithmes ...) dont nous avons besoin.

Dans une deuxième partie, nous montrons sous quelles conditions les systèmes de représentation adaptés généralistes offrent des opérateurs performants. Pour proposer une arithmétique généraliste sur les systèmes de représentation adaptés, il faut savoir si pour tout modulo, il existe un système de représentation adapté pour lequel la représentation n'est pas trop redondante. Pour reprendre les notations fixées, il faut que ρ_{min} soit d'une taille raisonnable. C'est à dire que le critère de redondance π doit être une petite constante.

Nous proposons une arithmétique généraliste qui reprend la totalité de l'algorithmique déjà définie dans la section 3.2. Comme dans la section 3.3, un algorithme `RedInt` permet à la totalité de l'arithmétique sur les systèmes de représentation adaptés de fonctionner. Dans un premier temps, nous utilisons des valeurs précalculées de taille variable pour pouvoir effectuer une réduction interne rapide. Nous proposons une analyse des différents compromis possibles en fonction de la taille mémoire.

Dans un second temps, nous montrons que les différents algorithmes classiques de multiplication modulaire peuvent être retranscrits aux systèmes de représentation adaptés.

3.4.1 Les réseaux euclidiens

Un *réseau* $\mathcal{L}$ est un sous groupe discret de $\mathbb{R}^n$.

Définition 20 *Un réseau $\mathcal{L}$ est l'ensemble des combinaisons linéaires entières de d vecteurs $\boldsymbol{b}_i$ indépendants de $\mathbb{R}^n$ avec $d \leq n$.*

$$\mathcal{L} = \mathbb{Z}\,\boldsymbol{b}_1 + \cdots + \mathbb{Z}\,\boldsymbol{b}_d = \{\lambda_1 \boldsymbol{b}_1 + \cdots + \lambda_d \boldsymbol{b}_d \,:\, \lambda_i \in \mathbb{Z}\} \tag{3.29}$$

d est la dimension du réseau. $\boldsymbol{\mathcal{B}} = (\boldsymbol{b}_1, \ldots, \boldsymbol{b}_d)$ est une *base* du réseau $\mathcal{L}$. Le nombre de bases d'un réseau est infini. Nous verrons par la suite que la recherche d'une “bonne” base est un des problèmes primordiaux des réseaux. Les réseaux que nous utilisons dans le cadre des systèmes de représentation adaptés sont des réseaux *totaux*, c'est à dire que $d = n$. De plus, nous n'utilisons que des vecteurs de $\mathbb{Z}^n$.

Définition 21 *Le domaine fondamental $\mathcal{H}$ d'un réseau $\mathcal{L}$ est donné par l'équation 3.30.*

$$\mathcal{H} = \{x \in \mathbb{R}^n \,:\, x = \sum_{i=1}^{n} x_i\,\boldsymbol{b}_i,\ 0 \leq x_i < 1\} \tag{3.30}$$

Exemple 20 *Nous pouvons voir sur la figure 3.3 qu'il correspond au parallélotope formé par les vecteurs $\boldsymbol{b}_i$ de la base.*

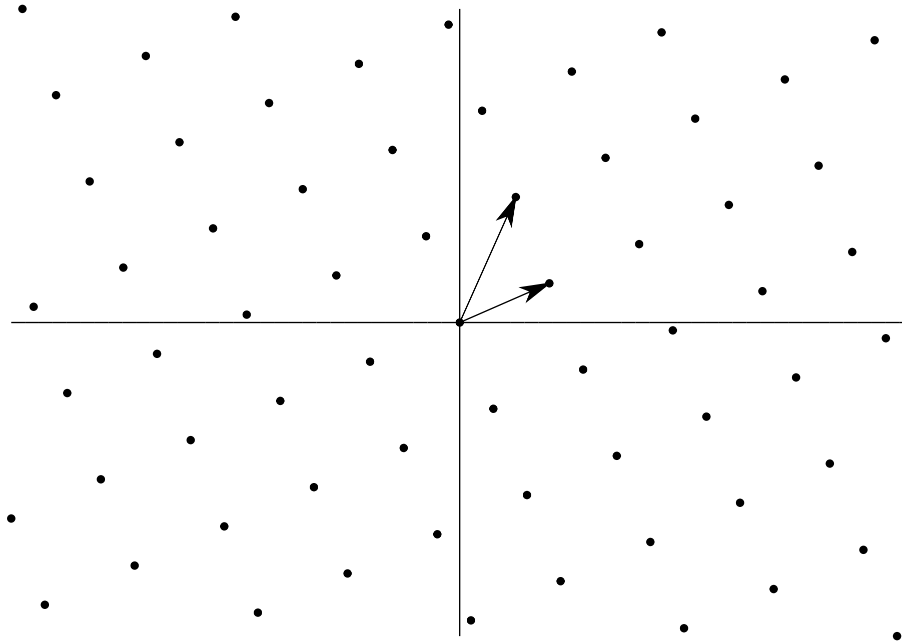

FIG. 3.2 – Le réseau $\mathcal{L}$ de $\mathbb{R}^2$ défini par les vecteurs $(8, 5)$ et $(5, 16)$.

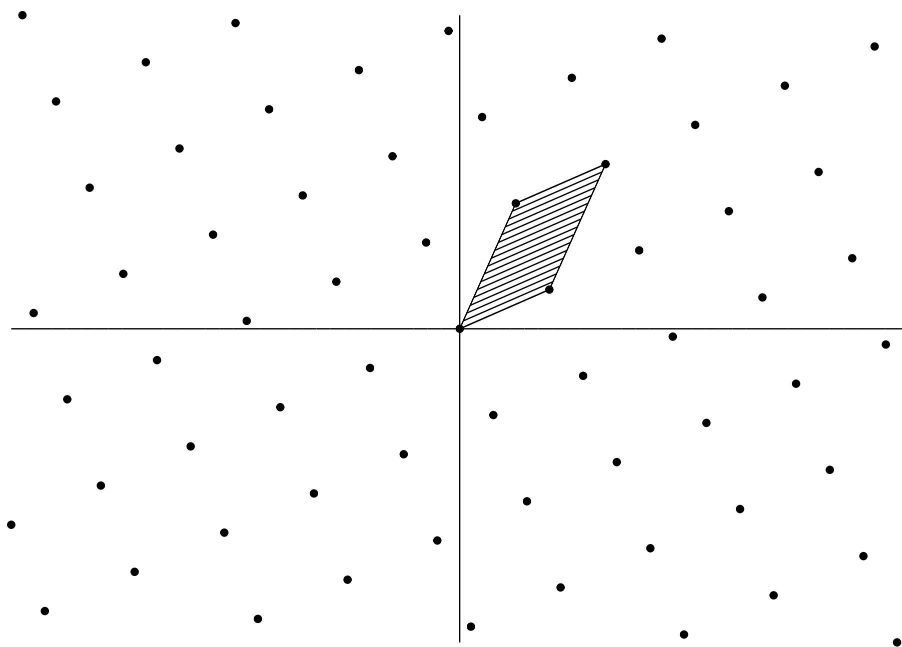

FIG. 3.3 – Le domaine fondamental du réseau $\mathcal{L}$.

Définition 22 *Le déterminant, noté* $\det(\mathcal{L})$*, d'un réseau* $\mathcal{L}$ *est le volume du domaine fondamental.*

Pour un réseau $\mathcal{L}$, le déterminant est invariant quelque soit la base $\boldsymbol{B}$ utilisée. Si le réseau est total alors le déterminant du réseau est la valeur absolue du déterminant de la matrice $\boldsymbol{B}$, formée des n vecteurs lignes $\boldsymbol{b}_i$ (Voir Équation 3.31).

$$\det(\mathcal{L}) = |\det(\boldsymbol{B})| \tag{3.31}$$

Exemple 21 *Nous calculons le déterminant* $\det(\mathcal{L})$ *du réseau* $\mathcal{L}$ *que nous avons présenté (Voir figure 3.2). Une base de ce réseau est* $\boldsymbol{B} = \begin{pmatrix} 8 & 5 \\ 5 & 16 \end{pmatrix}$.

Le déterminant de $\mathcal{L}$ *est* $\det(\mathcal{L}) = 103$ *:*

$$\det(\mathcal{L}) = \left| \det \begin{pmatrix} 8 & 5 \\ 5 & 16 \end{pmatrix} \right| = |128 - 25| = 103 \tag{3.32}$$

Les réseaux ont de nombreuses applications en cryptographie [52] : en cryptographie (NTRU [32]) comme en cryptanalyse (RSA [60]). Ces applications sont basées sur deux problèmes fondamentaux de la théorie des réseaux : SVP et CVP.

Définition 23 (SVP, "Shortest Vector Problem") *Le problème du plus court vecteur est de trouver à partir de la base* $\boldsymbol{\mathcal{B}}$ *d'un réseau* $\mathcal{L}$ *un vecteur* $\boldsymbol{u} \in \mathcal{L}$ *tel que*

$$\forall \boldsymbol{v} \in \mathcal{L}, 0 < \|u\| \leq \|v\| \tag{3.33}$$

Ce problème existe pour chaque norme. Il est très étudié pour la norme 2, c'est à dire la norme euclidienne. Mais certains problèmes nécessitent d'utiliser la norme 1 (la somme des valeurs absolues des coefficients du vecteur) ou la norme infinie aussi appelée norme max car elle correspond au max des valeurs absolues des coefficients du vecteur.

Le problème SVP a été prouvé `NP-dur` quelle que soit la norme utilisée [72]. Le problème a donc été relâché en un *problème d'approximation du plus court vecteur* (noté γSVP) qui est de trouver à partir de la base $\boldsymbol{B}$ d'un réseau $\mathcal{L}$ un vecteur $\boldsymbol{u} \in \mathcal{L}$ tel que

$$\forall \boldsymbol{v} \in \mathcal{L}, 0 < \|u\| \leq \gamma \|v\| \tag{3.34}$$

où γ est le facteur de relâchement de la difficulté de ce problème.

Définition 24 (CVP, "Closest Vector Problem") *Le problème du plus proche vecteur d'un vecteur* $w \in \mathbb{R}^n$ *est de trouver à partir de la base* $\boldsymbol{B}$ *d'un réseau* $\mathcal{L}$ *un vecteur* $\boldsymbol{u} \in \mathcal{L}$ *tel que*

$$\forall \boldsymbol{v} \in \mathcal{L}, 0 \leq \|u - w\| \leq \|v - w\| \tag{3.35}$$

Le problème CVP a été aussi prouvé `NP-dur` quelle que soit la norme utilisée [72]. Le problème a donc été une fois de plus relâché en un *problème d'approximation du plus proche vecteur* (noté γCVP) d'un vecteur w qui est de trouver à partir de la base $\boldsymbol{B}$ d'un réseau $\mathcal{L}$ un vecteur $\boldsymbol{u} \in \mathcal{L}$ tel que

$$\forall \boldsymbol{v} \in \mathcal{L}, 0 \leq \|u - w\| \leq \gamma \|v - w\| \tag{3.36}$$

où γ est le facteur de relâchement de la difficulté de ce problème.

Si le facteur γ n'est pas suffisamment grand, les problèmes γSVP et γCVP peuvent être encore difficiles, particulièrement pour la norme L_∞ qui est la norme que nous utilisons dans le cadre des systèmes de représentations adaptés.

Par la suite, nous verrons que les deux problèmes que nous devons résoudre dans le cadre des systèmes de représentation adaptés sont γSVP_∞ et γCVP_∞. En fait, Dinur [24] prouva que ces deux problèmes sont `NP-durs` pour un facteur de relâchement $\gamma = n^{1/\log\log n}$. Par contre si l'on relâche ces deux problèmes jusqu'à un facteur $\gamma = n/O(\log n)$, il a été démontré alors par Goldreich et Goldwasser [27] qu'ils ne sont plus `NP-durs`.

Pour résoudre ces deux problèmes γSVP_∞, il existe un algorithme proposé par Lenstra, Lenstra et Lovasz, [41] en 1982. C'est l'un des plus fameux algorithmes des réseaux euclidiens. On l'appelle le plus souvent LLL. LLL est un algorithme polynomial pour minimiser la base d'un réseau. Il peut être aussi utilisé pour résoudre γSVP (ainsi que γCVP). Le premier vecteur $\boldsymbol{b_1}$ de la base $\boldsymbol{B}$ retournée par l'algorithme LLL est tel que

$$\forall \boldsymbol{v} \in \mathcal{L}, \|\boldsymbol{b_1}\|_2 \leq (4/3)^{n/2} \|\boldsymbol{v}\|_2$$

LLL résout en un temps polynomial γSVP_2 avec $\gamma = (4/3)^{n/2}$. Il existe une seconde majoration du vecteur $\boldsymbol{b_1}$ obtenu, nous l'utilisons pour majorer non plus γSVP_2 mais γSVP_∞. Ce résultat est donné par l'équation 3.37.

$$\|\boldsymbol{b_1}\|_2 \leq (4/3)^{n/4} \det(\mathcal{L})^{1/n} \tag{3.37}$$

Nous pouvons utiliser cette équation pour donner une majoration de $\boldsymbol{b_1}$ mais cette fois ci pour la norme infinie. Il faut savoir que : $\forall u, \|u\|_1 \geq \|u\|_2 \geq ... \geq \|u\|_\infty$. Faute de mieux, nous pouvons utiliser cette propriété pour donner une majoration de $\boldsymbol{b_1}$ pour la norme L_∞ (Voir Équation 3.38).

$$\|\boldsymbol{b_1}\|_\infty \leq \|\boldsymbol{b_1}\|_2 \leq (4/3)^{n/4} \det(\mathcal{L})^{1/n} \tag{3.38}$$

Dans la littérature, nous trouvons de nombreuses solutions algorithmiques aux deux problèmes que nous étudions [3, 41, 59, 61, 28, 33]. L'un des premiers algorithmes de la théorie des réseaux a été proposé par L. Babai en 1986 [3] pour résoudre γCVP.

Pour terminer cette partie de rappel des réseaux euclidiens, il nous faut parler du théorème fondateur de la théorie des réseaux euclidiens qui fut proposé par Minkowski en 1896 [47]. Il majore la longueur du “plus court vecteur” (**SVP**). Ce vecteur est appelé *vecteur de Minkowski.* Ce résultat théorique très fort nous permet par la suite de majorer les éléments de nos systèmes de représentation adaptés.

Théorème 4 (Minkowski) *Soit $\mathcal{L}$ une réseau de dimension n. Alors il existe un vecteur $u \in \mathcal{L}$ tel que*

$$0 < \|u\|_\infty \leq \det(\mathcal{L})^{1/n} \tag{3.39}$$

Ce théorème possède des variantes pour chaque norme (L_1,L_2, ...). Une fois de plus nous donnons ici sa version pour L_∞ car c'est la norme que nous utilisons dans notre étude.

3.4.2 Théorème fondamental des systèmes de représentation adaptés

Pour avoir une algorithmique efficace sur les systèmes de représentation adaptés, nous devons au préalable faire l'étude de la taille des éléments que nous manipulons. La taille des coefficients est capitale dans le sens où quelle que soit la qualité des algorithmes utilisés, la taille elle même des éléments qu'ils manipulent ne doit pas être trop grande. La détermination de ρ est importante

pour l'analyse générale des systèmes de représentation adaptés. Nous rappelons que $\mathcal{B} = (p, n, \gamma, \rho)$ est un système de représentation modulaire s'il respecte l'équation 3.40. Nous rappelons aussi que ρ_{min} est défini comme le plus petit ρ respectant cette équation.

$$\forall a \in \mathbb{Z}, \exists A \in \mathbb{Z}[X] \text{ avec } \begin{cases} A(\gamma) \equiv a \pmod p \\ \deg A < n \\ \|A\|_\infty < \rho \end{cases} \tag{3.40}$$

Dans la section 3.3, nous avons proposé des systèmes de représentation adaptés avec $\rho_{min} \leq 2^k$ tel que $nk = \lfloor \log_2(p) + 1 \rfloor$. Les éléments représentés ont donc une représentation non redondante ($\pi = 0$). Nous ne pourrons pas toujours arriver à une telle qualité. Il faut fixer la redondance en bornant le critère de redondance π. Nous nous intéressons maintenant au cas général. C'est à dire aux systèmes de représentation adaptés construits pour tous les moduli qui n'appartiennent pas à la classe particulière proposée. Pour de tels systèmes de représentation, le théorème 5 fixe une majoration sur la valeur de π. Il utilise la théorie des réseaux. L'idée centrale de ce théorème est d'utiliser la représentation vectorielle des éléments de notre système de représentation adaptés. Les vecteurs représentant 0 dans un système de représentation adapté forme un réseau euclidien. Trouver la représentation de norme infinie minimale pour chaque vecteur correspond donc à résoudre CVP_∞ pour chaque vecteur. Nous voyons que la taille de toutes ces représentations minimales est bornée. Le théorème 5 permet de fixer cette borne.

Théorème 5 *Soit $E(X) = X^n - aX - b$ un polynôme irréductible dans $\mathbb{Z}[X]$, γ une racine de E dans $(\mathbb{Z}/p\mathbb{Z})[X]$ et ρ_{min} le plus petit entier tel que $\mathcal{B} = (p, n, \gamma, \rho_{min})_E$ soit un système de représentation adapté généralisé alors ρ_{min} respecte l'équation 3.41.*

$$\rho_{min} \leq (|a| + |b|)p^{1/n} \tag{3.41}$$

Dans les conditions du théorème 5, nous avons $\pi \leq 2 \lceil \log_2(|a| + |b|) \rceil$.

<u>*Preuve*</u>

Avant de commencer la preuve, nous exposons le plan de celle ci.

a) Nous prouvons dans un premier temps que tous les vecteurs représentant, dans les systèmes de représentation adaptés, un même élément de $\mathbb{Z}/p\mathbb{Z}$ sont équivalents modulo un réseau $\mathcal{L}$.

b) Nous prouvons ensuite grâce au théorème de Minkowski qu'il existe un vecteur $u \in \mathcal{L}$ tel que $\|u\|_\infty \leq p^{1/n}$. A partir de ce vecteur, nous pourrons créer un second réseau $\mathcal{L}' \subseteq \mathcal{L}$.

c) Nous prouvons ensuite que la base $\boldsymbol{B}'$ du réseau $\mathcal{L}'$ est telle que $\|\boldsymbol{B}'\|_\infty \leq (|a| + |b|)p^{1/n}$.

d) Partant du principe que le domaine fondamental d'un réseau représente la totalité des éléments de $\mathbb{R}^n$ modulo ce réseau, nous pourrons alors terminer en disant que le domaine fondamental de $\mathcal{L}'$ que nous avons majoré contient la totalité des vecteurs représentants, dans les systèmes de représentation adaptés, un élément de $\mathbb{Z}/p\mathbb{Z}$.

