UNIVERSITÉ DE TOULON

Habilitation à Diriger des Recherches

Discipline : Informatique
École doctorale : ED 548 – Mer et Sciences

Arithmétique et Représentation des Nombres

par

Nicolas Méloni

Jury

Rapporteur :	Philippe Elbaz-Vincent, Professeur des universités, Univ. Grenoble Alpes
Rapporteur :	Laurent Imbert, Directeur de recherche, CNRS
Rapporteur :	Arnaud Tisserand, Directeur de recherche, CNRS, ENSTA Bretagne
Examinatrice :	Valérie Berthé, Directrice de recherche, CNRS, Université Paris Cité
Examinateur :	Pascal Véron, Maître de conférences HDR, Université de Toulon
Garant :	Philippe Langevin, Professeur des universités, Université de Toulon

Date de soutenance : 10/12/2027

Table des matières

Préambule **3**

1 Introduction **4**
1.1 Contexte 4
1.2 Quatre niveaux d'arithmétique pour la cryptographie basée sur les courbes elliptiques 5
1.3 Résumé des principales contributions 6

2 Arithmétique des exposants **10**
2.1 Représentation à chiffres aléatoires 10
2.1.1 Algorithme de recodage 10
2.1.2 Densité moyenne 12
2.1.3 Ensembles optimaux 13
2.2 Schéma d'exponentiation randomisé 14
2.2.1 Densité moyenne et problème d'urne 15
2.2.2 Schéma de précalcul 16

3 Arithmétique des courbes elliptiques **18**
3.1 Courbes à endomorphisme et arithmétique co-Z 19
3.1.1 Courbes à endomorphisme efficace et méthode GLV 19
3.1.2 Arithmétique Co-Z 20
3.1.3 Chaînes d'addition euclidiennes et multiplication sur point fixe . 20
3.2 Injectivité depuis une paire (aP, bP) quelconque 22
3.3 Extension à la multiplication scalaire d'un point variable 23
3.3.1 Résultat principal d'injectivité 23
3.3.2 Algorithme de multiplication scalaire 24
3.4 Coût théorique et performances pratiques 25
3.4.1 Analyse théorique 25
3.4.2 Coût pratique de la multiplication modulaire 26
3.4.3 Résultats expérimentaux 26

4 Arithmétique des corps finis **28**
4.1 Le système de représentation PMNS 29

4.1.1 Définition . . . 29
4.1.2 Le réseau associé au PMNS . . . 30
4.1.3 Réduction interne par l'algorithme de Montgomery . . . 30
4.2 Réduction interne par les algorithmes de Babai . . . 31
4.2.1 Préliminaires : orthogonalisation de Gram-Schmidt . . . 31
4.2.2 Algorithme du plan le plus proche . . . 31
4.2.3 Algorithme par arrondi . . . 34
4.2.4 Versions optimisées et compromis erreur-vitesse . . . 36
4.3 Résultats . . . 38
4.3.1 Condition de validité . . . 38
4.3.2 Génération des paramètres . . . 38
4.3.3 Comparaison des complexités . . . 38
4.3.4 Résultats expérimentaux . . . 39

5 Arithmétique des ordinateurs **41**
5.1 Génération de paramètres . . . 42
5.1.1 Limitation de l'approche classique . . . 42
5.1.2 Algorithme de Montgomery sur les réseaux . . . 43
5.2 PMNS en arithmétique 128 bits . . . 44
5.2.1 Le problème de la taille des données . . . 44
5.2.2 Solution : $\phi = 2^{128}$ . . . 44

Préambule

Historiquement cantonnée aux sphères militaires et politiques, la cryptographie est devenue, depuis quelques décennies, un véritable enjeu de société. Du commerce électronique aux communications personnelles ou professionnelles, en passant par le vote électronique, la numérisation croissante de nos activités publiques, comme privées, rend indispensable la sécurisation de nos données.

L'une des principales avancées dans ce domaine a été l'introduction dans les années 70 du concept de cryptographie publique, rendant possible l'échange sécurisé de clés de chiffrement à grande échelle. Qu'il s'agisse de protocoles plus anciens, mais toujours en vigueur, comme RSA ou bien des standards actuels comme les protocoles basés sur les courbes elliptiques, la sécurité de ces approches repose sur la difficulté à résoudre certains problèmes de théorie des nombres. Ainsi, le déroulement de ces protocoles nécessite de nombreux calculs sur de grands entiers pouvant aller de quelques centaines à quelques milliers de bits. Par exemple, les protocoles reposant sur la courbe **Ed25519** nécessitent le calcul de multiplications de points sur cette courbe entraînant elles-mêmes plusieurs milliers d'additions et de multiplications modulaires sur le corps $\mathbb{F}_{2^{255}-19}$. En considérant par exemple que le moteur de recherche **Google** gère des dizaines voire des centaines de milliers de requêtes sécurisées par seconde, on comprend l'importance d'assurer l'efficacité de ces calculs.

Mes travaux de recherche portent principalement sur l'arithmétique pour la cryptographie. Cela couvre aussi bien les aspects algorithmiques de la multiplication de points (ou de manière équivalente de l'exponentiation) que l'arithmétique de la courbe elle-même ou encore l'arithmétique du corps sous-jacent et l'implantation logicielle de l'arithmétique des grands nombres.

Ce document présente une synthèse des différents travaux effectués dans ce domaine depuis ma soutenance de thèse en 2007 en tant que post-doctorant à l'université de Waterloo au Canada (2008-2010) puis en tant que maître de conférences à l'université de Toulon.

1. Introduction

1.1 Contexte

La cryptographie à clé publique repose sur le concept de fonction *à sens unique* : des fonctions dont les images sont faciles à calculer mais dont les antécédents sont calculatoirement difficiles à obtenir. L'exemple le plus familier est celui de la multiplication de deux nombres : il est facile d'effectuer un produit de deux nombres mais en général difficile d'obtenir la factorisation en facteurs premiers d'un entier. C'est à partir de ce problème qu'a été conçu le fameux protocole **RSA**.