Nous pouvons maintenant commencer la preuve proprement dite.

a) Observons le réseau $\mathcal{L}$ composé des vecteurs qui représentent 0 dans le système de représentation adapté $\mathcal{B} = (p, n, \gamma, \rho)$. L'équation 3.42 donne une base de $\mathcal{L}$.

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 & 0 & -\gamma^{n-1} \\ 0 & 1 & 0 & \dots & 0 & 0 & -\gamma^{n-2} \\ 0 & 0 & 1 & \dots & 0 & 0 & -\gamma^{n-3} \\ \vdots & & & \ddots & & & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & -\gamma^{2} \\ 0 & 0 & 0 & \dots & 0 & 1 & -\gamma \\ 0 & 0 & 0 & \dots & 0 & 0 & p \end{pmatrix} \tag{3.42}$$

Nous avons en effet $p \equiv 0 \pmod p$. De plus, les polynômes $B_i(X) = X^i - \gamma^i$ sont tels que $B_i(\gamma) \equiv 0 \pmod p$. Toute combinaison de vecteurs $\boldsymbol{b}_i$ représentera donc aussi 0.

La dimension du réseau $\mathcal{L}$ est n : les n vecteurs sont linéairement indépendants, ce qui est visible par la structure diagonale de $\boldsymbol{B}$. Le réseau $\mathcal{L}$ est donc un réseau total.

La structure diagonale de la matrice nous permet aussi d'évaluer le déterminant de $\mathcal{L}$: $\det(\mathcal{L}) = p$.

b) Nous allons maintenant définir le vecteur de Minkowski. Nous savons d'après le théorème 4 qu'il existe un vecteur que nous noterons $\boldsymbol{m}$ tel que $0 < \|\boldsymbol{m}\|_\infty \leq \det(\mathcal{L})^{1/n}$. Nous pouvons donc en déduire l'équation 3.43.

$$\exists \boldsymbol{m} \in \mathcal{L} \text{ tel que } 0 < \|\boldsymbol{m}\|_\infty \leq p^{1/n} \tag{3.43}$$

Observons maintenant le réseau $\mathcal{L}'$ défini par la base $\boldsymbol{B'}$ composée par les n vecteurs $\boldsymbol{b}'_i$ correspondants aux coefficients des polynômes $B_i \in \mathbb{Z}[X]/(E)$ définis par l'équation 3.44.

$$B_i = X^i M \bmod (E = X^n - aX - b) \tag{3.44}$$

où M est le polynôme correspondant au vecteur $\boldsymbol{m}$.

Nous allons dans un premier temps prouver que le réseau $\mathcal{L}'$ est bien un sous-réseau de $\mathcal{L}$. Nous devons pour cela prouver que les vecteurs $\boldsymbol{b}'_i$ représentent bien 0. Cela se prouve à partir de la définition même de la base $\boldsymbol{B}'$. D'après l'équation 3.44, $B_i = X^i M \bmod E$, c'est à dire $B_i = X^i M + KE$ où K est un polynôme de degré inférieur ou égal à $n-1$ ($\deg K \leq n-1$). Or $\boldsymbol{m} \in \mathcal{L}$ donc il est tel que $M(\gamma) \equiv 0 \pmod p$. De plus, une des hypothèses de ce théorème est que nous ayons $E(\gamma) \equiv 0 \pmod p$. Nous obtenons que $\forall i, B_i \equiv 0 \pmod p$. C'est à dire que $\mathcal{L}' \subseteq \mathcal{L}$.

Dans un second temps, il nous faut prouver que $\boldsymbol{B}$ est bien une base de $\mathcal{L}'$. C'est à dire que les vecteurs $\boldsymbol{b}_i$ sont bien linéairement indépendants. Si nous supposons qu'ils ne sont pas linéairement indépendants alors il existe un vecteur non nul $\boldsymbol{z}$ tel que

$$\sum_{i=0}^{n-1} z_i \boldsymbol{b_i} = 0$$

Lorsque nous transposons ceci à la forme polynomiale, nous obtenons

$$\sum_{i=0}^{n-1} z_i B_i = 0$$

Or d'après l'équivalence de l'équation 3.44, nous avons

$$\sum_{i=0}^{n-1} z_i X^i M = 0 \pmod{E}$$

Si nous notons Z le polynôme correspondant au vecteur $\boldsymbol{z}$ ($Z = \sum i = 0^{n-1} z_i X^i$), nous obtenons au final $ZM = 0 \bmod E$. Or nous avons par définition $\deg(E) = n$. De plus, les deux polynômes Z et M ne sont pas nuls et leur degré est tel que $0 < \deg(Z), \deg(M) < n$. Nous avons donc trouver une factorisation de E. Ce qui est impossible car celui ci est par hypothèse irréductible dans $\mathbb{Z}[X]$. $\boldsymbol{B}$ est donc bien une base de $\mathcal{L}'$.

c) Nous devons maintenant étudier la taille de la base $\boldsymbol{B'}$. Nous savons tout d'abord que les coefficients du polynôme M sont tels que $\|M\|_\infty \leq p^{1/n}$ (Voir Équation 3.43). Nous nous proposons de décomposer le polynôme M avant de le majorer.

$$M = \sum_{j=0}^{n-1} m_j X^j$$

$$M \leq (\sum_{j=0}^{n-1} X^j) p^{1/n}$$

A partir de cette réécriture, nous allons pouvoir séparer les éléments de $X^i M$ en deux polynômes ; le premier sera de degré inférieur à n et le second sera composé des éléments supérieurs à X^n.

$$X^i M \leq X^i (\sum_{j=0}^{n-1} X^j) p^{1/n}$$

$$X^i M \leq (\sum_{j=i}^{n-1+i} X^j) p^{1/n}$$

$$X^i M \leq (\sum_{j=i}^{n-1} X^j + \sum_{j=n}^{n-1+i} X^j) p^{1/n}$$

$$X^i M \leq (\sum_{j=i}^{n-1} X^j + X^n (\sum_{j=0}^{i-1} X^j)) p^{1/n}$$

Nous pouvons maintenant réduire modulo le polynôme $E = X^n - aX - b$. Le premier polynôme étant inférieur à X^n, il n'est pas modifié.

$$\boldsymbol{B'}_i \leq (\sum_{j=i}^{n-1} X^j + (|a|X + |b|)(\sum_{j=0}^{i-1} X^j)) p^{1/n}$$

$$\boldsymbol{B'}_i \leq (\sum_{j=i}^{n-1} X^j + |b|(\sum_{j=0}^{i-1} X^j) + |a|(\sum_{j=1}^{i} X^j)) p^{1/n}$$

Nous pouvons multiplier le premier polynôme par b sans fausser l'inégalité.

$$|\boldsymbol{B'}_i| \leq (|b|(\sum_{j=i}^{n-1} X^j) + |b|(\sum_{j=0}^{i-1} X^j) + |a|(\sum_{j=1}^{i} X^j))p^{1/n}$$

Les deux polynômes peuvent donc maintenant être fusionnés en un seul. Nous pouvons ainsi terminer notre majoration.

$$|\boldsymbol{B'}_i| \leq (|b|(\sum_{j=0}^{n-1} X^j) + |a|(\sum_{j=1}^{i} X^j))p^{1/n}$$

$$|\boldsymbol{B'}_i| \leq (|a| + |b|)(\sum_{j=0}^{n-1} X^j)p^{1/n}$$

Au final, nous obtenons l'équation 3.45.

$$\|\boldsymbol{B'}\|_\infty \leq (|a| + |b|)p^{1/n} \tag{3.45}$$

d) Nous pouvons maintenant terminer cette preuve. Le réseau $\mathcal{L}'$ est un réseau total dont les vecteurs représentent des éléments nuls du système de représentation adaptés $\mathcal{B} = (p, n, \gamma, \rho)$. Tout vecteur de $\mathbb{Z}^n$ peut donc être ramené dans le domaine fondamental créé par la base $\boldsymbol{B'}$ du réseau $\mathcal{L}'$. Entre autre tout vecteur $(0, ..., 0, a)$ possède un vecteur équivalent modulo le réseau $\mathcal{L}'$. Tout entier a possède un vecteur équivalent $\boldsymbol{a} \in \mathcal{H}'$. Or le domaine fondamental est créé par la base $\boldsymbol{B'}$. □

Un des premiers intérêts de ce théorème est la construction du corollaire 1 qui lui a des répercussions très pratiques sur l'utilisation des systèmes de représentation adaptés.

Corollaire 1 *Soit un nombre premier p et un entier $n \geq 2$, si $pgcd(n, p-1) = 1$ alors il existe un système de représentation adapté $(p, n, \gamma = 2^{1/n} \bmod p, \rho = \lceil 2p^{1/n} \rceil)$ avec un critère de redondance de valeur $\pi = 2$.*

<u>*Preuve*</u>

Il nous faut premièrement vérifier que $\gamma = 2^{1/n} \bmod p$ existe. Puisque p est premier, il existe un générateur $g \in \mathbb{Z}/p\mathbb{Z}$ tel que $\forall x \in \mathbb{Z}/p\mathbb{Z}, \exists y$ tel que $g^y \equiv x \pmod{p}$. Il existe donc un y qui respecte l'équation 3.46.

$$g^y \equiv 2 \pmod{p} \tag{3.46}$$

De plus, nous avons par définition que $pgcd(n, p-1) = 1$. Cela veut dire qu'il existe deux entiers u, v tel que $un + v(p-1) = y$ (puisque y est divisible par le pgcd de n et $p-1$). Nous pouvons maintenant reprendre l'équation 3.46.

$$g^{un+v(p-1)} \equiv 2 \pmod{p}$$

$$(g^u)^n(g^{p-1})^v \equiv 2 \pmod{p}$$

Or d'après le Théorème 3 de Fermat, nous avons que si p est premier alors $g^{p-1} \equiv 1 \pmod{p}$. Nous pouvons donc terminer :

$$(g^u)^n \equiv 2 \pmod p$$

Il existe bien un $\gamma = 2^{1/n} \bmod p$ ($\gamma = g^u$) qui est racine du polynôme $X^n - 2 \pmod p$.

Nous devons maintenant prouver la deuxième partie du théorème : c'est à dire que $(p, n, \gamma = 2^{1/n} \bmod p, \rho = \lceil 2p^{1/n} \rceil)$ est un système de représentation modulaire. Pour ceci, nous allons utiliser le théorème 5. Grâce à ce théorème nous savons que si $X^n - c$ est irréductible dans $\mathbb{Z}[X]$ alors nous avons l'équation 3.47.

$$\rho_{min} \leq (|a| + |b|)p^{1/n} \leq \lceil 2p^{1/n} \rceil = \rho \tag{3.47}$$

Or par définition si $\rho \geq \rho_{min}$ alors (p, n, c, ρ) est un système de représentation modulaire.

Il nous faut donc prouver que $X^n - 2$ est irréductible $\forall n > 1$. Nous savons qu'un polynôme est irréductible dans $\mathbb{Z}[X]$ s'il respecte les *critères d'Eisenstein* (Voir Théorème 6 [55]).

Théorème 6 (Eisenstein) *Soit* $A \in \mathbb{Z}[X]$ *avec* $A = X^n + \sum_{i=0}^{n-1} A_i X^i$ *tel que* $\forall p$ *premier,* $p | A_0$ *ssi* $\forall i, p | A_i$ *et* $p^2 \not| A_0$ *; alors* A *est irréductible dans* $\mathbb{Z}[X]$.

D'après le théorème 6, $X^n \pm c$ (avec $n \geq 2$) est irréductible pour $c = 2, 3, 5, 7, \ldots$. $X^n - 2$ est bien irréductible et respecte bien les conditions du théorème 5. Au final, le système de représentation adapté signé $(p, n, \gamma = 2^{1/n} \bmod p, \rho = \lceil 2p^{1/n} \rceil)$ est un système de représentation modulaire. □

Le point fort du corollaire 1 est qu'il nous assure l'existence pour tout nombre premier d'un système de représentation adapté avec deux avantages très forts :

1. Le surcoût de la représentation π est au maximum de 2 bits par chiffre. Pour les n chiffres, nous avons en plus un bit pour le signe et un bit pour le facteur 2 : $\pi \leq 2$.
2. La réduction externe possède un coût minimal. Avec $c = 2$ un simple décalage et une addition suffisent.

Exemple 22 *Dans cet exemple, nous allons créer un système de représentation adapté "raisonnable" en suivant le schéma du théorème 5. Nous allons créer un système de représentation adapté et trouver le polynôme de Minkowski (le plus court vecteur représentant* 0 *dans cette base). Ce vecteur sera en effet utile pour la réduction généraliste puisque il correspond au polynôme de réduction interne.*

Nous désirons travailler avec le nombre premier

$$p = 123456789120001$$

Nous désirons utiliser un système de représentation à 4 *chiffres. Nous prendrons* $n = 4$*. Après essai, nous choisirons comme polynôme de réduction externe*

$$E = X^n + 1$$

Ce polynôme est irréductible dans $\mathbb{Z}[X]$ *et il a une solution dans* $\mathbb{Z}/p\mathbb{Z}$*. Cette solution sera donc le* γ *que nous utiliserons :*

$$\gamma = 108155120318877$$

Le théorème 5, nous donne comme résultat que tous les entiers de $\mathbb{Z}/p\mathbb{Z}$ *sont représentables dans le système de représentation adapté si nous utilisons une taille de* 12 *bits pour les chiffres.*

$$\rho = 2^{12}$$

Le coût de notre représentation est de 12 *bits pour chacun des* 4 *chiffres plus* 1 *bit de signe pour chaque chiffre. Soit au final* 52 *bits alors que le modulo p était lui même sur* 47 *bits. Évidement plus le modulo sera grand et moins le surcoût, quasiment constant, sera important.*

Nous pouvons maintenant trouver le polynôme de Minkowski. Nous utiliserons l'algorithme LLL qui permet d'approcher la solution de SVP_∞. Mais il peut être intéressant de trouver le véritable vecteur de Minkowski. Le fait que la dimension n est petite permettra de le trouver sans grande difficulté.

Commençons par définir la base $\boldsymbol{B}$ du réseau $\mathcal{L}$ des vecteurs nuls de notre système de représentation adapté.

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & -\gamma^3 \\ 0 & 1 & 0 & -\gamma^2 \\ 0 & 0 & 1 & -\gamma \\ 0 & 0 & 0 & p \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & -1265147770120216135136206031960015107592133 \\ 0 & 1 & 0 & -11697530051190760502162541129 \\ 0 & 0 & 1 & -108155120318877 \\ 0 & 0 & 0 & 123456789120001 \end{pmatrix} \tag{3.48}$$

Nous pouvons maintenant utiliser LLL pour minimiser cette base $\boldsymbol{B}$ en une seconde base $\boldsymbol{B'}$.

$$\boldsymbol{B'} = LLL(\boldsymbol{B}) = \begin{pmatrix} -3497 & -1562 & 1332 & 9 \\ -306 & -3191 & 1629 & -297 \\ 297 & -306 & -3191 & 1629 \\ -1629 & 297 & -306 & -3191 \end{pmatrix} \tag{3.49}$$

Nous nous proposons d'utiliser le premier vecteur de la base retourner par LLL comme vecteur de Minkowski. Il est à noter que même si celui-ci n'est peut être pas le plus court, il a quand même tous ces coefficients inférieurs à 2^{12}, en valeur absolue. Cette "qualité" nous suffira amplement.

$$M[X] = -3497X^3 - 1562X^2 + 1332X + 9$$

3.4.3 Utilisation de tables mémoire

Le théorème 5 est central pour les systèmes de représentation adaptés.

Premièrement, le fait que la représentation utilisée ne soit que très légèrement redondante rend possible son utilisation. Deuxièmement, nous utilisons ce théorème pour créer des algorithmes efficaces et généralistes sur les systèmes de représentation adaptés.

Dans cette partie, nous utilisons des tables mémoires pour pouvoir effectuer la réduction des coefficients (`RedCoef`) qui manque aux systèmes de représentation adaptés généralistes pour pouvoir avoir une arithmétique complète. Nous rappelons qu'il ne manque à l'algorithmique proposé dans la section 3.2 qu'un algorithme pour effectuer la réduction interne `RedInt`. Grâce à l'algorithme `RedCoef` proposé dans la section 3.2, nous savons que notre besoin correspond exactement à un algorithme qui réduit un vecteur u tel que $\|u\|_\infty < 2^{k_e}$ en un vecteur v équivalent tel que $\|v\|_\infty < 2^{k_s}$ avec $k_s < k_e$.

Nous proposons de reprendre l'idée du premier algorithme `RedInt` en séparant le polynôme U de la partie haute du polynôme L de la partie basse des coefficients.

Cette fois ci, nous mémorisons pour chaque polynôme $U2^{k_s-1}$ un polynôme équivalent V mais dont les coefficients sont plus petit que 2^{k_s-1}. Ainsi après addition, nous retournons un vecteur $\boldsymbol{s}$ avec $\|\boldsymbol{s}\|_\infty < 2^{k_s}$.

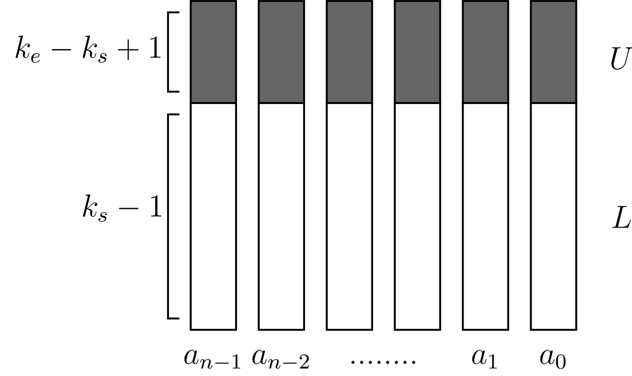

FIG. 3.4 – La décomposition des coefficients du vecteur $\boldsymbol{a}$.

Nous proposons l'algorithme 36 utilisant une table mémoire `MEM`.