Un second problème classique, proposé par Diffie et Hellman dès 1976, est celui du logarithme discret (PLD) dans un groupe fini. À l'origine le groupe considéré était le groupe $\mathbb{F}_p^\times$ (où p est un grand nombre premier) voire $\mathbb{F}_{2^n}^\times$. Il est apparu assez vite que la difficulté du PLD sur ces groupes, comme celle de la factorisation d'ailleurs, n'était pas exponentielle comme espéré mais sous-exponentielle.

Au milieu des années 80 Koblitz et Miller proposent alors d'utiliser le groupe des points d'une courbe elliptique définie sur un corps fini. Dans ces groupes, sous de bonnes conditions, aucun algorithme sous-exponentiel n'existe pour résoudre le PLD, permettant ainsi de manipuler des objets bien plus petits à niveau de sécurité équivalent et rendant ainsi les implantations logicielles et matérielles plus compactes et souvent plus rapides. C'est pour ces raisons qu'à partir des années 2000 les protocoles basés sur les courbes elliptiques se sont petit à petit imposés comme le standard des protocoles de chiffrement à clé publique. La cryptographie basée sur les courbes elliptiques (ECC pour *Elliptic Curve Cryptography*) est alors devenue un domaine de recherche extrêmement fructueux aussi bien sur des questions théoriques de sécurité que sur des aspects concrets d'efficacité des implantations logicielles et matérielles.

1.2 Quatre niveaux d'arithmétique pour la cryptographie basée sur les courbes elliptiques

Considérons $E(\mathbb{F}_p)$ le groupe des points $\mathbb{F}_p$-rationnels d'une courbe elliptique définie sur un corps fini. L'opération principale de la majorité des protocoles reposant sur les courbes elliptiques est la multiplication scalaire ou multiplication de point, c'est-à-dire le calcul de $k \cdot P = P + P \cdots + P$ où P est un point de la courbe et k un entier.

Le calcul de $k \cdot P$ se décompose naturellement en une *hiérarchie de quatre niveaux arithmétiques* imbriqués, chacun conditionnant le coût des niveaux supérieurs. C'est autour de ces quatre niveaux que s'organisent mes travaux de recherche.

Niveau 1 - Arithmétique des exposants. Le premier niveau concerne la représentation du scalaire k (ou de l'exposant selon que l'opération de groupe est considérée additivement ou multiplicativement). Calculer $k \cdot P$ revient à effectuer une suite d'additions et de doublements de points, dont la longueur dépend directement de la façon dont k est écrit. La méthode binaire classique requiert en moyenne $1{,}5 \log_2 k$ additions, mais des représentations alternatives (bases multiples, formes NAF et leurs généralisations, chaînes d'addition) permettent de réduire ce nombre. Plus généralement, ce niveau consiste en l'étude des *chaînes d'additions*.

Niveau 2 - Arithmétique des courbes. Le deuxième niveau porte sur le coût de chaque addition ou doublement de points. Ces opérations dépendent du choix des coordonnées (affines, projectives, co-Z) de la forme de la courbe (Weierstrass, Jacobi, Edwards etc.) et des formules algébriques utilisées. Lorsque la courbe admet un endomorphisme efficace ϕ (de type Frobenius sur les courbes de Koblitz, ou de type GLV), il est également possible de réduire les coûts de calcul, par exemple la multiplication scalaire $k \cdot P$ peut être décomposée sous la forme $k_1 P + k_2 \phi(P)$ avec $|k_1|, |k_2| \approx \sqrt{k}$, réduisant ainsi de moitié la longueur du calcul.

Niveau 3 - Arithmétique des corps finis. Chaque addition ou doublement de points se ramène à un certain nombre d'opérations dans le corps de base $\mathbb{F}_p$: multiplications, carrés, additions modulaires. Ce troisième niveau concerne les algorithmes spécifiques permettant de réaliser efficacement ces opérations élémentaires. On s'intéresse en particulier aux algorithmes de multiplications sur les grands nombres (Karatsuba, Toom-Cook, FFT) mais également aux différentes représentations des éléments du corps (représentation binaire, RNS, PMNS) et leurs compromis respectifs en termes d'efficacité, sécurité, niveau de parallélisme etc.

Niveau 4 - Arithmétique des ordinateurs. Enfin, si l'on se ramène aux standards en vigueur, les corps $\mathbb{F}_p$ utilisés nécessitent la manipulation d'entiers de 256 à 521 bits. Ces *grands entiers* sont typiquement découpés en mots de 64 bits, et les opérations

élémentaires sont reconstruites à partir des instructions disponibles sur le processeur. Ce niveau le plus bas conditionne la performance absolue de toute la chaîne.

Ces quatre niveaux forment une pile où chaque couche consomme les opérations fournies par celle du dessous. L'optimisation globale de la multiplication scalaire nécessite un travail coordonné sur chacun d'eux.

1.3 Résumé des principales contributions

Arithmétique des exposants (Chapitre 2).

— *Bases doubles pour la multiplication scalaire.* Le système à double base (Double Base Number System [DIM08]) permet de représenter un entier k sous la forme $k = \sum_i s_i 2^{b_i} 3^{t_i}$, avec un nombre de termes non nuls asymptotiquement en $O\left(\frac{\log k}{\log \log k}\right)$, inférieur à celui des représentations NAF classiques. Cependant, le calcul de telles expansions par l'algorithme glouton standard est nettement plus lent que les algorithmes de recodage usuels et le coût de calculs d'une exponentiation est en général supérieure à celle des algorithmes standards. Les doubles bases chaînées [DI06] permettent d'utiliser un schéma de type Horner mais induisent une augmentation du nombre de termes de la représentation. Dans [MH09 ; MH15], nous proposons une version généralisée de l'algorithme d'exponentiation de Yao adaptée aux expansions à double base, ainsi qu'une classe dite *contrainte* pour laquelle nous prouvons que la densité asymptotique reste en $O\left(\frac{\log k}{\log \log k}\right)$.

— *Représentation à chiffres aléatoires.* Les méthodes de recodage à fenêtre fractionnaire (*fractional wNAF*) reposent sur un ensemble de chiffres de la forme $\{1, 3, \ldots, 2n-1\}$, et il était connu que cet ensemble est optimal parmi tous les ensembles de n chiffres impairs en termes de densité moyenne. Dans [MH16], nous généralisons cette approche à tout ensemble de chiffres impairs contenant 1. Nous proposons un algorithme général de recodage et établissons une formule close permettant de calculer la densité moyenne de termes non nuls pour n'importe quel ensemble de chiffres, ce qui nous permet de caractériser les ensembles optimaux et de montrer qu'il en existe une infinité pour un nombre de chiffres donné. Nous proposons également un schéma d'exponentiation randomisé dans lequel l'ensemble de chiffres est tiré aléatoirement en début de calcul conservant une densité asymptotique pratiquement identique à celle de l'optimal.

Arithmétique des courbes (Chapitre 3).

— *Génération aléatoire de chaînes d'addition euclidiennes.* Une chaîne d'addition euclidienne (EAC) de longueur l calculant k permet d'effectuer la multiplica-

tion scalaire $k \cdot P$ en l additions différentielles de points, ce qui constitue une protection naturelle contre les attaques simples par canaux auxiliaires [Mél07]. L'inconvénient est que l'obtention de chaînes courtes calculant un entier donné k est difficile. Dans [Her+10], nous proposons deux algorithmes pour générer aléatoirement des EAC courtes et fournissons une première analyse théorique de leur longueur ainsi que de la distribution des clés associées. Nous introduisons également un schéma de multiplication scalaire à point de base fixe exploitant cette construction, alliant efficacité, faible empreinte mémoire et résistance SPA.

— *Extension aux courbes à endomorphisme efficace.* L'approche EAC est initialement limitée au cas du point de base fixe, car la génération d'une chaîne pour un entier k quelconque est coûteuse et ne contrôle pas la distribution des points obtenus. Dans [Dos+18], nous étendons cette méthode à la multiplication scalaire sur un point quelconque pour les courbes admettant un endomorphisme efficacement calculable (courbes de type GLV), en tirant parti de la décomposition $k \cdot P = k_1 P + k_2 \phi(P)$ pour travailler sur des scalaires deux fois plus courts. Nous généralisons les résultats sur la génération aléatoire de chaînes et la distribution des clés résultantes, proposons une implantation sur plusieurs plateformes et fournissons une étude expérimentale complète du coût pratique des multiplications modulaires en utilisant les bibliothèques standard Java et C.

Arithmétique des corps finis (Chapitre 4).

— *Multiplication dans $\mathbb{F}_{2^n}$ par recombinaison de blocs.* Fan et Hasan ont proposé une représentation de la multiplication dans un corps binaire comme produit matrice de Toeplitz-vecteur (TMVP), permettant de concevoir des multiplicateurs matériels à complexité spatiale sous-quadratique. Dans [Has+12 ; HMN11], nous proposons une extension de cette architecture fondée sur une décomposition en blocs du multiplicateur de Fan-Hasan suivie d'une recombinaison de ces blocs : en remplaçant un bloc de reconstruction par un bloc d'addition binaire de plus petite taille, on réduit la complexité spatiale tout en conservant la même complexité temporelle, surpassant ainsi les architectures sous-quadratiques existantes. Nous montrons en outre que cette technique de recombinaison s'applique à la conception d'une architecture efficace pour le calcul de la fonction GHASH utilisée dans le mode de chiffrement authentifié Galois/Counter Mode (GCM).

— *Réduction interne PMNS par algorithmes de réseau euclidien.* Le Système de Représentation Polynomiale Modulaire (PMNS) est une alternative à la représentation binaire multi-précision pour l'arithmétique dans les corps premiers : un élément de $\mathbb{Z}/p\mathbb{Z}$ est représenté par un polynôme à coefficients bornés, et l'opération critique est la réduction interne, étape de réduction de la taille des coefficients après chaque multiplication. Les travaux antérieurs réalisent cette

réduction via des algorithmes de type Montgomery ou Barrett adaptés aux polynômes. Dans [Mél22], je propose deux algorithmes alternatifs fondés sur les idées de Babai pour le problème du plus proche vecteur dans un réseau euclidien, et nous montrons qu'ils permettent de réduire significativement le nombre d'additions nécessaires à la réduction interne. En pratique, cela peut être vu comme une généralisation des approches précédentes, qui n'opéraient que sur un sous-réseau de celui considéré ici. Ces travaux incluent le développement d'un outil de génération automatique de code C produisant des implantations optimisées pour tout corps premier, ainsi qu'une analyse des compromis entre taille du PMNS et efficacité en fonction de la taille de p.

Arithmétique multi-précision (Chapitre 5).

— *PMNS creux et réduction CIOS.* L'efficacité de l'arithmétique PMNS dépend fortement du choix du premier p : sous certaines conditions, des nombres premiers engendrent des PMNS dits *creux* qui permettent une réduction interne de complexité linéaire, comparable à celle offerte par les premiers pseudo-Mersenne, mais la classe de ces premiers était jusqu'alors très restreinte. Dans [MPV25], nous adaptons l'algorithme de réduction de Montgomery CIOS au cadre du PMNS multi-précision en exploitant la structure de matrice de Toeplitz inhérente au système, permettant d'effectuer la réduction colonne par colonne et de traiter des premiers de grande taille. Dans [Dos+25], nous étendons la classe des premiers engendrant des PMNS creux à une famille beaucoup plus large de la forme $p = \frac{\alpha\gamma^k - 2}{\gamma - 1}$, et nous montrons que les performances obtenues sont compétitives avec, et dans certains cas supérieures à, l'arithmétique pseudo-Mersenne standard. Ces travaux incluent la contribution d'une base de données de courbes elliptiques construites sur ces premiers et satisfaisant le standard Brainpool.