Algorithme 36 : `RedInt` avec Mémorisation

Input : $\boldsymbol{a} \in \mathbb{Z}^n$ avec $\|\boldsymbol{a}\|_\infty < 2^{k_e}$
Data : $\mathcal{B} = (p, n, \gamma, \rho)$
`MEM` avec $\texttt{MEM}[u] = v$ tel que $u2^{k_s-1} \overset{\mathcal{B}}{\equiv} \boldsymbol{v}$ et $\|\boldsymbol{u}\|_\infty < 2^{k_e-k_s+1}, \|\boldsymbol{v}\|_\infty < 2^{k_s-1}$
Output : $\boldsymbol{s} \overset{\mathcal{B}}{\equiv} \boldsymbol{a}$ avec $\|\boldsymbol{s}\|_\infty < 2^{k_s}$
begin
  $\boldsymbol{a} = \boldsymbol{u}2^{k_s-1} + \boldsymbol{l}$
  $\boldsymbol{s} \leftarrow \texttt{MEM}(\boldsymbol{u}) + \boldsymbol{l}$
end

La remarque capitale pour la suite est qu'il faut impérativement que pour tout vecteur $\boldsymbol{u}$, il existe bien un vecteur $\boldsymbol{v}$ correspondant. Or ceci est possible uniquement si le système de représentation adapté $\mathcal{B}$ est tel que $2^{k_s-1} \leq \rho_{min}$. Nous voyons l'importance du théorème 5 qui donne directement la valeur minimum pour k_s qui est capitale pour la rapidité et le coût de la réduction des coefficients `RedCoef`.

$$k_s = \left\lceil \frac{1}{n} \log_2 p + \log_2 (|a| + |b|) + 1 \right\rceil \tag{3.50}$$

Or puisque nous avons $\rho = 2^{k_s}$, nous pouvons évaluer la qualité des systèmes de représentation qui utilisent l'algorithme 36.

$$\pi = 2 + 2\log_2(|a| + |b|)$$

La complexité de l'algorithme 36 est d'un appel mémoire et d'une addition vectorielle (2e ligne de l'algorithme). La première ligne de l'algorithme ne nécessite qu'un simple décalage. Cet algorithme a donc un très faible coût. Il nous faut aussi analyser la contrepartie de cette grande rapidité : la mémorisation. La table mémoire `MEM` a en entrée des vecteurs de n chiffres signés de $k_e - k_s + 1$ bits et en sortie des vecteurs de n chiffres signés de $k_s - 1$ bits. Au total, la taille #`MEM` de la table mémoire `MEM` est donnée par l'équation 3.51.

$$\#\texttt{MEM} = nk_s 2^{n(k_e-k_s+2)} \text{ bits.} \tag{3.51}$$

Cette mémorisation peut sembler trop coûteuse même si k_e peut être choisi très proche de k_s. Ceci ralentit la réduction mais n'utilise qu'une petite table mémoire `MEM`.

Dans l'idée de diminuer le nombre d'entrées de la table mémoire afin de limiter sa taille, nous utilisons la forme polynomiale des systèmes de représentation adaptés. Le vecteur $\boldsymbol{u}$ à mémoriser peut s'écrire sous la forme d'un polynôme U avec $\deg(U) < n$. Nous décomposons le polynôme U de degré $< n$ en $l = \lceil \frac{n}{d} \rceil$ polynômes U_i de degré $< d$.

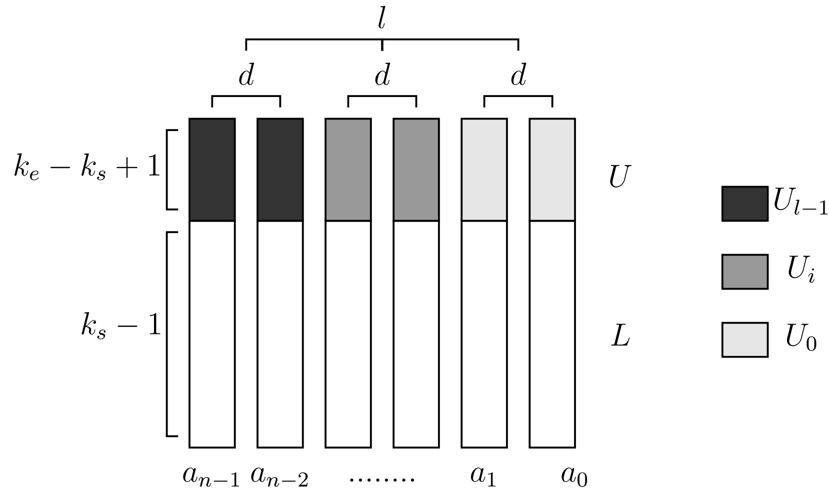

FIG. 3.5 – Une seconde décomposition des coefficients du vecteur $\boldsymbol{a}$.

La décomposition suit l'équation 3.52.

$$U = \sum_{i=0}^{l-1} U_i (X^d)^i \tag{3.52}$$

Nous déduisons de cette première équation une deuxième qui nous sert pour limiter le nombre de valeurs à mémoriser (Voir Équation 3.53).

$$\mathtt{MEM}(U) \equiv \sum_{i=0}^{l-1} \mathtt{MEM}(U_i)(X^d)^i \bmod (X^n - aX + b) \tag{3.53}$$

Grâce à cette équation, nous aurons en entrée de la table mémoire uniquement des éléments de d chiffres signés de $k_e - k_s + 1$ bits. Nous pouvons maintenant proposer l'algorithme 37.

Algorithme 37 : `RedInt` avec Faible Mémorisation

Input : A et (p, n, γ, ρ) avec $\|A\|_\infty < 2^{k_e}$

Data : `MEM` avec $\texttt{MEM}[U] = V$ tel que $U2^{k_s-1} \overset{\mathcal{B}}{\equiv} V$ et $\|U\|_2 < 2^{k_e-k_s+1}, \|V\|_2 < 2^{k_s-1}$

Output : $S \in (p, n, \gamma, \rho)$ avec $\|S\|_\infty < 2^{k_s}$ et $S \overset{\mathcal{B}}{\equiv} A$

```
begin
    A = U2^(ks-1) + L
    U = U0 + U1 X^d + U2 X^2d + ... + U2 X^(l-2)d + U2 X^(l-1)d
    S ← L
    for i ← 0 to l − 1 do
        S ← S + MEM(Ui)
        S ← S X^d mod E
    end
end
```

La taille #`MEM` de la table mémoire `MEM` est fortement diminuée (Voir Équation 3.54).

$$\#\texttt{MEM} = nk_s 2^{d(k_e-k_s+2)} \text{ bits.} \tag{3.54}$$

Évidement, la contrepartie cette fois ci du gain sur la mémorisation est l'augmentation de la complexité. Nous effectuons maintenant l appels à `MEM` avec l additions vectorielles, auxquels il faut encore ajouter une addition vectorielle pour la réduction externe.

Enfin, il faut réévaluer k_s. Sa valeur dépend maintenant du polynôme $V = \sum_{i=0}^{l-1} \texttt{MEM}(U_i)(X^d)^i \bmod E$. Celui-ci doit être tel que $\|\boldsymbol{v}\|_\infty < 2^{k_s-1}$ pour $\|\boldsymbol{s}\| < 2^{k_s}$. Après transformation, nous obtenons l'équation 3.55.

$$k_s = \left\lceil \frac{1}{n} \log_2 p + 2\log_2(|a| + |b|) + \log_2 l + 1 \right\rceil \tag{3.55}$$

Nous pouvons résumer les solutions proposées dans le tableau 3.7.

`RedInt`	avec Matrice	avec Mémorisation	avec Faible Mémorisation
π	≤ 2	$\leq 2 + \log_2(\|a\| + \|b\|)$	$\leq 2 + \log_2 l + 2\log_2(\|a\| + \|b\|)$
Add_k	$\phi(\boldsymbol{\mathcal{M}})$	n	$n(l+1)$
#`MEM`	-	$nk_s 2^{n(k_e-k_s+2)}$	$nk_s 2^{d(k_e-k_s+2)}$

Tab. 3.7 – Récapitulatifs des solutions pour `RedInt`.

3.4.4 Transcription d'algorithmes connus dans les systèmes de représentation adaptés

Les systèmes de représentation adaptés ont une forme vectorielle qui offre des propriétés de parallélisation intéressantes. Il peut être intéressant d'essayer d'adapter des algorithmes connus d'arithmétique classique sur les systèmes de représentation adaptés. Ceci permettrait de paralléliser des versions classiques de Montgomery, Barrett, ... (Voir Section 2.3).

Nous commençons par donner un plan général de l'adaptation d'un algorithme de réduction modulaire sur les systèmes de représentation adaptés. Nous donnons ensuite un exemple parmi les algorithmes de réduction modulaire.

Dans les systèmes de représentation classiques, l'opération facile est faite automatiquement : c'est la réduction interne. L'opération difficile (la réduction externe) est alors effectuée à l'aide d'un algorithme particulier : Montgomery, Barrett... Or dans les systèmes de représentation adaptés, c'est exactement l'inverse. L'opération facile est la réduction externe, la difficile est la réduction interne.

Nous allons partir de ce constat pour transcrire des algorithmes d'arithmétique modulaire dans les systèmes de représentation adaptés suivant quelques règles.

1. Une addition n'est pas changée.
2. Une multiplication sur les systèmes de représentation classiques avec la réduction interne `RedInt` intégrée est transformée en une multiplication polynomiale suivie d'une réduction externe `RedExt`.
3. Dans les systèmes de représentation classiques lors de la réduction externe, on réduit à l'aide du modulo p qui correspond au polynôme de réduction externe. Dans les systèmes de représentation adaptés, nous effectuons la réduction interne à l'aide du polynôme de réduction interne. Dans le cas généraliste, c'est le *polynôme de Minkowski.*

Exemple 23 *Nous transcrivons l'algorithme de Barrett (Voir Section 2.3) dans les systèmes de représentation adaptés. Pour cela, nous définissons la fonction $\lfloor . \rfloor$ qui transforme un polynôme A en un polynôme à coefficients entiers selon*

$$\lfloor A \rfloor = \sum_{i=0}^{\deg A} \lfloor A_i \rfloor X^i$$

***Algorithme 38** : Barrett Initial*

```
Input  : a, b, p
         with 0 ≤ a, b < p
Data   : v = ⌊β^{2k}/p⌋
Output : s = ab mod p
begin
    c ← ab
    u ← ⌊c/β^{k-1}⌋
    w ← uv
    q ← ⌊w/β^{k+1}⌋
    s ← c − qp
end
```

***Algorithme 39** : Barrett Polynomial*

```
Input  : B = (p, n, γ, ρ)
         with A, B, M ≡ a_B, b_B, 0_B
Data   : V = ⌊β^{2k} × (M^{-1} mod E)⌋
Output : S ≡ AB mod p
begin
    C ← A × B mod E
    U ← ⌊C/β^{k-1}⌋
    W ← U × V mod E
    Q ← ⌊W/β^{k+1}⌋
    S ← C − (Q × M mod E)
end
```

Nous analysons plus en détail les transformations de l'algorithme 38 à l'algorithme 39. Analysons ligne par ligne les transformations effectuées.

- *L'algorithme initial de Barrett reçoit en entrée trois nombres a, b et p tels que a et b soit entre 0 et p. Il est à noter que p est le plus petit entier différent de 0 tel que celui-ci soit équivalent à 0 modulo p. Dans la version polynomiale, l'entrée est composée de trois polynômes A, B et M tel que A et B soient inclus dans le système de représentation adapté correspondant $\mathcal{B} = (p, n, \gamma, \rho)$. Là aussi, M doit correspondre au plus petit polynôme non nul représentant 0. Ce polynôme correspond au polynôme de Minkowski.*
- *Le précalcul était initialement de $\lfloor \beta^{2k}/p \rfloor$, il devient $\lfloor \beta^{2k} \times (M^{-1} \bmod E) \rfloor$. La division par p se transforme très naturellement en une "division" par M, c'est à dire en une multiplication par l'inverse de M modulo le polynôme E. Ceci prend du temps mais est effectué en précalcul. La multiplication par β^{2k} ne pose pas de problème de compréhension.*

- *La sortie de cet algorithme n'est pas optimale. Comme l'algorithme initial proposé par Barrett, l'algorithme modifié retourne un résultat exact mais pas minimal. Dans l'algorithme de Barrett, il faut parfois réduire le résultat obtenu pour avoir la valeur comprise entre* 0 *et* p*. De même, il faudra aussi soustraire* M *pour obtenir le polynôme de coefficient minimum. Barrett doit soustraire au plus deux fois* p *pour obtenir le résultat exact* ($s < 3p$)*. Pour ce qui est de l'algorithme modifié, nous rappelons que* CVP_∞ *est un problème* `NP-dur`*. Nous nous contentons donc de conserver le polynôme obtenu même si celui-ci ne possède pas la taille minimum.*

Conclusion

Les systèmes de représentation adaptés apparaissent comme une solution intéressante aux problèmes d'arithmétique posés par la cryptographie à clé publique. De par sa définition (Section 3.1), ce système de représentation permet de tenir compte du modulo sur lequel nous désirons opérer. Il est donc plus flexible aux besoins arithmétiques du corps utilisé.

La classe de moduli présentée dans la section 3.3 répond aux deux contraintes d'une “bonne” classe de moduli nécessaire à l'implantation des protocoles d'ECC :

a) La densité de cette classe permet de présenter des moduli là où d'autres classes ne pouvaient en présenter (nombres de Mersenne, nombres de Mersenne Généralisés). En particulier dans l'annexe B.1, nous présentons des moduli intéressants pour chacune des tailles cryptographiques classiques : $160, 192, \ldots, 512$ bits.

b) L'efficacité de l'arithmétique proposée sur cette classe de moduli est souvent meilleure que celle des autres classes de moduli. La réduction modulo des Pseudo nombres de Mersenne requière plusieurs multiplications, là où l'arithmétique que nous proposons pour cette nouvelle classe n'en utilise aucune. Les réductions modulo les nombres de Mersenne Généralisés n'utilisent pas non plus de multiplication. Par contre, le nombre d'additions demandé pour réduire modulo les nombres de Mersenne proposés dans les standards cryptographiques est supérieur à celui demandé par notre système pour réduire modulo les nombres proposés en annexe (Annexe B.1).

C'est en cela que nous pouvons affirmer que cette classe de moduli répond correctement au exigence de l'utilisation des protocoles d'ECC.

Dans les travaux de la partie généraliste des systèmes de représentation adaptés, un point important est enoncé dans le théorème fondamental (Théorème 5). Grâce à ce théorème, nous savons que pour tout modulo, il existe un système de représentation adapté avec une redondance maîtrisée. Ce théorème a permis la construction des premiers algorithmes arithmétiques présentés plus haut. Mais plus généralement, il donne la condition nécessaire de l'existence d'une arithmétique généraliste basée sur ce système. C'est donc à partir de cette brique fondamentale que nous travaillons pour proposer de nouvelles approches.

La seconde partie importante du travail que nous avons développé autour de ce système de représentation concerne son utilisation dans le cadre du protocole RSA (Section 3.4). Les contraintes de ce protocole sont différentes de celles des protocoles de ECC. L'arithmétique à proposer doit être généraliste : c'est à dire que nos algorithmes doivent pouvoir fonctionner pour tout modulo. Cette contrainte très forte a évidement des répercussions sur l'efficacité des solutions proposées. Nous avons pu développer plusieurs méthodes aussi bien à travers l'utilisation de table mémoire que par la retranscription d'algorithmes classiques de réduction modulo un polynôme minimal. Ces méthodes peuvent apparaître comme très semblables à des méthodes de réduction modulaire. Mais elles ont pour avantage d'utiliser une forme vectorielle qui permet une très forte parallélisation. Ces solutions sont donc conseillées pour l'implantation matérielle du protocole RSA.

Les possibilités de parallélisation dans le système que nous proposons doivent dans tous les cas être étudiées plus précisément pour les algorithmes généralistes comme pour ceux dédiés aux moduli de la classe que nous proposons.

Chapitre 4

Arithmétique modulaire pour de petits moduli

Sommaire

4.1 Multiplication modulaire à précalculs bornés 90
4.1.1 Approximation du quotient 91
4.1.2 Évaluation de l'erreur sur le quotient 92
4.1.3 Une nouvelle multiplication modulaire avec approximation du quotient . . . 94
4.1.4 Transformation de la division précalculée en suite de multiplications 95
4.1.5 Hiérarchisation des moduli en fonction du coûts de leurs précalculs 98
4.2 Étude du changement de base en RNS 98
4.2.1 Le changement de base en RNS 99
4.2.2 Des bases RNS optimisées 100
4.2.3 Une nouvelle multiplication modulaire par un inverse 101
4.2.4 La multiplication modulaire en représentation RNS 103
4.3 Carré et multiplication modulaire via une table des inverses 106
4.3.1 La décomposition de la multiplication en inversion 106
4.3.2 L'utilisation d'une table mémoire 106

L'arithmétique modulaire est au coeur de nombreuses applications : les protocoles cryptographiques (les protocoles sur ECC ou RSA), le traitement numérique du signal. Les tailles des moduli sont parfois importantes. Dans ce cas, il est possible d'utiliser des systèmes de représentation comme les "Residue Number Systems" ou RNS basés sur le Théorème des Restes Chinois ou TRC [68]. Ces systèmes ont la particularité de transformer une opération sur de grands nombres en opération sur leurs restes modulo de petites valeurs. Des algorithmes de multiplications modulaires ont été mis au point sur ces représentations [57, 4, 6]. Une implantation de RSA été proposée dans ces systèmes [7]. Les travaux exposés dans ce chapitre sont consacrés à l'arithmétique modulaire sur de "petits" moduli ayant des tailles allant de 8 à 32 bits. Nous étudions leurs spécificités et les algorithmes qui leurs sont dédiés.

Le premier travail que nous présentons est un algorithme de multiplication modulaire pour de petits moduli. Dans ce cadre, nous trouvons deux types d'algorithmes : ceux où le modulo est une donnée non précisée et ceux dont le modulo est fixé. Lorsque le modulo est fixé, les méthodes les plus efficaces comme celles présentées section 2.3 : Montgomery [48], Barrett [14], Taylor [70], utilisent cette connaissance pour précalculer certaines valeurs pour améliorer les performances. Nous remarquons toutefois que le stockage des ces valeurs et leurs évaluations sont importants. Nous

proposons ici une alternative où les précalculs sont plus légers tout en maintenant des performances comparables. Il est ainsi possible d'envisager de changer de modulo suivant une certaine fréquence rendant ainsi le système beaucoup plus souple. Cet algorithme fonctionne sur toute taille de moduli. Il devient particulièrement compétitif lors de son utilisation sur des chiffres : $8, 16, 32$ bits. Ce travail a été publié dans [9].