Dans la suite de ce document j'illustrerai mes contributions dans chacun des différents niveaux d'arithmétiques à travers un article choisi.

— Le chapitre 2 est dédié à l'arithmétique des exposants à partir d'un travail sur la représentation de ceux-ci à partir de systèmes de chiffres aléatoires [MH16] ;
— Le chapitre 3 est consacré à l'arithmétique des courbes et l'exploitation des endomorphismes pour la génération d'exposants directement sous forme de chaîne d'addition euclidienne [Dos+18] ;
— Le chapitre 4 illustre mes travaux sur l'arithmétique des corps finis à travers un travail sur les PMNS et leur relation avec les algorithmes de Babai sur la recherche de plus proche vecteur dans un réseau [Mél22] ;
— Enfin le chapitre 5 descend au niveau l'arithmétique des ordinateurs via une

étude poussée sur l'implantation concrête des PMNS sur les architectures 64-bits [MPV23].

2. Arithmétique des exposants

La multiplication scalaire est l'opération centrale de la quasi-totalité des protocoles reposant sur les courbes elliptiques. Ce problème est souvent présenté sous sa forme multiplicative, à savoir calculer g^k où g est un élément d'un groupe fini G.

Posons $k = (k_{t-1} \dots k_0)_2$, l'algorithme classique *double-and-add* ou *square-and-multiply* garantit de pourvoir toujours calculer une exponentiation en au plus $2\log_2(k)$ opérations de groupes. Il existe de nombreux raffinements de cet algorithme, en général basés sur l'idée originale de Brauer [Bra39], exploitant une représentation en base 2^w.

Ce chapitre présente une approche développée dans [MH16] visant à généraliser la méthode des fenêtres fractionnaires (*fractional wNAF*) introduite par Möller [Möl04] à tout ensemble de chiffres contenant 1, établit une formule close pour la densité moyenne via un modèle probabiliste et propose un schéma d'exponentiation randomisé.

2.1 Représentation à chiffres aléatoires

2.1.1 Algorithme de recodage

Soit $\mathcal{D} = \{d_1, \dots, d_l\}$ un ensemble d'entiers naturels impairs. On pose $\overline{\mathcal{D}} = \mathcal{D} \cup \{-d_1, \dots, -d_l\}$. On appelle $\mathcal{D}$-*représentation* de k tout recodage de la forme $k = \sum_i k_i 2^i$ avec $k_i \in \overline{\mathcal{D}} \cup \{0\}$. On note $N(\mathcal{D})$ l'ensemble des entiers admettant une telle représentation.

Pour que $N(\mathcal{D}) = \mathbb{Z}$, une condition nécessaire est $\gcd(\mathcal{D}) = 1$ ne serait-ce que pour garantir que 1 soit représentable. Cela dit il n'est pas évident que cela soit une condition suffisante. En revanche, $1 \in \mathcal{D}$ est une condition suffisante : la représentation binaire usuelle de tout entier est une $\mathcal{D}$-représentation. On se restreint donc dans la suite aux ensembles de la forme $\mathcal{D} = \{1, d_2, \dots, d_n\}$.

Notations. Pour tout entier $w > 0$ et tout entier x, on pose $p_w(x) := x \bmod 2^w$. On définit $\mathcal{D}_w = p_w(\mathcal{D})$, $\overline{\mathcal{D}}_w = \mathcal{D}_w \cup \{2^w - d : d \in \mathcal{D}_w\}$ et $W_n = \lfloor \log_2(\max_i d_i) \rfloor$.

Pour tout entier impair k, on définit $w_{\max}(k)$ comme le plus grand entier $w \leqslant W_n + 2$

tel qu'il existe $d_i \in \mathcal{D}$ vérifiant simultanément $d_i < k$ et $p_w(k) \in \overline{\mathcal{D}}_w$. On définit alors l'application $\mathrm{digit}_{\mathcal{D}} : \mathbb{N} \to \overline{\mathcal{D}} \cup \{0\}$ par :

— si k est pair : $\mathrm{digit}_{\mathcal{D}}(k) = 0$;

— si k est impair, en posant $W_{\max} = w_{\max}(k)$:

  — si $p_{W_{\max}}(k) \in \mathcal{D}_{W_{\max}}$, alors $\mathrm{digit}_{\mathcal{D}}(k) = d$ pour tout $d < k$ tel que $p_{W_{\max}}(k) = p_{W_{\max}}(d)$;

  — si $2^{W_{\max}} - p_{W_{\max}}(k) \in \mathcal{D}_{W_{\max}}$, alors $\mathrm{digit}_{\mathcal{D}}(k) = -d$ pour tout $d < k$ tel que $p_{W_{\max}}(k) = 2^{W_{\max}} - p_{W_{\max}}(d)$.

Cette application est bien définie : comme $1 \in \mathcal{D}$, on a $1 \in \mathcal{D}_w$ pour tout w, ce qui garantit $W_{\max} \geqslant 2$.

L'algorithme suivant utilise cette application pour calculer la $\mathcal{D}$-représentation d'un entier quelconque.

Algorithm 1 Représentation à chiffres aléatoires (RDR) de k

Require: Un entier k et un ensemble $\mathcal{D} = \{1, d_2, \ldots, d_n\}$
Ensure: $(k_{t-1} \ldots k_0)$ tel que $k = \sum_i k_i 2^i$, $k_i \in \overline{\mathcal{D}} \cup \{0\}$

```
i ← 0
while k ≠ 0 do
    k_i ← digit_D(k)
    k ← (k − k_i)/2
    i ← i + 1
end while
return (k_{i−1} ... k_0)
```

Remarque 2.1. *Lorsque $\mathcal{D} = \{1, 3, \ldots, 2n-1\}$, l'Algorithme 1 produit exactement le recodage* fractional windows *de Möller.*

Exemple 2.2. *Soit $k = 31415$ et $\mathcal{D} = \{1, 3, 23, 27\}$, de sorte que $W_n = 4$, $\mathcal{D}_2 = \{1, 3\}$, $\mathcal{D}_3 = \{1, 3, 7\}$, $\mathcal{D}_4 = \{1, 3, 7, 11\}$ et $\mathcal{D}_5 = \mathcal{D}_6 = \mathcal{D}$. L'algorithme produit les étapes suivantes :*

1. *k est impair, $k \bmod 2^6 \equiv 55 \equiv -19$ et $k \bmod 2^5 \equiv 23$, donc $W_{\max} = 5$ et $k_0 = 23$;*
2. *$k \leftarrow (k-23)/2 = 15696$; $k_1 = k_2 = k_3 = k_4 = 0$ et $k \leftarrow k/2^4 = 981$;*
3. *$k \bmod 2^6 \equiv 21$, $k \bmod 2^5 \equiv 21 \equiv -11$, $k \bmod 2^4 \equiv 5 \equiv -11$, donc $W_{\max} = 4$ et $k_5 = -27$ (car $p_4(27) = 11$) ;*
4. *$k \leftarrow (k+27)/2 = 504$; $k_6 = k_7 = k_8 = 0$ et $k \leftarrow k/8 = 63$;*
5. *$k \bmod 2^6 \equiv -1$, donc $W_{\max} = 6$ et $k_9 = -1$;*
6. *$k \leftarrow (k+1)/2 = 32$; $k_{10} = \cdots = k_{14} = 0$ et $k = k_{15} = 1$.*

On obtient finalement $k = (1, 0, 0, 0, 0, 0, -1, 0, 0, 0, -27, 0, 0, 0, 0, 23)_2$*, représentation à* 4 *chiffres non nuls sur* 16 *positions.*

2.1.2 Densité moyenne

Théorème 2.3 (Densité moyenne de la RDR, [MH16, Thm. 1])**.** *Soit* $\mathcal{D} = \{1, d_2, \ldots, d_n\}$ *un ensemble de chiffres. Pour tout* $w \geqslant 2$*, on pose* $D(w) = \#\overline{\mathcal{D}}_w / 2^{w-1}$*. La densité asymptotique moyenne de termes non nuls de la RDR est* $\frac{1}{a_{\mathcal{D}}+1}$*, où*

$$a_{\mathcal{D}} = 2\, D(W_n + 2) + \sum_{w=2}^{W_n+1} D(w).$$

Démonstration. Soit k un entier impair supérieur à d_n et posons $d = \mathrm{digit}_{\mathcal{D}}(k)$. Évaluons d'abord la probabilité $P(w)$ que $W_{\max} = w$ pour chaque $w \leqslant W_n + 2$.

Par définition, $D(w)$ est la probabilité que le résidu d'un entier impair quelconque modulo 2^w appartienne à $\overline{\mathcal{D}}_w$. On a donc $P(W_n + 2) = D(W_n + 2)$. Pour $w < W_n + 2$, on remarque que

$$(d \bmod 2^{w+1} \in \overline{\mathcal{D}}_{w+1}) \;\Rightarrow\; (d \bmod 2^w \in \overline{\mathcal{D}}_w),$$

ce qui donne $P(w) = D(w) - D(w+1)$ pour $w < W_n + 2$.

Par définition de $W_{\max}$, on a $(k - d) \equiv 0 \pmod{2^{W_{\max}}}$, c'est-à-dire que les $W_{\max}$ bits de poids faible de k coïncident avec ceux de d. Si $W_{\max} < W_n + 2$, le bit suivant $k'_{W_{\max}}$ est non nul ; si $W_{\max} = W_n + 2$, le nombre moyen de zéros consécutifs suivant ce bit vaut 1 pour une séquence de bits aléatoires. Le nombre moyen $a_{\mathcal{D}}$ de zéros suivant un chiffre non nul dans la RDR se calcule donc par :

$$\begin{aligned} a_{\mathcal{D}} &= (W_n + 2)\, P(W_n + 2) + W_n\, P(W_n + 1) + \cdots + 2\, P(3) + P(2) \\ &= (W_n + 2)\, D(W_n + 2) + W_n\, (D(W_n + 1) - D(W_n + 2)) + \cdots + (D(2) - D(3)) \\ &= 2\, D(W_n + 2) + D(W_n + 1) + \cdots + D(3) + D(2). \end{aligned}$$

□

Exemple 2.4. *Pour* $B_n = \{1, 3, \ldots, 2n-1\}$*, on a* $W_n = \lfloor \log_2(2n-1) \rfloor$*. D'une part,* $D(w) = 1$ *pour* $2 \leqslant w \leqslant W_n + 1$ *; d'autre part,* $D(W_n + 2) = 2n/2^{W_n+1}$*, ce qui donne*

$$a_{\mathcal{D}} = W_n + \frac{2n}{2^{W_n}},$$

retrouvant le résultat classique sur la densité du frac-wNAF.

Exemple 2.5. *Pour* $\mathcal{D} = \{1, 3, 23, 27\}$*, on vérifie que* $\overline{\mathcal{D}}_2 = \{1, 3\}$*,* $\overline{\mathcal{D}}_3 = \{1, 3, 5, 7\}$*,*

$\overline{\mathcal{D}}_4 = \{1,3,5,7,9,11,13,15\}$, $\overline{\mathcal{D}}_5 = \{1,3,5,9,23,27,29,31\}$. *Le théorème donne*

$$a_{\mathcal{D}} = 2 \times \tfrac{1}{4} + \tfrac{1}{2} + 1 + 1 + 1 = 4,$$

soit une densité de $\frac{1}{5}$*, identique à celle du 4-NAF avec* $\mathcal{D} = \{1,3,5,7\}$*. Cette densité optimale peut donc être atteinte par deux ensembles différents de même cardinal.*

2.1.3 Ensembles optimaux

La représentation RDR généralise le *frac-wNAF*. On montre maintenant que la densité minimale est atteinte par l'ensemble $\{1,3,\ldots,2n-1\}$, mais que cette borne est également atteinte par une infinité d'autres ensembles.