Le second résultat que nous présentons dans ce chapitre est fortement lié aux systèmes RNS. En effet dans les algorithmes de multiplications modulaires en RNS, il est nécessaire de changer de base RNS (l'ensemble de moduli premiers entre-eux définissant le système). Cette opération représente le coût principal de ces arithmétiques. Nous montrons comment améliorer ce changement de base en sélectionnant des bases RNS composées de moduli voisins (ie telles que la différence maximale entre deux moduli soit le plus petite possible). Ce choix permet de transformer des produit par des inverses en divisions par de très petits nombres. Le gain obtenu est considérable, il permet de passer d'une complexité à l'origine quadratique à une complexité linéaire pour ces opérations. Ce travail a été publié dans [13, 12].

Nous terminons ce chapitre par un résultat inspiré de l'approche de Taylor [70] où les opérations modulaires d'inversion, de multiplication et d'élévation au carré sont réduites à quelques lectures de table et quelques additions. Cette méthode peut être implantée pour des moduli pouvant atteindre des tailles de 20 bits.

4.1 Multiplication modulaire à précalculs bornés

Les implantations de protocoles cryptographiques sont souvent confrontées à de lourdes contraintes matérielles : espace mémoire, taille des opérateurs, ... C'est en particulier le cas des intégrations de tels systèmes sur des cartes à puce où les approches gourmandes en mémoire ne sont pas envisageables. D'autre part, dans le cadre d'une implantation résistante aux attaques par canaux cachés, l'utilisation d'une représentation RNS offre la possibilité de changer de base lors du calcul de l'exponentiation modulaire de RSA [8]. Ces changements répétés et aléatoires troublent suffisamment le déroulement des calculs pour rendre aléatoires les *fuites* de la puce électronique où est implanté le protocole. Ces fuites : variation de la consommation électrique, dégagement de chaleur ou de magnétisme, sont analysées par les attaquants [38, 39, 2].

Nous proposons un algorithme de multiplication modulaire modulo un nombre de petite taille (par exemple 32 bits) permettant de doser les temps de précalcul en fonction du nombre de moduli pouvant intervenir. Cet algorithme se décompose en une multiplication suivie d'une réduction modulaire. La réduction modulaire est équivalente à une division euclidienne pour laquelle nous ne conservons que le reste. L'évaluation du quotient rend la division euclidienne trop coûteuse. Paul Barrett [14] propose une méthode où ce quotient est approché pour accélérer son évaluation (Voir Section 2.3). Notre méthode est identique sur ce point, la différence est due au calcul de l'approximation. Comme pour l'algorithme de Barrett, cette approximation permet d'effectuer la réduction modulaire sans division. L'algorithme de Barrett utilise comme précalcul une division. L'avantage de notre approche est que ce précalcul s'effectue sans division ni inversion mais uniquement par une suite de multiplications et d'additions. Le coût exact de ce précalcul dépend de la classe de moduli dans laquelle est le modulo sur lequel nous opérons.

Dans cette section, nous notons n le nombre de chiffres du modulo p :

$$2^{n-1} \leq p < 2^n \tag{4.1}$$

De même, nous définissons c l'entier correspondant à la forme “Pseudo Mersenne” de p :

$$c = 2^n - p \tag{4.2}$$

Nous étudions donc la réduction modulo p, ainsi défini. Pour ceci nous notons x la valeur issue d'un produit de deux nombres inférieurs à p. Nous avons $x < p^2 < 2^{2n}$.

4.1.1 Approximation du quotient

Le quotient q de la division euclidienne de x par p est $q = \left\lfloor \frac{x}{p} \right\rfloor$. Avec ce quotient, nous pouvons obtenir le résidu modulo p d'un entier x (Voir Équation 4.3).

$$x \bmod p = x - qp \text{ avec } q = \left\lfloor \frac{x}{p} \right\rfloor \tag{4.3}$$

Nous décomposons ce quotient en utilisant 2^n (Équation 4.1).

$$q = \left\lfloor \frac{x}{p} \right\rfloor = \left\lfloor \frac{x2^n}{p2^n} \right\rfloor$$

Nous utilisons l'entier c (Équation 4.2).

$$q = \left\lfloor \frac{x2^n}{p2^n} \right\rfloor = \left\lfloor \frac{x(p+c)}{p2^n} \right\rfloor$$

$$q = \left\lfloor \frac{xp + xc}{p2^n} \right\rfloor = \left\lfloor \frac{x + \frac{xc}{p}}{2^n} \right\rfloor$$

Nous décomposons $\frac{xc}{p}$ pour faire apparaître une constante ne dépendant que de p, et où est introduit un facteur 2^n pour que les parties entières soient significatives.

$$q = \left\lfloor \frac{x + \frac{xc}{p}}{2^n} \right\rfloor = \left\lfloor \frac{x + \frac{xc2^n}{p2^n}}{2^n} \right\rfloor$$

$$q = \left\lfloor \frac{x + \frac{x}{2^n}\frac{c2^n}{p}}{2^n} \right\rfloor$$

Nous posons $u = \frac{x}{2^n}$ obtenu par un simple décalage et $v = \frac{c2^n}{p}$ un facteur précalculé.

$$q = \left\lfloor \frac{x + uv}{2^n} \right\rfloor \tag{4.4}$$

Comme nous ne disposons pas des valeurs exactes de u et v, nous utilisons des approximations entières $\hat{u}$ et $\hat{v}$ qui sont inférieures ou égales à la partie entière de ces valeurs. Nous obtenons ainsi un approximation $\hat{q}$ du quotient q cherché :

$$\hat{q} = \left\lfloor \frac{x + \hat{u}\hat{v}}{2^n} \right\rfloor \tag{4.5}$$

Nous allons maintenant évaluer l'erreur commise lors de cette approximation.

4.1.2 Évaluation de l'erreur sur le quotient

Nous poserons ξ_q, ξ_u et ξ_v, les différentes erreurs d'approximation.

$$\left\{ \begin{array}{l} q = \hat{q} + \xi_q \\ u = \hat{u} + \xi_u \\ v = \hat{v} + \xi_v. \end{array} \right. \quad \text{avec} \quad \left\{ \begin{array}{l} q = \left\lfloor \frac{x+uv}{2^n} \right\rfloor \\ u = \frac{x}{2^n} \\ v = \frac{c2^n}{p} \end{array} \right. \tag{4.6}$$

Pour ξ_u, $u = \frac{x}{2^n}$ est approché par $\hat{u} = \left\lfloor \frac{x}{2^n} \right\rfloor$. Nous avons :

$$u = \hat{u} + \xi_u \text{ avec } 0 \leq \xi_u \leq \frac{2^n - 1}{2^n} < 1 \tag{4.7}$$

Ainsi, l'erreur ξ_q peut être majorée en fonction de l'erreur sur v.

Propriété 7 *Si c définie par l'équation 4.2, est tel que* $c < \frac{\sqrt{5}-1}{\sqrt{5}+1} 2^n$ *(*$\sim 0,38.2^n$*) alors* ξ_q *l'erreur sur le quotient q est majoré par :* $\xi_q \leq \lceil \xi_v \rceil$.

Preuve
Considérons l'équation 4.4.

$$q = \left\lfloor \frac{x + uv}{2^n} \right\rfloor$$

Nous exprimons successivement u et v en fonction de leurs approximations et des erreurs commises.

$$q = \left\lfloor \frac{x + (\hat{u} + \xi_u)v}{2^n} \right\rfloor = \left\lfloor \frac{x + \hat{u}v + \xi_u v}{2^n} \right\rfloor$$

$$q = \left\lfloor \frac{x + \hat{u}(\hat{v} + \xi_v) + \xi_u v}{2^n} \right\rfloor = \left\lfloor \frac{x + \hat{u}\hat{v} + \hat{u}\xi_v \xi_u v}{2^n} \right\rfloor$$

Nous isolons $\hat{q}$ définie par l'équation 4.5.

$$q = \left\lfloor \frac{x + \hat{u}\hat{v}}{2^n} + \frac{\xi_v \hat{u} + \xi_u v}{2^n} \right\rfloor$$

Pour tout a, b, nous avons $\lfloor a + b \rfloor \leq \lfloor a \rfloor + \lceil b \rceil$. Nous obtenons

$$q \leq \left\lfloor \frac{x + \hat{u}\hat{v}}{2^n} \right\rfloor + \left\lceil \frac{\xi_v \hat{u} + \xi_u v}{2^n} \right\rceil$$

$$q \leq \hat{q} + \left\lceil \frac{\xi_v \hat{u} + \xi_u v}{2^n} \right\rceil$$

Nous majorons ξ_q :

$$\xi_q \leq \left\lceil \frac{\xi_v \hat{u} + \xi_u v}{2^n} \right\rceil \tag{4.8}$$

En remarquant que $\hat{u} = \left\lfloor \frac{x}{2^n} \right\rfloor \leq \frac{x}{2^n} = u$, nous obtenons

$$\xi_q \leq \left\lceil \frac{\xi_v \hat{u} + \xi_u v}{2^n} \right\rceil \leq \left\lceil \frac{\xi_v u + \xi_u v}{2^n} \right\rceil$$

En considérant maintenant les valeurs exactes de u et de v.

$$\xi_q \leq \left\lceil \frac{\xi_v \frac{x}{2^n} + \xi_u \frac{c2^n}{p}}{2^n} \right\rceil = \left\lceil \xi_v \frac{x}{2^{2n}} + \xi_u \frac{c}{p} \right\rceil$$

Comme x est le résultat d'une multiplication de deux nombres inférieurs à p et que $\xi_u < 1$ (Voir Équation 4.7), nous obtenons

$$\xi_q \leq \left\lceil \xi_v \frac{p^2}{2^{2n}} + \frac{c}{p} \right\rceil$$

Nous décomposons cette équation en deux parties.

$$\xi_q \leq \left\lceil (\xi_v - 1) \frac{p^2}{2^{2n}} + (\frac{p^2}{2^{2n}} + \frac{c}{p}) \right\rceil$$

Nous savons par définition que $p < 2^n$. Nous obtenons l'inéquation 4.9.

$$\xi_q \leq \left\lceil (\xi_v - 1) + (\frac{p^2}{2^{2n}} + \frac{c}{p}) \right\rceil \tag{4.9}$$

Nous allons maintenant observer plus en détail $\frac{p^2}{2^{2n}} + \frac{c}{p}$ pour voir quand cette valeur reste inférieure à 1.

$$\frac{p^2}{2^{2n}} + \frac{c}{p} < 1$$

$$\frac{p^2}{2^{2n}} < 1 - \frac{c}{p}$$

$$\frac{p^2}{2^{2n}} < \frac{p-c}{p}$$

$$p^3 < (p-c)2^{2n} = (p-c)(p+c)^2 = (p-c)(p^2 + 2pc + c^2)$$

$$p^3 < p^3 + p^2c - pc^2 - c^3$$

$$0 < p^2c - pc^2 - c^3$$

Aprés simplification, nous obtenons l'inéquation 4.10 que nous pouvons résoudre avec des outils classiques.

$$0 < -c^2 - pc + p^2 \tag{4.10}$$

Nous calculons le déterminant et en déduisons les deux racines.

$$\delta = (-p)^2 - 4(-1)p^2 = 5p^2$$

$$c = \frac{p \pm \sqrt{5p^2}}{2(-1)} = p\frac{-1 \pm \sqrt{5}}{2}$$

Nous obtenons que si $c < \frac{\sqrt{5}-1}{2}p$ alors $\frac{p^2}{2^{2n}} + \frac{c}{p} < 1$.
En réutilisant l'équation 4.9, ce résultat nous donne dans ces conditions une évaluation de ξ_q

$$\xi_q \leq \lceil \xi_v \rceil$$

La majoration de c peut s'exprimer en fonction de n indépendamment de p en utilisant l'équation 4.2 $p + c = 2^n$. Nous obtenons l'équivalence des inégalités suivantes

$$c < \frac{\sqrt{5}-1}{2}p$$

$$c + \frac{\sqrt{5}-1}{2}c < \frac{\sqrt{5}-1}{2}p + \frac{\sqrt{5}-1}{2}c$$

$$c\frac{\sqrt{5}+1}{2} < \frac{\sqrt{5}-1}{2}2^n$$

$$c < \frac{\sqrt{5}-1}{\sqrt{5}+1}2^n$$

□

4.1.3 Une nouvelle multiplication modulaire avec approximation du quotient

Grâce à la propriété 7, nous proposons l'algorithme 40 de multiplication modulaire. L'approximation de v par $\hat{v}$ étant précalculée, nous considérons pour l'instant $\hat{v} = \left\lfloor \frac{c2^n}{p} \right\rfloor$. Ainsi nous aurons $\xi_v < 1$. La propriété 7 permet d'en déduire que $\xi_q \leq 1$ et que par conséquent l'approximation du quotient est de 1.

Algorithme 40 : Multiplication Modulaire avec Approximation du quotient

Input : a, b, p tels que $0 \leq a, b < p$ avec $0,619.2^n < p < 2^n$
Data : $\hat{v} = \left\lfloor \frac{c2^n}{p} \right\rfloor$
Output : s tel que $s \equiv ab \pmod p$
begin
1. $x \leftarrow ab$
2. Soient $l, \hat{u}$ tels que $x = l + 2^n \hat{u}$
3. $\hat{q} \leftarrow \left\lfloor \frac{x + \hat{u}\hat{v}}{2^n} \right\rfloor$
4. $\hat{s} \leftarrow (x - \hat{q}p \bmod 2^{n+1})$
5. **if** $\hat{s} \geq p$ **then** $s \leftarrow \hat{s} - p$ **else** $s \leftarrow \hat{s}$

end

Les trois premières lignes de l'algorithme 40 se déduisent directement des calculs de l'approximation du quotient (Voir Équation 4.5). Notons simplement que $x = l + 2^n \hat{u}$ est la division euclidienne de x par 2^n : ceci correspond à un décalage.

Pour la quatrième ligne, elle correspond à la reconstruction de $\hat{s}$. Nous savons que $\hat{s}$ est soit égal à s soit égal à $s + p$, nous avons donc $\hat{s} \leq s + p < 2p < 2^{n+1}$. Le résultat du calcul de $\hat{s}$ est sur $n + 1$ bits. Il est donc suffisant de le faire sur $n + 1$ bits : ce qui correspond au modulo 2^{n+1}.

Pour la cinquième ligne, nous avons deux cas possibles. Si nous avons $\hat{s} \geq p$ alors $\hat{s} = s + p$ et donc $s = \hat{s} - p$. Si nous avons $\hat{s} < p$ alors $\hat{s} = s$.

Avant de terminer l'analyse de cet algorithme, nous remarquons que les deux dernières lignes peuvent être simplifiées par une astuce : nous transformons le calcul de $\hat{s} \leftarrow (x - \hat{q}p \bmod 2^{n+1})$. L'idée proposée est d'effectuer le calcul de $x - \hat{q}p$ uniquement sur $n$bits. Nous savons que ce résultat peut être sur $n+1$. Pour éviter de faire une erreur, nous effectuons à la fois le calcul modulo 2^n (sur $n$bits) mais aussi modulo 3 (sur 2 bits). Ce double calcul permet de corriger l'erreur faite en effectuant le calcul modulo 2^n : si en calculant $\hat{s} \leftarrow (x - \hat{q}p \bmod 2^n)$, nous n'effectuons pas d'erreur alors $\hat{s}$ modulo 3 doit être égal à $x - \hat{q}p \bmod 3$. Si ce n'est pas le cas alors $\hat{s} \leftarrow (x - \hat{q}p \bmod 2^n) + 2^n$. L'intérêt de cette astuce est dans les simplifications du calcul de $x - \hat{q}p \bmod 2^n$:

$$\begin{aligned} x - \hat{q}p &= x - \hat{q}p \bmod 2^n \\ &= l + 2^n\hat{u} - \hat{q}p \bmod 2^n \\ &= l - \hat{q}p \bmod 2^n \\ &= l - \hat{q}(2^n - c) \bmod 2^n \\ &= l + \hat{q}c \bmod 2^n \end{aligned} \tag{4.11}$$

Ainsi, nous transformons $\hat{s} \leftarrow (x - \hat{q}p \bmod 2^{n+1})$ par $\hat{s} \leftarrow l + \hat{q}c \bmod 2^n$. Il faut ensuite tester si $\hat{s} \not\equiv x - \hat{q}p \bmod 3$. Dans ce cas, il faut alors corriger $\hat{s} \leftarrow \hat{s} + 2^n$. Notons aussi que nous sommes alors dans le cas où $\hat{s} \geq p$, nous pouvons donc directement effectuer

$$\begin{aligned} \hat{s} \leftarrow \hat{s} + 2^n - p &= \hat{s} + (p + c) - p \\ &= \hat{s} + c \end{aligned} \tag{4.12}$$

Évaluons la complexité de l'algorithme 40.

1. La première ligne correspond à une multiplication de deux chiffres de n bits.

$$Mul_n$$

2. La deuxième ligne ne coûte rien (simple décalage).
3. La troisième ligne coûte une multiplication de n sur n bits plus deux additions n bits.

$$Mul_n + 2Add_n$$

4. La quatrième ligne coûte une multiplication de n sur n bits dont on ne conserve que la partie basse correspondant qu $n+1$ bits de poids faible ; ceci peut accélérer la multiplication. Il faut aussi ajouter une addition $n+1$ bits.

$$MulBas_n + Add_{n+1}$$

5. La cinquième ligne coûte au plus une addition.

$$Add_n$$

4.1.4 Transformation de la division précalculée en suite de multiplications

Le précalcul nécessaire à l'algorithme 40 est l'approximation entière par $\hat{v}$ de $v = \frac{c2^n}{p}$. Nous décomposons son calcul.

$$v = \frac{c2^n}{p} = \frac{c2^n}{2^n - c} = c\frac{1}{1 - \frac{c}{2^n}} \tag{4.13}$$

L'équation 4.13 possède une forme que nous développons en série (Voir Équation 4.14).

$$\frac{1}{1-\frac{c}{2^n}} = 1 + \frac{c}{2^n} + \cdots + (\frac{c}{2^n})^i + \ldots \tag{4.14}$$

Nous utilisons cette série pour réécrire v.

$$\forall i, v = c + \frac{c^2}{2^n} + \frac{c^3}{2^{2n}} + \cdots + \frac{c^i}{2^{(i-1)n}} + \frac{c^{i+1}}{p2^{(i-1)n}} \tag{4.15}$$

Nous utilisons l'équation 4.15 pour proposer l'algorithme 41 permettant d'effectuer le calcul de $\hat{v}$ à moindre coût.