Définition 2.6. *Soit* $n > 0$ *et* $\mathbb{D}_n$ *l'ensemble de tous les ensembles de* n *entiers impairs contenant* 1*. Un ensemble* $\mathcal{D} \in \mathbb{D}_n$ *est dit* optimal *si*

$$a_{\mathcal{D}} = \max_{\mathcal{D}' \in \mathbb{D}_n} a_{\mathcal{D}'}.$$

Théorème 2.7 ([MH16, Thm. 2])**.** *Soit* $n > 0$*,* $w_n = \lfloor \log_2 n \rfloor$ *et* $\mathcal{D} \in \mathbb{D}_n$*. Si* $\mathcal{D}$ *est optimal, alors*

$$a_{\mathcal{D}} = w_n + \frac{n}{2^{w_n}} + 1.$$

Démonstration. À partir du Théorème 2.3, on décompose $a_{\mathcal{D}}$ en trois parties :

$$a_{\mathcal{D}} = \underbrace{2\,D(W_n+2)}_{A} + \underbrace{\sum_{i=w_n+3}^{W_n+1} D(i)}_{B} + \underbrace{\sum_{i=2}^{w_n+2} D(i)}_{C}.$$

Obervons d'abord que $\forall\, w \geqslant 2$, $\#\overline{\mathcal{D}}_w \leqslant \min(2^{w-1}, 2n)$. Par définition de w_n, on a $2n < 2^{w_n+2}$, donc $\#\overline{\mathcal{D}}_w \leqslant 2n$ pour tout $w \geqslant w_n + 3$. On en déduit les bornes :

$$A = \frac{2 \cdot 2n}{2^{W_n+1}} = \frac{n}{2^{W_n-1}},$$

$$B \leqslant \sum_{i=w_n+3}^{W_n+1} \frac{2n}{2^{i-1}} \leqslant \frac{4n}{2^{w_n+2}} \left(1 - \frac{1}{2^{W_n-w_n-1}}\right) \leqslant \frac{n}{2^{w_n}} - \frac{n}{2^{W_n-1}},$$

$$C \leqslant \sum_{i=2}^{w_n+2} 1 = w_n + 1.$$

On obtient ainsi $a_{\mathcal{D}} \leqslant w_n + n/2^{w_n} + 1$. L'égalité est atteinte pour $\mathcal{D} = \{1,3,\ldots,2n-1\}$. En effet, si n est une puissance de deux, alors $W_n = w_n$ et $a_{\mathcal{D}} = w_n + 2 = w_n + n/2^{w_n} + 1$.

Sinon, $W_n = w_n + 1$, $D(2) = \cdots = D(W_n + 1) = 1$ et $2D(W_n + 2) = 4n/2^{W_n+1}$, ce qui donne $a_{\mathcal{D}} = W_n + n/2^{W_n-1} = w_n + 1 + n/2^{w_n}$, confirmant l'égalité. □

Corollaire 2.8 ([MH16, Cor. 1]). *Soit $n > 0$, $w_n = \lfloor \log_2 n \rfloor$ et $\mathcal{D} \in \mathbb{D}_n$. Alors $\mathcal{D}$ est optimal si et seulement si*

$$\#\overline{\mathcal{D}}_{w_n+3} = 2n \quad \text{et} \quad \#\overline{\mathcal{D}}_{w_n+2} = 2^{w_n+1}.$$

Démonstration. D'après la preuve du Théorème 2.7, $\mathcal{D}$ est optimal si et seulement si, d'une part, $\#\overline{\mathcal{D}}_i = 2n$ pour tout $i \geqslant w_n + 3$, et d'autre part $D(i) = 1$, c'est-à-dire $\#\overline{\mathcal{D}}_i = 2^{i-1}$, pour tout $2 \leqslant i \leqslant w_n + 2$.

Les deux implications se déduisent de l'inégalité $\#\overline{\mathcal{D}}_{w+1} \leqslant 2\,\#\overline{\mathcal{D}}_w$, vraie car tout élément de $\overline{\mathcal{D}}_{w+1}$ s'écrit $d' + b\,2^w$ avec $b \in \{0, 1\}$ et $d' \in \overline{\mathcal{D}}_w$. Elle implique que $(D(w) = 1) \Rightarrow (D(w-1) = 1)$, et aussi que pour tout $w \leqslant w_n + 3$,

$$2n \leqslant \#\overline{\mathcal{D}}_{w_n+3} \leqslant \#\overline{\mathcal{D}}_w \leqslant \min(2n, 2^w) \leqslant 2n,$$

ce qui conclut la preuve. □

Remarque 2.9. *Le Corollaire 2.8 implique directement qu'il existe une* infinité *d'ensembles optimaux. Par exemple, tous les ensembles de la forme*

$$\mathcal{D} = \{1, 3, \ldots, 2n-3\} \cup \{2n - 1 + 2^w\}, \quad w \geqslant w_n + 3,$$

sont optimaux. Plus généralement, pour tout $(t_2, \ldots, t_n) \in \mathbb{N}^{n-1}$, l'ensemble $\{1,\ 3 + t_2\,2^{w_n+3},\ \ldots,\ 2n - 1 + t_n\,2^{w_n+3}\}$ est optimal.