Algorithme 41 : Précalcul de $\hat{v}$

Input : p, c tel que $p + c = 2^n$
Output : $\hat{v}$ tel que $\hat{v} \sim \frac{c2^n}{p}$
begin
  $\hat{v} \leftarrow 0$
  $w \leftarrow c$
  while $w \neq 0$ **do**
    $\hat{v} \leftarrow \hat{v} + w$
    $w \leftarrow \lfloor \frac{wc}{2^n} \rfloor$
  end
end

Le coût de l'algorithme 41 est de $\delta - 1$ multiplications n bits sur n bits avec δ le premier entier tel que $\frac{c^\delta}{p2^{(\delta-2)n}} < 1$. Si $\frac{c^\delta}{p2^{(\delta-2)n}} < 1$, alors l'approximation entière de la série présentée dans l'équation 4.14 n'évolue plus : $\left\lfloor \frac{c^\delta}{p2^{(\delta-2)n}} \right\rfloor = 0$. Nous évaluons δ :

$$c^\delta < p2^{(\delta-2)n}$$

$$\delta \log_2 c < (\delta - 2)n + \log_2 p$$

$$(2n - \log_2 p) < \delta(n - \log_2 c)$$

$$\delta = \left\lceil \frac{2n - \log_2 p}{n - \log_2 c} \right\rceil$$

Évaluons maintenant ξ_v qui est nécessaire au bon déroulement de l'algorithme 40. Celui ci est donné par propriété 8

Propriété 8 *Soit* $\delta = \left\lceil \frac{2n - \log_2 p}{n - \log_2 c} \right\rceil \leq n$, *si* $\delta < \frac{2^n}{c}$ *alors l'algorithme 41 retourne* $\hat{v}$ *avec* $\xi_v \leq \delta$.

Preuve
Nous approchons l'équation donnant v par l'équation 4.16.

$$\hat{v} = c + \left\lfloor \frac{c^2}{2^n} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{c^2}{2^n} \right\rfloor c}{2^n} \right\rfloor + \left\lfloor \frac{\left\lfloor \frac{\left\lfloor \frac{c^2}{2^n} \right\rfloor c}{2^n} \right\rfloor c}{2^n} \right\rfloor + \cdots + \left\lfloor \frac{c^{i+1}}{p2^{(i-1)n}} \right\rfloor \tag{4.16}$$

$$\hat{v} < c + \frac{c^2}{2^n} + 1 + \frac{(\frac{c^2}{2^n} + 1)c}{2^n} + 1 + \frac{(\frac{(\frac{c^2}{2^n}+1)c}{2^n} + 1)c}{2^n} + 1 + \cdots + \frac{c^{\delta+1}}{p2^{(\delta-1)n}} + 1$$

Nous décomposons

$$\left\lfloor \frac{c^2}{2^n} \right\rfloor < \frac{c^2}{2^n} + 1$$

$$\left\lfloor \frac{\left\lfloor \frac{c^2}{2^n} \right\rfloor c}{2^n} \right\rfloor < \frac{c^3}{2^{2n}} + \frac{c^2}{2^n} + 1$$

$$\left\lfloor \frac{\left\lfloor \frac{\left\lfloor \frac{c^2}{2^n} \right\rfloor c}{2^n} \right\rfloor c}{2^n} \right\rfloor < \frac{c^4}{2^{3n}} + \frac{c^3}{2^{2n}} + \frac{c^2}{2^n} + 1$$

$$\dots$$

$$\left\lfloor \frac{\left\lfloor \frac{\left\lfloor \frac{\dots}{2^n} \right\rfloor c}{2^n} \right\rfloor c}{2^n} \right\rfloor < \frac{c^\delta}{2^{(\delta-1)n}} + \cdots + 1$$

$$\left\lfloor \frac{c^{i+1}}{p2^{(i-1)n}} \right\rfloor < \frac{c^{\delta+1}}{p2^{(\delta-1)n}} + 1$$

Nous pouvons extraire $v = c + \frac{c^2}{2^n} + \frac{c^3}{2^{2n}} + \cdots + \frac{c^{\delta+1}}{p2^{(\delta-1)n}}$:

$$\hat{v} < v + (\delta - 1) + (\delta - 2)\frac{c}{2^n} + (\delta - 3)(\frac{c}{2^n})^2 + (\delta - 4)(\frac{c}{2^n})^3 + \dots$$

$$\hat{v} < v + (\delta - 1) + (\delta - 1)\frac{c}{2^n} + (\delta - 1)(\frac{c}{2^n})^2 + (\delta - 1)(\frac{c}{2^n})^3 + \dots$$

$$\hat{v} < v + (\delta - 1)\frac{1}{1 - \frac{c}{2^n}} = v + (\delta - 1)\frac{2^n}{p}$$

$$\xi_v < \frac{(\delta - 1)2^n}{p} \tag{4.17}$$

Si l'on désire avoir $\xi_v < \delta$, nous devons avoir $\frac{(\delta-1)2^n}{p} \leq \delta$.

$$(\delta - 1)(p + c) \leq \delta p$$

$$(\delta - 1)c \leq p$$

$$\delta \leq \frac{2^n}{c}$$

□

4.1.5 Hiérarchisation des moduli en fonction du coûts de leurs précalculs

Pour terminer cette section, nous donnons la complexité effective de l'algorithme 40 que nous proposons. Nous rappelons que la complexité de l'algorithme 40 est donnée par l'équation 4.18.

$$C_{algo\ 40} = 2Mul_{n,n} + MulBas_{n,n} + 5Add_n \tag{4.18}$$

Dans le tableau 4.1, nous tenons compte du fait que nous connaissons la taille k de c : $k = \lfloor log_2(c) + 1 \rfloor$. Nous connaissons aussi la taille de $\hat{v} < 2^{k+1}$. L'équation 4.18 se transforme en l'équation 4.19.

$$C_{algo\ 40} = Mul_{n,n} + Mul_{n,k+1} + MulBas_{n,k} + 5Add_n \tag{4.19}$$

Nous utilisons cette équation pour évaluer le coût de cette méthode. Nous évaluons à nk opérations binaires une multiplication $Mul_{n,k}$ et n opérations binaires une addition Add_n.

c	ξ_q	Précalcul	Multiplication Modulaire
$k \sim \frac{n}{2}$	1	$-$	$\frac{11}{8}n^2 + 5n$
$k \sim \frac{2n}{3}$	2	$\frac{4}{9}n^2 + \frac{2}{3}n$	$\frac{17}{9}n^2 + 7n$
$k \sim \frac{3n}{4}$	3	$\frac{15}{16}n^2 + \frac{3}{2}n$	$\frac{31}{16}n^2 + 8n$
$c < 0,38.2^n$	1	$Div_{2n,n}$	$\frac{5}{2}n^2 + 5n$
$c < 0,5.2^n$	2	$Div_{2n,n}$	$\frac{5}{2}n^2 + 6n$

Tab. 4.1 – Hiérarchisation des moduli en fonction du coûts de leurs précalculs.

4.2 Étude du changement de base en RNS

Le *RNS*, pour "Residue Number Systems", est un système de représentation qui permet une parallélisation en transformant de lourds calculs sur des grands nombres en calculs sur leurs résidus modulo de petits nombres. Ce système repose sur le *théorème des restes chinois* (Voir Théorème 7).

Théorème 7 (Théorème des Restes Chinois) *Soit n entiers m_i premier deux à deux et l'entier M tel que $M = \prod_{i=0}^{n-1} m_i$. Si l'on prend n entiers x_i tel que $0 \leq x_i < m_i$ alors il existe un unique entier X avec $0 \leq X < M$ tel que $\forall i, x_i = X \bmod m_i$.*

Nous dirons que $(x_{n-1}, \ldots, x_0)$ est la représentation de X dans la base RNS $(m_{n-1}, \ldots, m_0)$.

Le grand intérêt de cette représentation est que les calculs (additions, multiplications, ...) sur X peuvent se faire sur ses résidus x_i modulo m_i. La représentation RNS de $XY \bmod M$ est $[x_{n-1}y_{n-1} \bmod m_{n-1}, \ldots, x_0y_0 \bmod m_0]$. Cette parallélisation transforme une opération sur un grand nombre en n opérations modulaires sur des petits nombres. Elle est utilisée dans de nombreux cas [71, 68]. Cette parallélisation est en particulier utilisée dans le cadre cryptographique ; les calculs de RSA sont parallélisés de cette manière [7]. Cette technique offre une alternative résistante aux attaques par canaux cachés. Ces attaques profitent des bruits naturels du matériel pour déduire des informations permettant d'attaquer des protocoles mathématiquement fiables [38, 39, 2]. La séparation des calculs ainsi que des changements de base RNS permettent de brouiller les différentes fuites émises [8].

Pour implanter le protocole RSA en représentation RNS, il faut être capable d'effectuer une multiplication modulaire en représentation RNS. Il existe plusieurs algorithmes pour effectuer cette

opération [5]. Le changement de base est central pour une majorité de multiplication modulaire en représentation RNS.

Dans cette partie, nous commençons par rappeler une des méthodes pour effectuer un changement de base. Nous montrons que l'opération la plus coûteuse de cette méthode correspond aux nombreuses multiplications modulaires par des inverses modulaires. Nous proposons un choix sur la forme des moduli utilisés pour les bases et un algorithme pour effectuer ces produits par des inverses. Ces opérations sont particulièrement efficaces lorsque le diviseur est “petit”. Ce qui est le cas dans les bases RNS que nous proposons. Enfin, nous analysons les répercutions des ces choix sur l'algorithme [5] de multiplication modulaire en représentation RNS dont le coût est de deux changements de base et cinq multiplications modulaires sur les résidus.

4.2.1 Le changement de base en RNS

Le *changement de base* revient à trouver à partir de la représentation $\boldsymbol{x} = [x_{n-1}, \ldots, x_0]$ de X dans une base RNS $\boldsymbol{m} = [m_{n-1}, \ldots, m_0]$, sa représentation $\tilde{\boldsymbol{x}} = [x_{n-1}, \ldots, x_0]$ dans une seconde base RNS $\tilde{\boldsymbol{m}} = [\tilde{m}_{n-1}, \ldots, \tilde{m}_0]$.

Nous donnons l'algorithme 42 de changement. Celui ci n'est pas dans le cas général le plus efficace mais les choix de bases que nous préconisons permettent de l'améliorer de façon conséquente.

Algorithme 42 : CDB

Input : Deux bases RNS $\boldsymbol{m}, \tilde{\boldsymbol{m}}$ et une représentation $\boldsymbol{x}$ de X dans $\boldsymbol{m}$
Output : La représentation $\tilde{\boldsymbol{x}}$ de X dans $\tilde{\boldsymbol{m}}$

```
begin
    x' ← x
    for i ← 0 to n − 1 do
        for j ← i + 1 to n − 1 do
            x'_j ← (x'_j − x'_i) m_i^{-1} mod m_j
        end
    end
    for i ← 0 to n − 1 do
        x̃_i ← x'_{n−1}
        for j ← n − 2 to 0 do
            x̃_i ← x̃_i m_j + x'_j mod m̃_i
        end
    end
end
```

L'algorithme 42 comporte deux parties distinctes.

1. La première partie est le changement de la première base RNS à une base *MRS*. Le MRS pour “Mixed Radix Systems”, est un système de représentation où un entier X a pour représentation $\boldsymbol{x} = (x'_{n-1}, \ldots, x'_0)$ dans la base MRS $(m_{n-1}, .., m_0)$ si il satisfait

$$X = x'_0 + x'_1 m_0 + x'_2 m_0 m_1 + \cdots + x'_{n-1} m_0 \ldots m_{n-2} \tag{4.20}$$

L'algorithme 42 calcule $\boldsymbol{x'}$ la représentation de X dans la base MRS comportant les mêmes éléments que la base RNS initial. Pour cela, il utilise l'équation 4.21 qui transforme $\boldsymbol{x} = X_{RNS(\boldsymbol{m})}$ en $\boldsymbol{x'} = X_{MRS(\boldsymbol{m})}$.

$$\begin{cases} x'_0 = x_0 \bmod m_0 \\ x'_1 = (x_1 - x'_0)m_0^{-1} \bmod m_1 \\ x'_2 = ((x_2 - x'_0)m_0^{-1} - x'_2)m_1^{-1} \bmod m_2 \\ \dots \\ x'_{n-1} = (\dots(x_{n-1} - x'_0)m_0^{-1} - x'_2)m_1^{-1} - \dots - x'_{n-2})m_{n-2}^{-1} \bmod m_{n-1} \end{cases} \quad (4.21)$$

Le coût de cette partie est de $\frac{(n-1)n}{2}$ soustractions modulaires et autant de multiplications par un inverse. La multiplication par une inverse $\frac{x}{d} \bmod m$ se fait habituellement en précalculant $d^{-1} \bmod m$. Pour cela, il faut utiliser les algorithmes d'inversion modulaire mieux décrits dans la section 2.2. Il faut ensuite effectuer une multiplication modulaire de x par d^{-1}. Les m_i étant des constantes, nous pouvons précalculer les $m_i^{-1} \bmod m_j$. Le coût de cette partie est de $\frac{(n-1)n}{2}$ multiplications modulaires et autant de soustractions modulaires.

2. La seconde partie est le changement de la base MRS à la seconde base RNS. Pour cela, l'algorithme 42 utilise l'équation 4.22 qui transforme $\boldsymbol{x'} = X_{MRS(\boldsymbol{m})}$ en $\tilde{\boldsymbol{x}} = X_{RNS(\tilde{\boldsymbol{m}})}$.

$$\forall i, \tilde{x}_i = x'_0 + m_0(x'_1 + m_1(x'_2 + m_2 x'_3 + \dots + m_{n-2}x'_{n-1})\dots) \bmod \tilde{m}_i \quad (4.22)$$

Le coût de cette partie est de $n(n-1)$ additions et autant de multiplications modulaires.

Le coût total d'un changement de base est de $\frac{3(n-1)n}{2}$ additions et multiplications modulaires si nous considérons une soustraction modulaire comme de coût équivalent à une addition modulaire.

$$C_{algo\ 42} = \frac{3(n-1)n}{2}(AddMod_k + MulMod_k)$$

Pour terminer cette partie, précisons que l'algorithme de multiplication modulaire en représentation RNS proposé par [5] utilise deux changements de base ainsi que $5n$ multiplications modulaires et n additions sur les résidus.

Notons aussi qu'il existe d'autres méthodes de changements de bases (par exemple ceux directement basé sur la preuve du théorème des restes chinois) sur lesquels les constantes utilisées sont difficiles à caractériser.

4.2.2 Des bases RNS optimisées

L'algorithme de multiplication modulaire en représentation RNS proposé dans [5] utilise deux changements de base. Remarquons que la majorité des opérations modulaires coûteuses de ces changements de bases sont soit des multiplications modulaires par des inverses constants ($RNS \rightarrow MRS$) soit des multiplications sur les résidus par des constantes ($MRS \rightarrow RNS'$). La solution que nous proposons pour alléger le coût de ces changements de bases est de faire que ces constantes soient le plus petites possibles. Il en existe deux : m_j modulo m_i pour la multiplication modulaire par un inverse de la première partie et m_j modulo $\tilde{m}_i$ pour la multiplication modulaire de la seconde partie.

Nous choisissons des bases $\boldsymbol{m}$ et $\tilde{\boldsymbol{m}}$ telles que la différence entre deux de leurs éléments soit le plus petite possible. Ceci revient à trouver des bases telles que t apparaissant dans les inéquations suivantes soit le plus petit possible.

$$\boldsymbol{m}, \tilde{\boldsymbol{m}} \text{ tel que } \begin{cases} \forall i \neq j, |m_i - m_j| \leq 2^t \\ \forall i \neq j, |\tilde{m}_i - \tilde{m}_j| \leq 2^t \\ \forall i, j, |m_i - \tilde{m}_j| \leq 2^t \end{cases} \quad (4.23)$$

L'intérêt de cette équation est que si t est "petit" alors les calculs du changement de base vont pouvoir se simplifier. La multiplication par m_i modulo $\tilde{m}_j$ va se transformer en une multiplication par $\tilde{d}_{i,j} = (m_i - \tilde{m}_j)$. Or l'on sait maintenant que l'entier $\tilde{d}_{i,j}$ est tel que $|\tilde{d}_{i,j}| \leq 2^t$. La multiplication modulaire initiale va devenir une multiplication modulaire par un petit entier $\tilde{d}_{i,j}$. La multiplication modulaire par un inverse devient de la même façon une multiplication modulaire par un petit inverse $d_{i,j} = (m_i - m_j)$.

Même si cette solution semble être intéressante, il faut avant tout savoir quel ordre de grandeur nous pouvons espérer pour t. Comme les éléments d'une base RNS sont premiers entre eux, et que les bases RNS de l'algorithme de multiplication modulaire sont premières entre elles, nous avons comme conditions sur ces bases :

$$\boldsymbol{m}, \tilde{\boldsymbol{m}} \text{ tels que } \begin{cases} \forall i \neq j, pgcd(m_i, mj) = 1 \\ \forall i \neq j, pgcd(\tilde{m}_i, \tilde{m}_j) = 1 \\ \forall i, j, pgcd(m_i, \tilde{m}_j) = 1 \end{cases} \tag{4.24}$$

Dans le tableau 4.2, nous évaluons t pour des tailles de résidus intéressantes pour des tailles cryptographiques classiques, $p < \prod_{i=0}^{n-1} m_i, \prod_{i=0}^{n-1} \tilde{m}_i$. Notre choix se porte sur des nombres dits Pseudo Mersenne ce qui permet d'accélérer les calculs au niveau de la réduction pour les résidus.