Exemple 2.10. *L'ensemble $\mathcal{D} = \{1, 3, 23, 27\}$ est optimal : avec $n = 4$ et $w_n = 2$, on vérifie $\#\overline{\mathcal{D}}_4 = \#\{1, 3, 5, 7, 9, 11, 13, 15\} = 8 = 2^3$ et $\#\overline{\mathcal{D}}_5 = \#\{1, 3, 5, 9, 23, 27, 29, 31\} = 8 = 2n$.*

2.2 Schéma d'exponentiation randomisé

L'existence d'une infinité d'ensembles optimaux ouvre la voie à un schéma d'exponentiation dont l'ensemble de chiffres est tiré aléatoirement en début de calcul. Soient g un élément de groupe, k un exposant, et m, l deux entiers avec $l \leqslant (m+1)/2$. On calcule g^k comme suit :

1. choisir aléatoirement $l - 1$ entiers impairs $\{d_2, \ldots, d_l\}$ dans $\{3, \ldots, m\}$;
2. calculer la RDR de k via l'Algorithme 1 avec $\mathcal{D} = \{1, d_2, \ldots, d_l\}$;
3. calculer g^k via l'Algorithme 2.

Algorithm 2 Calcul de g^k

Require: Un entier $k = (k_{l-1} \dots k_0)_2$, un élément g, un ensemble $\mathcal{D}$
Ensure: g^k

```
h ← 1
for d ∈ D do
    G_d ← g^d
end for
for i = l − 1 à 0 do
    h ← h^2
    if k_i ≠ 0 then
        h ← h × G_{k_i}
    end if
end for
return h
```

2.2.1 Densité moyenne et problème d'urne

Le Théorème 2.3 permet de calculer la densité de la RDR pour un ensemble $\mathcal{D}$ fixé. Une question naturelle est le comportement moyen de cette densité lorsque les chiffres sont tirés aléatoirement dans $B_m = \{1, 3, \dots, m\}$.

Pour appliquer le théorème, il faut évaluer $D(w) = \#\overline{\mathcal{D}}_w / 2^{w-1}$, c'est-à-dire compter le nombre de classes d'équivalence de $\mathcal{D}$ par rapport à la relation $\mathcal{R}_w$ définie pour $x, y \in B_m$ par :

$$x \,\mathcal{R}_w\, y \iff x \equiv y \pmod{2^w} \text{ ou } x \equiv -y \pmod{2^w}.$$

Ce problème se prête naturellement à une modélisation de type tirage d'urne. Soient $N = (m+1)/2$ boules (une par élément de B_m) et $C_w = 2^{w-2}$ couleurs (une par classe de $\mathcal{R}_w$). Chaque couleur c contient E_w^c boules. Le problème revient à tirer l boules sans remise et à estimer le nombre moyen de couleurs distinctes obtenues.

Soit $M(l, c, N)$ le nombre de tirages de l boules parmi N donnant exactement c couleurs. Un résultat de Walton montre que ce nombre se lit sur le polynôme générateur

$$F_w(X, Y) = \prod_{c=1}^{C_w} \Big(Y \big\{ (1+X)^{E_w^c} - 1 \big\} + 1 \Big),$$

via $F_w(X, Y) = \sum_{c,l} M(l, c, N)\, X^l Y^c$. On peut alors calculer $D(w)$ pour des valeurs pratiques de w.

La Table 2.1 résume les résultats obtenus pour $w \leqslant 10$, en fixant $l = \lfloor (m+1)/4 \rfloor$ pour maximiser le nombre d'ensembles possibles. La colonne « wNAF » indique la densité optimale du *frac-wNAF* utilisant autant de chiffres que la RDR.

m	$\#\mathcal{D}$	RDR	wNAF
7	2	3,833	4
15	4	4,771	5
31	8	5,728	6
63	16	6,706	7
127	32	7,695	8
255	64	8,689	9
511	128	9,686	10
1023	256	10,69	11

TABLE 2.1 – Inverse de la densité de la RDR et du wNAF, avec $\lfloor (m+1)/4 \rfloor$ chiffres.

L'écart entre la RDR aléatoire et le *frac-wNAF* optimal est systématiquement inférieur à un demi-bit, confirmant que les ensembles tirés aléatoirement atteignent une densité quasi-optimale.

2.2.2 Schéma de précalcul

La première étape du schéma d'exponentiation consiste à calculer les g^{d_i} pour $d_i \in \mathcal{D}$. L'approche naïve, qui suit la chaîne $(1, 2, 3, 5, 7, \ldots, m)$ en ne conservant que les éléments nécessaires, requiert $m/2$ opérations. Nous proposons une méthode plus efficace inspirée de l'algorithme de Pippenger, tirant parti du fait que tous les d_i sont impairs et petits.

Soit $0 < b < W$ un paramètre et $q = \lfloor m/2^b \rfloor$. On procède en trois étapes :

1. calculer $X = \{g, g^2, g^3, g^5, g^7, \ldots, g^{2^b-1}\}$ (2^{b-1} opérations) ;
2. calculer $Y = \{g^{2^b}, g^{2 \cdot 2^b}, \ldots, g^{q \cdot 2^b}\}$ (q opérations) ;
3. pour chaque $d_i \notin X$, obtenir $g^{d_i} = x \cdot y$ avec $(x, y) \in X \times Y$ ($\#\mathcal{D}$ opérations au plus).

Le coût total est borné par $2^{b-1} + \lfloor m/2^b \rfloor + \#\mathcal{D}$ opérations de groupe. La proportion de chiffres d_i appartenant déjà à X est $2^b/(m+1)$; avec $\lfloor (m+1)/4 \rfloor$ chiffres tirés aléatoirement, la méthode économise en moyenne 2^{b-2} opérations par rapport à l'approche naïve.

La Table 2.2 compare le coût total (en carrés S et multiplications M) pour des exposants de 1024 bits, en distinguant la RDR aléatoire, la RDR restreinte aux ensembles optimaux (Opt. RDR), et le *frac-wNAF*.