	128	160	192	224	256	288	320	512	1024
16	6	7	7	7	8	8	8	9	10
24	6	6	6	7	7	7	7	8	10
32	5	6	6	6	6	6	7	8	9
48	4	5	5	6	6	6	6	7	8
64	3	-	4	-	5	-	6	6	8

TAB. 4.2 – Valeur de t pour différentes tailles de chiffres et différentes tailles cryptographiques.

4.2.3 Une nouvelle multiplication modulaire par un inverse

Dans la partie précédente, est apparue la nécessité d'utiliser des bases dont les différences entre les éléments sont majorées par 2^t avec t "petit", pour accélérer la multiplication sur les résidus par une constante (ici donc une petite constante). La multiplication modulaire par un inverse ne possède pas a priori d'avantages directs à utiliser de petites constantes : lorsque x est petit, nous n'avons aucune assurance que $y = x^{-1} \bmod m$ le soit aussi. La propriété 9 nous donne même l'assurance du contraire.

Propriété 9 *Soit* $1 < x, y < m$, *si* $y = x^{-1} \bmod m$ *alors* $\log_2 y \leq \log_2 m - \log_2 x$.

<u>*Preuve*</u>

Par définition, nous avons $xx^{-1} \bmod m = 1$.

$$\exists k \in \mathbb{Z}, xy = 1 + km$$

$$\exists k \in \mathbb{Z}, km = xy - 1$$

Or $x, y, m > 1$, nous obtenons que $x, y, m \geq 2$.

$$\exists k \in \mathbb{Z}, k \geq \frac{3}{m} > 0$$

$$\exists k \in \mathbb{Z}, k \geq 1$$

Nous reprenons la première équation.

$$xy = 1 + km \geq 1 + m$$

$$y \geq \frac{m}{x}$$

$$\log_2 y \geq \log_2 m - \log_2 x$$

□

Pour effectuer cette multiplication modulaire par un inverse, l'idée que nous proposons peut être très simplement comprise par l'observation du cas de la multiplication sur les résidus par l'inverse de 2, que nous décrivons dans l'exemple suivant :

Exemple 24 *Si l'on veut calculer* $\frac{12345}{2} \bmod 56789$*, la méthode classique est de calculer* $2^{-1} \bmod 56789 = 28395$*, effectuer la multiplication* $12345 \times 28395 = 350536275$ *et réduire le résultat* $350536275 \bmod 56789 = 34567$*.*

Ce que nous proposons est de calculer $12345 \bmod 2$*. Si celui ci est égal* 1*, nous ajoutons* 56789 *à* 12345*. Si c'est* 0*, nous n'ajoutons rien. Dans notre cas,* $12345 + (1234 \bmod 2)556789 = 69134$*. Nous pouvons ainsi faire la division par 2 :* $\frac{69134}{2} = 34567$*.*

Nous généralisons l'approche de l'exemple 24 par l'équation 4.25.

$$xd^{-1} \bmod m = \frac{x + (-xm^{-1} \bmod d)m}{d} \tag{4.25}$$

où le numérateur du second membre est un multiple de d congru à $x \bmod m$.

L'algorithme 43 reprend l'équation 4.25 où pour simplifier les calculs, nous utilisons la division euclidienne de x par d, dont nous récupérons le quotient et le reste. Nous notons cette opération par $x = x_1 d + x_0$.

Algorithme 43 : Multiplication Modulaire par l'inverse d'une petite constante

Input : x, m tels que $0 < x < m$
Data : $0 < d < m$, $\hat{m} = -m^{-1} \bmod d$ et $m = m_1 d + m_0$
Output : y tel que $y = xd^{-1} \bmod m$
begin
  Soient x_0, x_1 tels que $x = x_1 d + x_0$ avec $x_0 < d$
  $q \leftarrow x_0 \hat{m} \bmod d$
  $y \leftarrow \frac{x_0 + q m_0}{d} + x_1 + q m_1$
end

Essayons d'évaluer la complexité de l'algorithme 43 qui se compose de deux divisions euclidiennes par un petit diviseur d, un produit modulo une petite valeur d et des additions de valeurs plus petites que m.

a) La division euclidienne par une petite constante $x = x_1 d + x_0$ est réalisée suivant l'algorithme 44 où nous utilisons une "petite" division euclidienne de u par d.

Algorithme 44 : Division Euclidienne par une petite constante

```
Input  : x avec x = (x_{l-1}, ..., x_0)_β
Data   : d, t tels que 0 < d < 2^t = β
Output : q, r tels que x = qd + r avec 0 ≤ r < d
begin
    r ← x_{l-1}
    for i ← l − 2 to 0 do
        u ← rβ + x_i
        Soient q_i, r tels que u = q_i d + r avec r < d
    end
    q = q_{l-2} β^{l-2} + ··· + q_1 β + q_0
end
```

L'algorithme 44 transforme la division euclidienne par une petite constante en $l-1$ "petites" divisions euclidiennes : l représente le nombre de chiffres de t bits du dividende. Ces divisions euclidiennes peuvent être effectuées au choix avec l'algorithme 40 que nous proposons dans la section précédente, avec l'algorithme de Barrett (Voir Section 2.3) ou tout autre algorithme de réduction modulaire qui conserve le quotient. Au final, nous avons

$$C_{algo\ 44} = (l-1)(Div_{2t,t} + Add_t)$$

b) La seconde ligne correspond à une multiplication modulaire classique de taille t.

c) La dernière ligne est composée de deux parties. La première est composée d'une addition sur $2t$ bits, une multiplication t par t bits et une division par une constante de t bits. Le coût de cette partie peut être considéré comme une multiplication modulaire et une addition classique. La seconde partie est composée d'une addition de lt bits et une multiplication t par $(l-1)t$ bits. Cette seconde partie peut être évaluée à une réduction, $l+2$ additions et l multiplications de mots de t bits.

Au final, l'algorithme 43 a une complexité de $(l+1)$ multiplications modulaires et $(2l+1)$ additions sur des mots de t bits avec $0 < d < 2^t < m < 2^{lt}$.

$$C_{algo\ 43} = (l+1)MulMod_t + (8l+7)Add_t$$

Si nous utilisons l'algorithme 40 pour effectuer les réductions, nous pouvons évaluer exactement la complexité de l'algorithme 43.

$$C_{algo\ 43} = (3l+3)Mul_t + (8l+7)Add_t$$

4.2.4 La multiplication modulaire en représentation RNS

Avant d'évaluer la multiplication, nous rappelons les différents choix et améliorations. Nous pouvons prendre des nombres Pseudo Mersenne comme moduli ce qui nous permet d'utiliser l'algorithme 20 qui a une complexité de l multiplications t par t bits. Ainsi, les constantes intervenant dans les multiplications modulaires deviennent inférieures à 2^t. Pour une multiplication par ces constantes, nous avons un coût de $Mul_{k,t} \sim lMul_{t,t}$ car $k \leq lt$. Pour réduire les t bits débordant

du résultat de cette multiplication, nous avons besoin d'une seule multiplication $Mul_{t,t}$ (grâce à la forme des nombres Pseudo Mersenne). Enfin, comme nous l'avons vu dans la partie précédente, la multiplication modulaire par un inverse constant nécessite $3(l+1)Mul_t$ au lieu d'une multiplication modulaire qui coûterait au minimum $2l^2 Mul_t$ ($2l^2 + l$ pour Montgomery, $2l^2 + 4l$ pour Barrett). Dans le tableau 4.3, nous résumons ces différents points.

Opération	Méthode	Coût
Multiplication Modulaire $\tilde{Mul}_t$	Algo 40	$3Mul_{t,t}$
$xm_j^{-1} \bmod m_i$	Algo 43	$(l+1)\tilde{Mul}_t = 3(l+1)Mul_{t,t}$
RNS $\rightarrow$ MRS	Algo 42	$\frac{n(n-1)}{2} 3(l+1)Mul_{t,t}$
Multiplication	Par une constante	$lMul_{t,t}$
$xm_j \bmod \tilde{m}_i$	Algo 20	$(l+1)Mul_{t,t}$
MRS $\rightarrow$ RNS'	Algo 42	$n(n-1)(l+1)Mul_{t,t}$
Changement de base `CDB`	Algo 42	$\frac{5}{2}(n-1)n(l+1)Mul_{t,t}$
Multiplication Modulaire $MulMod_k$	Algo 20	$(l^2 + l)Mul_{t,t}$
Multiplication Modulaire en RNS	$2\texttt{cdb} + 5\tilde{Mul}_k$ [5]	$5n(n+l-1)(l+1)$

TAB. 4.3 – Récapitulatif des choix et différents algorithmes utilisés.

Nous reprenons les cas intéressants du tableau 4.2 qui nous permettent de construire des bases efficaces pour des tailles cryptographiques dans le tableau 4.4.

				RNS amélioré	Montgomery
$\|p\|$	n	k	l	$5n(n+l-1)(l+1)Mul_t$	$2((nl)^2 + nl)Mul_t$
256	16	16	2	$4080Mul_8$	$2080Mul_8$
512	16	32	4	$7600Mul_8$	$8256Mul_8$
1024	16	64	8	$16560Mul_8$	$32896Mul_8$

TAB. 4.4 – Comparaison des méthodes de multiplication modulaires pour de grands moduli.

Cette méthode est intéressante pour l'implantation des protocoles de ECC mais devient vraiment concurrentielle pour l'implantation du protocole RSA qui utilise des tailles de $1024, 2048, 4096\ldots$

Figure 4.1, nous présentons les coûts en multiplication 16 bits de l'algorithme de Montgomery et de la multiplication modulaire en RNS avec des bases dont les éléments sont sur 32 ou 64 bits.

De même, sur la figure 4.2 nous présentons les coûts en multiplication 8 bits de l'algorithme de Montgomery et de la multiplication modulaire en RNS avec des bases dont les éléments sont sur 32 ou 64 bits.

Il apparaît que plus l est grand, plus cette méthode est efficace. Sur la figure 4.1 , nous observons que la solution la moins efficace est le choix du RNS avec des chiffres de 32 bits où $l = \frac{32}{16} = 2$. La seconde solution qui concurrence l'algorithme de Montgomery est le choix du RNS avec des chiffres de 64 bits où $l = \frac{64}{16} = 4$. De même, sur la seconde figure 4.2, c'est le choix du RNS avec des chiffres de 32 bits où $l = \frac{32}{8} = 4$ qui concurrence l'algorithme de Montgomery. Pour le dernier cas, c'est en utilisant du RNS avec des chiffres de 64 bits où $l = \frac{64}{8} = 8$ que cette méthode se révèle réellement compétitive.

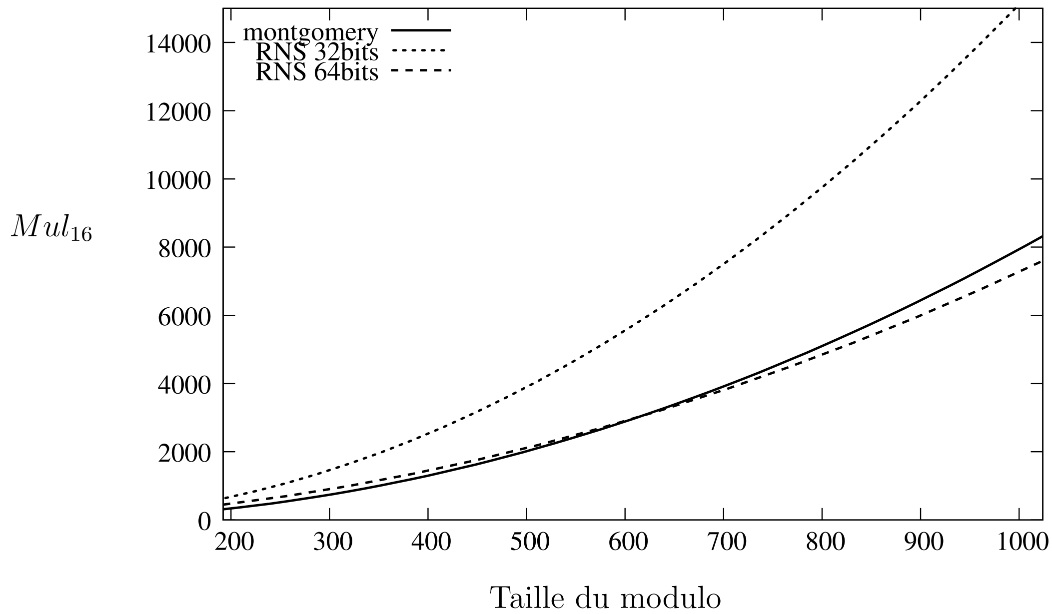

FIG. 4.1 – Comparaison des coûts de Montgomery avec des chiffres de 16bits, du RNS avec des chiffres sur 32 et 64 bits.

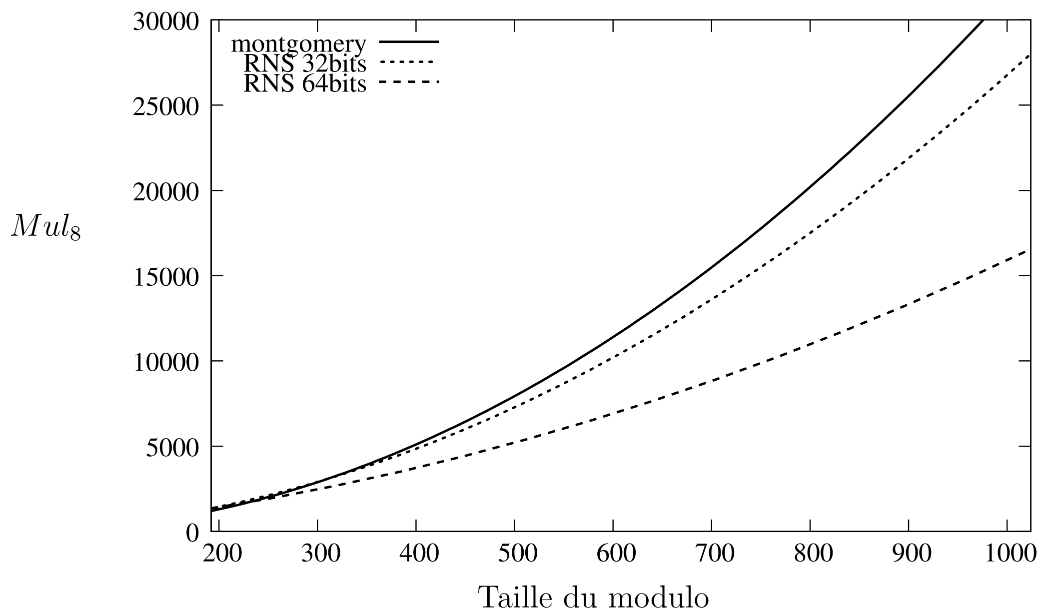

FIG. 4.2 – Comparaison des coûts de Montgomery avec des chiffres de 8bits, du RNS avec des chiffres sur 32 et 64 bits.

4.3 Carré et multiplication modulaire via une table des inverses

L'idée principale de cette section est l'utilisation de l'inversion modulaire pour effectuer un carré ou une multiplication modulaire. Comme Taylor [70] pour la mise au carré modulaire (Voir Section 2.3), nous proposons d'utiliser une table mémoire. Enfin nous montrons les applications cryptographiques de cette méthode.

4.3.1 La décomposition de la multiplication en inversion

La multiplication peut se décomposer en inversion en deux étapes :

i) Nous rappelons la décomposition de Taylor qui permet d'évaluer un produit à partir d'une table des carrés (Voir Équation 2.7 dans la section 2.3).

$$\frac{(a+b)^2}{4} - \frac{(a-b)^2}{4} = \frac{a^2+2ab+b^2}{4} - \frac{a^2-2ab+b^2}{4} = \frac{4ab}{4} = ab$$

ii) Avec des inverses nous observons que

$$\frac{1}{u-2} - \frac{1}{u+2} = \frac{(u+2)-(u-2)}{(u-2)(u+2)} = \frac{4}{u^2-4}$$

Nous pouvons remarquer que la seconde étape répond à une inversion prés au besoin de la première. Nous poursuivons la décomposition.

$$\frac{1}{\frac{1}{u-2} - \frac{1}{u+2}} = \frac{u^2-4}{4}$$

$$\frac{1}{\frac{1}{u-2} - \frac{1}{u+2}} - \frac{1}{\frac{1}{v-2} - \frac{1}{v+2}} = \frac{u^2-4}{4} - \frac{v^2-4}{4} = \frac{u^2}{4} - \frac{v^2}{4}$$

Nous terminons en posant $u = a+b$ et $v = a-b$.

$$\frac{1}{\frac{1}{a+b-2} - \frac{1}{a+b+2}} - \frac{1}{\frac{1}{a-b-2} - \frac{1}{a-b+2}} = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4} = ab$$

Nous décomposons la multiplication modulaire en inversion modulaire à l'aide de l'équation 4.26.

$$ab \equiv ((a+b-2)^{-1} - (a+b+2)^{-1})^{-1} - ((a-b-2)^{-1} - (a-b+2)^{-1})^{-1} \pmod{p} \quad (4.26)$$

4.3.2 L'utilisation d'une table mémoire

La décomposition de la multiplication en inversion ne semble pas utile car une multiplication modulaire est bien moins coûteuses qu'une inversion modulaire (Voir Section 2.2). Par contre si nous utilisons cette décomposition avec une table mémoire alors nous obtenons la multiplication et le carré avec une table de taille p fois celle des valeurs manipulées au lieu du cube p^2 dans le cadre du produit. Taylor [70] transforme une multiplication modulaire en deux mises au carré modulaires (certes plus coûteuse) mais il utilise lui aussi une table mémoire.

Nous proposons de mémoriser dans une table mémoire `MEM` tous les inverses modulo p.

$$\text{Pour tout } x \text{ tel que } 0 < x < p, \quad \texttt{MEM}[x] \leftarrow x^{-1} \bmod p$$

L'algorithme 45 utilise l'équation 4.26 et la table mémoire `MEM` pour effectuer une multiplication modulaire.

Algorithme 45 : Multiplication Modulaire par Inversion

Input : a, b, p tels que $0 \leq a, b < p$
Data : `MEM` avec $\mathtt{MEM}[x] = x^{-1} \bmod p$
Output : s tel que $s = ab \bmod p$
begin
  $s \leftarrow 0$
  $u \leftarrow a + b \bmod p$
  $v \leftarrow a - b \bmod p$
  if $u \notin \{-2, 2\}$ **then** $s \leftarrow \mathtt{MEM}[\mathtt{MEM}[u_1 - 2] - \mathtt{MEM}[u_2 + 2]]$
  if $v \notin \{-2, 2\}$ **then** $s \leftarrow s - \mathtt{MEM}[\mathtt{MEM}[v_1 - 2] - \mathtt{MEM}[v_2 + 2]]$
  if $s < 0$ **then** $s \leftarrow s + p$
end

Nous proposons aussi l'algorithme équivalent de mise au carré modulaire basé sur le même type d'équation :

$$\frac{1}{\frac{1}{x-1} - \frac{1}{x+1}} = \frac{x^2 - 1}{2}$$

Algorithme 46 : Mise au Carré Modulaire par Inversion

Input : a, p tels que $0 \leq a < p$
Data : `MEM` avec $\mathtt{MEM}[x] = x^{-1} \bmod p$
Output : s tel que $s = a^2 \bmod p$
begin
  $s \leftarrow 0$
  if $a \notin \{-1, 1\}$ **then** $s \leftarrow \mathtt{MEM}[\mathtt{MEM}[a - 1] - \mathtt{MEM}[a + 1]]$
  if $s < 0$ **then** $s \leftarrow s + p$
  $s \leftarrow 2s + 1$
  if $s \geq p$ **then** $s \leftarrow s - p$
end

Conclusion

Ce chapitre regroupe trois travaux sur l'arithmétique sur de "petits" moduli. Ils sont assez distincts à la fois par leurs méthodes, leurs contraintes d'utilisation ou leurs applications.

a) La multiplication modulaire à précalculs bornés possède plusieurs points intéressants. Premièrement, elle permet de lier l'efficacité de la réduction modulaire avec le temps de précalcul accordé : le temps de précalcul d'un algorithme de réduction diffère suivant le nombre de réduction (une seule ou un grand nombre comme pour l'exponentiation). Deuxièmement, il hiérarchise les classes de moduli en fonction de l'efficacité de leur réduction. Enfin, cet algorithme permet de faire le lien entre la réduction modulo un nombre de Pseudo Mersenne [22] et l'algorithme de Barrett [14] : l'algorithme proposé se comporte comme une réduction modulo un nombre Pseudo Mersenne pour la première classe et comme une réduction par l'algorithme de Barrett pour la dernière classe.

b) Le changement de base en RNS fait appel à différentes problématiques de l'arithmétique modulaire : cette opération nécessite l'utilisation d'inverses modulaires, de la multiplication modulaire, que ce soit par des constantes ou par des variables, le tout sur de petits moduli. Le changement de base est l'opération centrale de la multiplication modulaire pour de grands nombres [8] (pour RSA en particulier). Pour accélérer cette opération, nous avons amélioré ou modifié de nombreuses opérations : la multiplication modulaire par un inverse, la division par une petite constante, la multiplication modulaire . . . Ces nombreuses améliorations ont permis de faire ressortir des bases RNS ayant un changement de base efficace, et ainsi accélérer la multiplication modulo de grands entiers pour RSA.

c) La multiplication modulaire via une table des inverses est une amélioration de l'algorithme de Taylor [70]. Celui ci utilise une table mémoire pour effectuer le carré et la multiplication modulaire avec quelques additions. L'algorithme que nous proposons permet avec quelques additions de plus d'effectuer la carré et la multiplication modulaire mais permet aussi d'obtenir un inversion modulaire à très faible coût.

L'arithmétique modulo de “petits” moduli est utilisée dans des contextes très spécifiques. Nous avons proposé dans ce chapitre des algorithmes basés sur ces contraintes (temps de précalcul précis, forte mémorisation . . .), qui offrent une arithmétique très efficace pour ces contextes précis.

Conclusion

Bilan

Les travaux présentés dans cette thèse sont consacrés à l'arithmétique modulaire pour la cryptographie.

La cryptographie possède des contraintes concernant l'arithmétique modulaire. Nous avons commencé notre travail par spécifier les besoins et contraintes de cette arithmétique afin par la suite d'avoir des propositions les plus adéquates possibles. De cette analyse, il est apparu que nous pouvions complètement séparer la problématique des protocoles d'ECC [36] de celle du protocole RSA [58]. Les premiers utilisent des multiplications et des inversions modulo un nombre premier, de taille 160 à 512 bits mais surtout librement choisi. Le protocole RSA utilise des multiplications modulo un nombre composé de plus de 1024 bits, imposé par le protocole. Nous avons pu en déduire deux axes de recherche : une arithmétique très efficace modulo une classe particulière et une arithmétique modulaire généraliste fonctionnant pour tout modulo.

Le deuxième chapitre correspond à l'état de l'art des deux axes de recherches proposés ci-dessus. Trois opérateurs sont nécessaires à l'arithmétique modulaire pour la cryptographie : l'addition, la multiplication et l'inversion. Nous présentons les différents algorithmes existants pour l'addition et l'inversion. Nous développons particulièrement l'analyse de la multiplication modulaire. Il existe un grand nombre d'algorithmes effectuant cette opération. Ces différents algorithmes correspondent aux multiples contraintes existantes (mémorisation, précalcul, redondance ...). Nous les divisons en deux catégories : les algorithmes généralistes dont la complexité n'est pas liée au type de modulo utilisé (Montgomery [48], Barrett [14] ...) et les algorithmes opérant sur des classes particulières de nombre (nombres de Mersenne [35], nombres de Mersenne Généralisés [66] ...). Les algorithmes généralistes sont utilisés pour l'implantation du protocole RSA. Les classes de nombres particulières sont utilisées pour l'implantation des protocoles d'ECC.

Dans le chapitre suivant, nous proposons un nouveau système de représentation : le système de représentation adapté. Ce système polynomial inclut dans sa définition le modulo sur lequel les opérations sont effectuées. Après analyse de ce système de représentation, nous avons proposé deux types de solutions aux différentes contraintes de l'arithmétique modulaire pour la cryptographie. Pour les protocoles d'ECC, nous avons défini une nouvelle classe de nombres. Cette classe inclut toutes les autres classes dérivant de celle des nombres de Mersenne. De plus, nous avons extrait de cette classe une liste de moduli pour lesquels le coût des opérations modulaires est optimisé. Le coût de la réduction modulaire pour ces nombres est inférieur à celui des nombres de Mersenne Généralisés proposés dans les standard (NIST [50] et SEC [67]).

Le seconde partie du travail sur ce système de représentation est axé sur l'algorithmique plus générale, pour l'implantation du protocole RSA entre autre. Le point central de ce travail est le théorème fondamental (Théorème 5) qui assure l'existence d'un système de représentation adapté avec une redondance maîtrisée. Ce théorème utilise une théorie particulièrement puissante : la géométrie des nombres ou encore théorie des réseaux euclidiens. La démonstration utilise en par-

ticulier le théorème fondateur de la géométrie des nombres : le théorème de Minkowski [47]. Les outils algorithmique existants (LLL [41], Babai [3]) ont pu être utilisés pour résoudre les différents problèmes posés par l'algorithmique généraliste sur les systèmes de représentation adaptés. Plusieurs algorithmes aux contraintes variées (précalcul, mémorisation ...) ont pu émerger de ces travaux.

Le dernier chapitre de cette thèse regroupe différents travaux effectués sur l'algorithmique modulaire pour de "petits" moduli. Au delà de la cryptographie, d'autres utilisations sont faites de l'arithmétique modulaire : pour le traitement du signal, pour les opérations sur les très grands nombres à l'aide du RNS (Residue Number System) ... Nous proposons trois travaux aux contraintes très variés. Le premier porte sur la réduction modulaire avec précalcul borné : ce travail permet de lié les classes de moduli, le temps de précalcul et l'efficacité de la réduction modulaire. Le deuxième travail porte sur le changement de base en RNS. Le choix de bases particulières pour ce système et la proposition de toute une algorithmique particulière dédiée au changement de base, a permis d'améliorer cette opération centrale en RNS. Le changement de base est une opération très coûteuse en RNS et elle est utilisée en particulier pour effectuer une multiplication modulaire sur de grands entiers [8] (pour les protocole RSA par exemple). Le dernier travail se pose dans le contexte d'une très forte mémorisation. Dans ce contexte, il existe déjà des travaux faits par Taylor [70] qui propose une multiplication à partir de quelques additions. Pour quelques additions de plus, nous avons pu offrir une algorithmique complète : multiplication et inversion.

Perspectives

Le premier objectif suite à cette thèse, est l'implantation logicielle et matérielle des différentes propositions. Elle permettra de juger de la réelle qualité des approches que nous avons faites. Ceci est particulièrement vrai pour l'implantation des systèmes de représentation adaptés qui permettent une parallélisation qui n'a pas du tout été évaluée : une implantation matérielle de ce système pourrait se révéler très intéressante.

C'est autour de ce système de représentation adaptés que sont envisageables plusieurs travaux futurs. Le théorème fondamental offre en effet une solide base pour construire une algorithmique généraliste complète. En particulier pour la multiplication modulaire, la forme polynomial de ce système permet d'utiliser l'interpolation. L'interpolation est en effet utilisée dans la majorité des algorithmes éfficaces de multiplication classique (Karatsuba [34], Toom Cook [35], Schönhage Strassen [62] ...). Ceci devrait permettre d'inclure totalement la réduction dans la multiplication, là où pour l'instant les deux opérations sont souvent consécutives et donc plus coûteuses.

ANNEXE

Annexe A

Moduli standards de la cryptographie

Sommaire

A.1 Moduli conseillés par le NIST 113
A.2 Moduli conseillés par le SEC 113

A.1 Moduli conseillés par le NIST

Modulo pour 192 bits

$p_{192} = 2^{192} - 2^{64} - 1$

Modulo pour 224 bits

$p_{224} = 2^{224} - 2^{96} + 1$

Modulo pour 256 bits

$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

Modulo pour 384 bits

$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

Modulo pour 521 bits

$p_{521} = 2^{521} - 1$

A.2 Moduli conseillés par le SEC

Modulo pour 112 bits

$secp_{112} = \frac{2^{128}-3}{76439}$

Modulo pour 128 bits

$secp_{128} = 2^{128} - 2^{97} - 1$

Moduli pour 160 bits

$secp_{160a} = 2^{160} - 2^{32} - 2^{14} - 2^{12} - 2^{9} - 2^{8} - 2^{7} - 2^{3} - 2^{2} - 1$
$secp_{160b} = 2^{160} - 2^{31} - 1$

Moduli pour 192 bits

$secp_{192a} = 2^{192} - 2^{32} - 2^{12} - 2^{8} - 2^{7} - 2^{6} - 2^{3} - 1$
$secp_{192b} = 2^{192} - 2^{64} - 1$

Moduli pour 224 bits

$secp_{224a} = 2^{224} - 2^{32} - 2^{12} - 2^{11} - 2^{9} - 2^{7} - 2^{4} - 2 - 1$
$secp_{224b} = 2^{224} - 2^{96} + 1$

Moduli pour 256 bits

$secp_{256a} = 2^{256} - 2^{32} - 2^{9} - 2^{8} - 2^{7} - 2^{6} - 2^{4} - 1$
$secp_{256b} = 2^{224}(2^{32} - 1) + 2^{192} + 2^{96} - 1$

Modulo pour 384 bits

$secp_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

Modulo pour 521 bits

$secp_{521} = 2^{521} - 1$

Annexe B

Systèmes de représentations adaptés à cryptographie

Sommaire

B.1 Proposition de systèmes de représentation adaptés à différentes tailles de moduli . . . 115
B.1.1 Modulo pour 128 bits . . . 115
B.1.2 Modulo pour 160 bits . . . 116
B.1.3 Moduli pour 192 bits . . . 116
B.1.4 Modulo pour 224 bits . . . 117
B.1.5 Modulo pour 256 bits . . . 117
B.1.6 Moduli pour 288 bits . . . 118
B.1.7 Moduli pour 320 bits . . . 119
B.1.8 Modulo pour 352 bits . . . 120
B.1.9 Moduli pour 384 bits . . . 120
B.1.10 Modulo pour 416 bits . . . 122
B.1.11 Moduli pour 448 bits . . . 123
B.1.12 Moduli pour 480 bits . . . 124
B.1.13 Modulo pour 512 bits . . . 126
B.2 Systèmes de représentation adaptés avec $X^n - 2$ et $\xi = X^i + 1$ . . . 126

B.1 Proposition de systèmes de représentation adaptés à différentes tailles de moduli

B.1.1 Modulo pour 128 bits

$\mathcal{B}_{128}$ proposée pour 128 bits

1. $p = 340282366604025813516997721482669850617$
2. $n = 4$
3. $\gamma = 85070591670813493993980456716992511998$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^4 - 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (X^3 + 1)$
7. Matrice de réduction :

$$B = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 1 & 2 & 0 \\ 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 1 \end{pmatrix}$$

B.1.2 Modulo pour 160 bits

$\mathcal{B}_{160}$ proposé pour 160 bits

1. $p = 1461501635629491084391274140357585917716910308863$
2. $n = 5$
3. $\gamma = 1141798153101385508590078088964518468124671999 6$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^5 - 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (2X^4 + 1)$
7. Matrice de réduction :

$$B = \begin{pmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 1 & 4 & 0 & 0 \\ 0 & 0 & 1 & 4 & 0 \\ 0 & 0 & 0 & 1 & 4 \\ 2 & 0 & 0 & 0 & 1 \end{pmatrix}$$

B.1.3 Moduli pour 192 bits

$\mathcal{B}_{192_a}$ proposé pour 192 bits

1. $p = 6277101709079651337817656387084824259474884836727266977721$
2. $n = 6$
3. $\gamma = 2032631443346898800397288080957128056888104126243072369514$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^6 - 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - 3(X^5 + 1)$
7. Matrice de réduction :

$$B = 3 \times \begin{pmatrix} 1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 & 2 & 0 \\ 0 & 0 & 0 & 0 & 1 & 2 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$\mathcal{B}_{192_b}$ **proposé pour** 192 **bits**

1. $p = 6277101726617670944954607416215071121599116894624280477697$
2. $n = 6$
3. $\gamma = 748288837442299435208332359863083989853465892351488$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^6 - 2^5$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (2^7 X^5 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = 3 \times \begin{pmatrix} 1 & 4096 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4096 & 0 & 0 & 0 \\ 0 & 0 & 1 & 4096 & 0 & 0 \\ 0 & 0 & 0 & 1 & 4096 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4096 \\ 128 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

B.1.4 Modulo pour 224 bits

$\mathcal{B}_{224}$ **proposé pour** 224 **bits**

1. $p = 26959946623210927677651784112208183154001000463259786712774250332159$
2. $n = 7$
3. $\gamma = 26933618550336698256091577291786104850139748732497285870384222060543$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^7 - 4$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (4X^5 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 16 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 16 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 16 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 16 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 16 \\ 4 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

B.1.5 Modulo pour 256 bits

$\mathcal{B}_{256}$ **proposé pour** 256 **bits**

1. $p = 115792089021636622262124715160334756877804245386980633020041035952359812890593$
2. $n = 8$
3. $\gamma = 14474011127704577782765589395224532314179217058921488395049827733759590399996$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^8 - 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (X^5 + 1)$
7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
1 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 2 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 2 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

B.1.6 Moduli pour 288 bits

$\mathcal{B}_{288a}$ proposé pour 288 bits

1. $p = 497323236409786642155382248115435331423522746978168770324742557462704777140587378769919$
2. $n = 9$
3. $\gamma = 497323236178202463680749857268293361419300446007606927257294592878180816069443383721983$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^9 - (X^3 + 1)$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - X^5$
7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

$\mathcal{B}_{288b}$ proposé pour 288 bits

1. $p = 497323235367657839990094568824828462041945545349236437619473251602052119101286605914111$
2. $n = 9$
3. $\gamma = 388533777649075201397679612463958312408994823253350570160745892712804363966031215984 64$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^9 - 4$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (256X^7 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 1024 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1024 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1024 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1024 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1024 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1024 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1024 \\ 256 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 256 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

B.1.7 Moduli pour 320 bits

$\mathcal{B}_{320_a}$ proposé pour 320 bits

1. $p = 21359870354235868459852350640141698664558836822561966191496938903817557488874810530104287114035 21$
2. $n = 10$
3. $\gamma = 2135987034926263609691240511096103813732633827164428376205616157606932336033779140005808 46$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{10} - (X - 1)$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (-X^9 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$\mathcal{B}_{320_b}$ proposé pour 320 bits

1. $p = 21359870309476777235077992970597648284553050354459098527979456030651190752831865331863583 99205377$
2. $n = 10$
3. $\gamma = 66749594717114928859618728033117650889228282357684682899935800095784971102599579162073834 192896$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{10} - 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (32X + 1)$

7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 64 \\ 32 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 32 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 32 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 32 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 32 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 32 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 32 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 & 1 \end{pmatrix}$$

B.1.8 Modulo pour 352 bits

$\mathcal{B}_{352}$ proposé pour 352 bits

1. $p = 9173994440464428678666050658295679598606820496488998559285440608353738183621929033091248019400881787109371$
2. $n = 11$
3. $\gamma = 4586997220232214339333025329147839799303410248247637830506029139649346395519072052106497355124047871999998$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{11} - 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (X^2 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

B.1.9 Moduli pour 384 bits

$\mathcal{B}_{384a}$ proposé pour 384 bits

1. $p = 39402006196394479212279040100143613805079739270465446667433571061944013948862675878711237608764522494360577160249343$
2. $n = 12$
3. $\gamma = 27173797373027643126829564463639754245761883181071510823054963010376399786188005479186486584419141314216779441558492$

4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{12} - (2X^6 + 1)$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - X^5$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

$\mathcal{B}_{384b}$ **proposé pour** 384 **bits**

1. $p = 39402006086306545785730751804193164315815021596355911139433085982544209289648889841937882994060312769795918212890753$
2. $n = 12$
3. $\gamma = 36939380705912386674122579816431091546076582746581981696554567425655342972538821215369640244402589485014124932890745$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{12} + 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (-X^7 + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$\mathcal{B}_{384c}$ **proposé pour** 384 **bits**

1. $p = 39402006086306545785730751804193164315815021596355911139433085982544209289648889841937882994060312769795917676019713$

2. $n = 12$

3. $\gamma = 9619630392164684029719421827195596756790776756922829868025655757457439158028447102788641178759929286062899068928$

4. $\rho = 2^{32}$

5. Polynôme de réduction externe : $E(X) = X^{12} - 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (4X^5 + 1)$

7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

B.1.10 Modulo pour 416 bits

$\mathcal{B}_{416}$ proposé pour 416 bits

1. $p = 16923032749807756149413365667477188459898656902606732952379128913595031979067716390418092688102572260374433254886008178710911 9$

2. $n = 13$

3. $\gamma = 14807653656081786630736694959042539902411324789780891333331737799395652981684270110386276470728305618754383567849228617750915 1$

4. $\rho = 2^{32}$

5. Polynôme de réduction externe : $E(X) = X^{13} - 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (X^8 + 1)$

7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 2 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

B.1.11 Moduli pour 448 bits

$\mathcal{B}_{448a}$ **proposé pour** 448 **bits**

1. $p = 726838721926382301990655389776903695938161504111219182772671511244795432245749664871811594716791834636824152838958040806659926802825217$
2. $n = 14$
3. $\gamma = 721160294411332440256353394544271635813644617360350282907260015063195467931329745615000566633066898428723963525933661902513322802020353$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{14} + 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (4X^5 + 1)$
7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -8 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -8 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -8 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -8 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -8 \\
4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$\mathcal{B}_{448b}$ **proposé pour** 448 **bits**

1. $p = 726838721926382301990655389776903695938161504111219182772671511244795432245749664871811594716791834636824152838958040662544730137034753$

2. $n = 14$

3. $\gamma = 45427420120398893874415961861056480996135094006951198923291969452799714515359354054488$
$2246697994896648015095524348775414090459020001 28$

4. $\rho = 2^{32}$

5. Polynôme de réduction externe : $E(X) = X^{14} - 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (16X + 1)$

7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 32 \\ 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 16 & 1 \end{pmatrix}$$

B.1.12 Moduli pour 480 bits

$\mathcal{B}_{480_a}$ **proposé pour** 480 **bits**

1. $p = 31217485394134113847166782947072787204190280025856944325852515605746211204608832133816 4$
$116671041571708820793910940618714187976691174008178712575 9$

2. $n = 15$

3. $\gamma = 31213674665935700455923322179742833671340919757670269697023540421733336091533019537813 4$
$485435304483973080986784796255550858132479654008178712575 7$

4. $\rho = 2^{32}$

5. Polynôme de réduction externe : $E(X) = X^{15} + 2$

6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (-X^{14} + 1)$

7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 2 \\
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

$\mathcal{B}_{480_b}$ **proposé pour** 480 **bits**

1. $p = 3121748549589153507085990339243842497860593608311329790171838899343881539639126332006908179737300920446063365560644101289181797745753683677675519$
2. $n = 15$
3. $\gamma = 72683872412637656253902016655631421549725197449124646144263420704376919652315004076969534618251512768819540868528282765512862190272512 0$
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{15} + X - 1$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (X^{14} + 1)$
7. Matrice de réduction :

$$
\boldsymbol{B} = \begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
$$

B.1.13 Modulo pour 512 bits

$\mathcal{B}_{512}$ proposé pour 512 bits

1. $p =$ 13407807879994620381738796116804133702649173966794733022817237752015709119395748728288655164446878240747825728634251000645023123708146793592887881358114817
2. $n = 16$
3. $\gamma =$ 99895953261229164310933705430632919053408896082742221842728049938387875854748262828212517334733302946822654051500997988540152541175682548468023248
4. $\rho = 2^{32}$
5. Polynôme de réduction externe : $E(X) = X^{16} - 2$
6. Polynôme de réduction interne : $\xi(X) = 2^{32} - (8X^{15} + 1)$
7. Matrice de réduction :

$$\boldsymbol{B} = \begin{pmatrix}
1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 16 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}$$

B.2 Systèmes de représentation adaptés avec $X^n - 2$ et $\xi = X^i + 1$

Moduli				Critères			Coût des réductions			
$\mathcal{B}$	$\lvert p\rvert$	n	k	$\sim \pi$	ω	ϕ	`RedExt`	`RedInt`	Total	Gain
$secp_{128r1}$	128	4	32	-	-	-	13	6	19	-
$\mathcal{B}_{128}$	128	4	32	1	3	8	6	8	14	-26%
$secp_{160r1}$	160	5	32	-	-	-	~ 12	8	20	-
$\mathcal{B}_{160}$	160	5	32	1	5	10	8	10	18	-10%
$secp_{192r1}$	192	6	32	-	-	-	14	10	24	-
$\mathcal{B}_{192_a}$	192	6	32	1	9	12	10	$12+6$	28	$+17\%$
$\mathcal{B}_{192_b}$	192	6	32	1	4097	12	10	12	22	-8%
$secp_{224r1}$	224	7	32	-	-	-	17	12	29	-
$\mathcal{B}_{224}$	224	7	32	1	17	14	12	14	26	$-11,5\%$
$secp_{256r1}$	256	8	32	-	-	-	46	14	60	-
$\mathcal{B}_{256}$	256	8	32	1	3	16	14	16	30	-50%
$\mathcal{B}_{288_a}$	288	9	32	1	2	14	36	14	50	
$\mathcal{B}_{288_b}$	288	9	32	1	1025	18	16	18	34	
$\mathcal{B}_{320_a}$	320	10	32	2	2	11	36	11	47	
$\mathcal{B}_{320_b}$	320	10	32	1	33	20	18	20	38	
$\mathcal{B}_{352}$	352	11	32	1	3	22	20	22	42	
$secp_{384r1}$	384	12	32	-	-	-	62	22	84	-
$\mathcal{B}_{384_a}$	384	12	32	1	3	19	54	19	73	-13%
$\mathcal{B}_{384_b}$	384	12	32	2	3	24	22	24	46	-45%
$\mathcal{B}_{384_c}$	384	12	32	1	9	24	22	24	46	-45%
$\mathcal{B}_{416}$	416	13	32	1	3	26	24	26	50	
$\mathcal{B}_{448_a}$	448	14	32	2	9	28	26	28	54	
$\mathcal{B}_{448_b}$	448	14	32	1	33	28	26	28	54	
$\mathcal{B}_{480_a}$	480	15	32	2	3	30	28	30	58	
$\mathcal{B}_{480_b}$	480	15	32	1	2	16	56	16	78	
$\mathcal{B}_{512}$	512	16	32	1	17	32	30	32	62	

TAB. B.1 – Comparaison des moduli pour différentes tailles cryptographiques.

X^2-2	X^3-2	X^4-2	X^5-2	X^6-2	X^7-2
$2^{129}-(X+1)$	$2^{54}-(X^2+1)$	$2^{57}-(X+1)$	$2^{33}-(X^3+1)$	$2^{59}-(X^6+1)$	$2^{44}-(X+1)$
$2^{132}-(X+1)$	$2^{116}-(X^2+1)$	$2^{63}-(X^3+1)$	$2^{44}-(X^2+1)$		$2^{48}-(X^3+1)$
$2^{159}-(X+1)$	$2^{132}-(X^2+1)$	$2^{83}-(X+1)$	$2^{78}-(X^4+1)$		$2^{62}-(X^2+1)$
$2^{171}-(X+1)$	$2^{165}-(X+1)$	$2^{86}-(X+1)$			
$2^{175}-(X+1)$		$2^{117}-(X^3+1)$			

X^8-2	X^9-2	$X^{10}-2$	$X^{11}-2$	$X^{12}-2$	$X^{13}-2$
$2^{28}-(X^7+1)$	$2^{20}-(X^7+1)$	$2^{36}-(X^3+1)$	$2^{15}-(X^5+1)$	$2^{31}-(X^{11}+1)$	$2^{20}-(X^{11}+1)$
$2^{32}-(X^5+1)$	$2^{23}-(X^7+1)$	$2^{42}-(X^9+1)$	$2^{32}-(X^2+1)$	$2^{38}-(X^{11}+1)$	$2^{21}-(X^9+1)$
$2^{60}-(X^5+1)$	$2^{36}-(X^8+1)$		$2^{35}-(X^5+1)$	$2^{43}-(X^5+1)$	$2^{24}-(X^8+1)$
	$2^{38}-(X+1)$		$2^{38}-(X^4+1)$		$2^{32}-(X^8+1)$
			$2^{44}-(X+1)$		$2^{34}-(X+1)$

TAB. B.2 – Systèmes de représentation adaptés avec X^n-2 et $\xi = X^i+1$.

Bibliographie

[1] L. M. Adleman and J. DeMarrais. A subexponential algorithm for discrete logarithms over all finite fields. In Douglas R. Stinson, editor, *Proc. CRYPTO 93*, pages 147–158. Springer, 1994. Lecture Notes in Computer Science No. 773.

[2] D. Agrawal, J. R. Rao, and P. Rohatgi. Multi-channel attacks. In *CHES : International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, 2003.

[3] L. Babai. On Lovasz' lattice reduction and the nearest lattice point problem. *COMBINAT : Combinatorica*, 6, 1986.

[4] J.-C. Bajard, L.-S Didier, and P. Kornerup. An IWS montgomery modular multiplication algorithm.

[5] J.-C. Bajard, L.-S Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7) :766–776, 1998.

[6] J.-C. Bajard, L.-S Didier, and P. Kornerup. Montgomery modular multiplication in residue arithmetic, November 08 2000.

[7] J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEETC : IEEE Transactions on Computers*, 53, 2004.

[8] J.-C. Bajard, L. Imbert, P. Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *CHES : International Workshop on Cryptographic Hardware and Embedded Systems, CHES, LNCS*, 2004.

[9] J.-C. Bajard, L. Imbert, and T. Plantard. Improving euclidean division and modular reduction for some classes of divisors. In *37th IEEE Asilomar Conference on Signals, Systems, and Computers*, November 2003.

[10] J.-C. Bajard, L. Imbert, and T. Plantard. Modular number systems : Beyond the Mersenne family. In *SAC'04 : 11th International Workshop on Selected Areas in Cryptography*, pages 159–169, August 2004.

[11] J.-C. Bajard, L. Imbert, and T. Plantard. Arithmetic operations in the polynomial modular number system. In *ARITH'17 : IEEE Symposium on Computer Arithmetic*, June 2005.

[12] J.-C. Bajard, N. Meloni, and T. Plantard. Efficient RNS bases for cryptography. In *IMACS'05 : World Congress : Scientific Computation, Applied Mathematics and Simulation*, July 2005.

[13] J.-C. Bajard and T. Plantard. RNS bases and conversions. In *SPIE'04 : Advanced Signal Processing Algorithms, Architectures and Implementations XIV*, August 2004.

[14] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology – CRYPTO '86*, volume 263 of *LNCS*, pages 311–326. Springer-Verlag, 1986.

[15] G. R. Blakley. A Computer Algorithm for Calculating the Product AB modulo M. *IEEE Transactions on Computers*, C-32 :497–500, 1983.

[16] A. Bosselaers, R. Govaerts, and J. Vandewalle. Comparison of three modular reduction functions. In D. R. Stinson, editor, *Advances in Cryptology – CRYPTO '93*, volume 773 of *LNCS*, pages 175–186. Springer-Verlag, 1994.

[17] J. W. S. Cassels. *An Introduction to The Geometry of Numbers.* Springer-Verlag, 1959.

[18] J. Chung and A. Hasan. More generalized Mersenne numbers. In M. Matsui and R. Zuccherato, editors, *Selected Areas in Cryptography – SAC 2003*, volume 3006 of *LNCS*, Ottawa, Canada, August 2003. Springer-Verlag.

[19] H. Cohen. *A course in computational algebraic number theory*, volume 138 of *Graduate Texts in Mathematics*. Springer-Verlag, 1993.

[20] H. Cohen and G. Frey. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* Chapman and Hall/CRC, 2006.

[21] J. H. Conway and N. J. A. Sloane. *Sphere Packings, Lattices and Groups.* Springer Verlag, 1988.

[22] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent number 5159632, 1992.

[23] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6) :644–654, November 1976.

[24] I. Dinur. Approximating SVP∞ to within almost-polynomial factors is NP-hard. *Theor. Comput. Sci.*, 285(1) :55–71, 2002.

[25] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology : Proceedings of CRYPTO '84*, volume 196 of *LNCS*, pages 10–18. Springer-Verlag, 1985.

[26] P. Giorgi. *Arithmétique et algorithmique en algèbre linéaire exacte pour la bibliothèque Linbox.* PhD thesis, Ecole Normale Supérieure de Lyon, 2004.

[27] O. Goldreich and S. Goldwasser. On the limits of non-approximability of lattice problems. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 1–9, New York, May 23–26 1998. ACM Press.

[28] O. Goldreich, S. Goldwasser, and S. Halevi. Public-key cryptosystems from lattice reductions problems. In *Proc. 17th International Cryptology Conference – CRYPTO '97*, pages 112–131, 1997.

[29] D. M. Gordon. A survey of fast exponentiation methods. *ALGORITHMS : Journal of Algorithms*, 27, 1998.

[30] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.

[31] A. A. Hiasat. New Efficient Structure for a Modular Multiplier for RNS. *IEEE Transactions on Computers*, 49(2) :170–174, Feb 2000.

[32] J. Hoffstein, J. Pipher, and J. H. Silverman. NTRU : A ring-based public key cryptosystem. In *ANTS*, pages 267–288, 1998.

[33] R. Kannan. Improved algorithms for integer programming and related lattice problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 193–206, Boston, Massachusetts, 25–27 April 1983.

[34] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Doklady Akademii Nauk SSSR*, 145(2) :293–294, 1962.

[35] D. E. Knuth. *The Art of Computer Programming, Vol. 2. Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1981.

[36] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177) :203–209, January 1987.

[37] N. Koblitz. *A Course in Number Theory and Cryptography.* Springer-Verlag, 1994.

[38] P. C. Kocher. Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems. 1996.

[39] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. 1999.

[40] A. K. Lenstra, H. W. Lenstra, M. S. Manasse, and J. M. Pollard. The number field sieve. In ACM, editor, *Proceedings of the twenty-second annual ACM Symposium on Theory of Computing, Baltimore, Maryland, May 14–16, 1990*, pages 564–572, New York, NY, USA, 1990. ACM Press.

[41] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Ann.*, 261 :513–534, 1982.

[42] L. Lovász. *An Algorithmic Theory of Numbers, Graphs and Convexity*, volume 50 of *CBMS-NSF Regional Conference Series in Applied Mathematics.* SIAM Publications, 1986.

[43] U. M. Maurer. Towards the equivalence of breaking the Diffie-Hellman protocol and computing discrete algorithms. In Yvo G. Desmedt, editor, *Advances in Cryptology—CRYPTO '94*, volume 839 of *Lecture Notes in Computer Science*, pages 271–281. Springer-Verlag, 21–25 August 1994.

[44] U. M. Maurer and S. Wolf. Diffie-hellman oracles. In *CRYPTO*, pages 268–282, 1996.

[45] A. J. Menezes, P. C. van Oorschot, and S. A. Vanston, editors. *Handbook of Applied Cryptography.* CRC Press, 1996.

[46] D. Micciancio and S. Goldwasser. *Complexity of Lattice Problems, A Cryptographic Perspective.* Kluwer Academic Publishers, 2002.

[47] H. Minkowski. *Geometrie der Zahlen. I.* B. G. Teubner, Leipzig, 1896.

[48] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) :519–521, Apr 1985.

[49] National Institute of Standards and Technology. *FIPS PUB 46-3 : Data Encryption Standard (DES).* National Institute for Standards and Technology, October 1999.

[50] National Institute of Standards and Technology. *FIPS PUB 186-2 : Digital Signature Standard (DSS).* National Institute for Standards and Technology, Gaithersburg, MD, USA, January 2000.

[51] National Institute of Standards and Technology. *FIPS PUB 197 : Advanced Encryption Standard (AES).* FIPS PUB. National Institute for Standards and Technology, November 2001.

[52] P. Q. Nguyen. *La Géométrie des Nombres en Cryptologie.* PhD thesis, Université de Paris 7, 1999.

[53] P. Q. Nguyen and J. Stern. The two faces of lattices in cryptology. In *CaLC*, pages 146–180, 2001.

[54] J. Omura. A public key cell design for smart card chips. In *IT Workshop, Hawaii, USA, November 27–30*, pages 983–985, 1990.

[55] M. Pohst and H. Zassenhaus. *Algorithmic Algebraic Number Theory.* Cambridge University Press, Cambridge, 1989.

[56] J. M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32 :918–924, 1978.

[57] K. C. Posch and R. Posch. Modulo reduction in residue number systems. *IEEETPDS : IEEE Transactions on Parallel and Distributed Systems*, 6, 1995.

[58] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2) :120–126, Feb 1978.

[59] C. P. Schnorr. A hierarchy of polynomial time lattice basis reduction algorithms. *Theoretical Computer Science*, 53(2–3) :201–224, 1987.

[60] C.-P. Schnorr. Factoring integers and computing discrete logarithms via diophantine approximations. In *EUROCRYPT*, pages 281–293, 1991.

[61] C.-P. Schnorr and M. Euchner. Lattice basis reduction : Improved practical algorithms and solving subset sum problems. *Math. Program*, 66 :181–199, 1994.

[62] A. Schonhage and V. Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7 :281–292, 1971.

[63] V. Shoup. Lower bounds for discrete logarithms and related problems. In *EUROCRYPT*, pages 256–266, 1997.

[64] J. H. Silverman. *The Arithmetic of Elliptic Curves*, volume 106 of *Graduate Texts in Mathematics*. Springer-Verlag, 1986.

[65] J. H. Silverman and J. Tate. *Rational Points on Elliptic Curves*. Springer-Verlag, 1992.

[66] J. Solinas. Generalized Mersenne numbers. Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 1999.

[67] Standard for Efficient Cryptography. *SECG SEC2 : Recommended Elliptic Curve Cryptography Domain Parameters*. Standard for Efficient Cryptography Group, Sep 2000.

[68] N. S. Szabó and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill series in information processing and computers. McGraw-Hill, New York, NY, USA, 1967.

[69] N. Takagi. A radix-4 modular multiplication algorithm for modular exponentiation. *IEEE Transactions on Computers*, 41(8) :949–956, August 1992.

[70] F. J. Taylor. Large moduli multipliers for signal processing. C-28 :731–736, July 1981.

[71] F. J. Taylor. Residue arithmetic : A tutorial with examples. *IEEE Computer*, 17(5) :50–62, May 1984.

[72] P. van Emde Boas. Another NP-complete partition problem and the complexity of computing short vectors in lattices. Technical Report 81-04, Mathematics Department, University of Amsterdam, 1981.