Taille de $\mathcal{D}$	Méthode	$a_{\mathcal{D}}+1$	Nombre d'opérations de groupe
8	RDR	5,701	$1023S+191M$
	Opt. RDR	5,970	$1024S+183M$
	frac-wNAF	5,997	$1023S+178M$
16	RDR	6,666	$1023S+175M$
	Opt. RDR	6,960	$1023S+169M$
	frac-wNAF	6,962	$1022S+161M$
24	RDR	7,209	$1023S+175M$
	Opt. RDR	7,454	$1023S+167M$
	frac-wNAF	7,454	$1023S+160M$
32	RDR	7,634	$1023S+175M$
	Opt. RDR	7,940	$1023S+170M$
	frac-wNAF	7,950	$1022S+160M$
48	RDR	8,178	$1023S+190M$
	Opt. RDR	8,434	$1023S+180M$
	frac-wNAF	8,440	$1022S+168M$
64	RDR	8,692	$1023S+207M$
	Opt. RDR	8,922	$1023S+196M$
	frac-wNAF	8,940	$1022S+177M$

TABLE 2.2 – Densité et nombre d'opérations de groupe pour la RDR et le *frac-wNAF*, sur des exposants de 1024 bits.

Le surcoût de la randomisation est négligeable : pour $n=8$, la RDR aléatoire requiert $13M$ de plus que le *frac-wNAF*, soit moins de 7% sur les multiplications. Restreindre aux ensembles optimaux réduit l'écart à $5M$.

3. Arithmétique des courbes elliptiques

Après l'arithmétique des exposants, le niveau suivant concerne l'arithmétique du groupe dans lequel sont réalisées les opérations. Dans le cas des courbes elliptiques l'arithmétique y est particulièrement riche et offre la possibilité de nombreuses méthodes alternatives aux algorithmes d'exponentiation classique. L'exploitation conjointe de formules de composition spécifiques, de structures algébriques spéciales et d'algorithmes d'exponentiation adaptés permet de mettre en oeuvre des méthodes originales, inenvisageables dans d'autres contextes.

Une de ces approches repose sur les *chaînes d'addition euclidiennes* (EAC) en combinaison avec l'arithmétique Co-Z. Sans entrer immédiatement dans les détails, une chaîne d'addition euclidienne peut être vue comme une suite d'additions permettant de construire un entier k à partir des valeurs initiales 1 et 2, sans recourir à l'opération de doublement. Par exemple $1, 2, 3, 5, 8, 13$ et $1, 2, 3, 5, 7, 9, 11, 13$ sont des EAC calculant 13. La première est considérée comme plus efficace que la seconde car plus courte. Il se trouve que ce type de chaînes d'additions est parfaitement adapté à l'arithmétique Co-Z sur les courbes elliptiques. Cette arithmétique permet des additions de points extrêmement efficaces (plus efficaces que les doublements) lorsque les deux opérandes partagent la même coordonnée Z et l'utilisation des EAC permet de maintenir cette propriété tout au long d'une multiplication de point. L'inconvénient en retour est qu'il est très difficile d'obtenir des EAC efficaces (c'est à dire courtes). Plus précisément, étant donné un entier k, il est difficile d'obtenir une EAC courte le calculant. Autrement dit, partant des points $(P, 2P)$, il est difficile d'obtenir un algorithme efficace calculant kP.

Pour contourner le problème, dans [Her+10], nous avons proposé des algorithmes de génération aléatoire d'EAC courtes : le scalaire est tiré aléatoirement parmi les entiers calculés par des chaînes de longueur n, et l'on garantit que des chaînes distinctes produisent des points distincts lorsque les points de départs sont $(F_{n+2}P, F_{n+3}P)$ où (F_i) est la suite de Fibonacci. Cependant, cette approche est limitée au cas d'un point

P fixe dans la mesure où le pré-calcul des deux points de départs est nécessaire pour que la méthode soit efficace.

Ce chapitre présente les travaux de [Dos+18], dans lesquels nous étendons la méthode EAC à la multiplication scalaire à partir d'un *point quelconque* pour les courbes admettant un endomorphisme efficacement calculable. L'idée centrale consiste à remplacer la paire initiale $(P, 2P)$ par la paire $(P, \phi(P))$: l'endomorphisme fournit gratuitement le second point de départ qui permet de garantir que les points calculés en fin de multiplication scalaire sont bien distintcs. Nous généralisons les résultats d'injectivité au départ depuis une paire (aP, bP) quelconque (Section 3.2), puis établissons les conditions sur l'ordre de la courbe garantissant l'injectivité depuis $(P, \phi(P))$ (Section 3.3). La dernière section présente une étude complète du coût théorique et pratique des implantations sur plateformes mobiles (Android) et x64.

3.1 Courbes à endomorphisme et arithmétique co-Z

3.1.1 Courbes à endomorphisme efficace et méthode GLV

Soit E une courbe elliptique définie sur $\mathbb{F}_p$ telle que $\#E(\mathbb{F}_p) = N \cdot h$ avec N un grand premier et h un petit cofacteur. On suppose que E admet un endomorphisme non trivial ϕ défini sur $\mathbb{F}_p$, de polynôme caractéristique $X^2 + rX + s$. On note λ l'unique élément de $[0, N-1]$ tel que $\phi(P) = \lambda P$ pour tout point P d'ordre N.

Pour tout scalaire $k \in [1, N-1]$, il est possible de trouver k_1, k_2 tels que $k \equiv k_1 + k_2\lambda \pmod N$ et $\max(|k_1|, |k_2|) \leqslant c\sqrt{N}$ pour une constante c explicite [SCQ03]. On peut alors calculer

$$k \cdot P = k_1 P + k_2 \phi(P)$$

en effectuant une double multiplication scalaire sur des scalaires de longueur $\ell/2$, où $\ell = \lceil \log_2 N \rceil$.

La méthode GLV standard (W-GLV) effectue $\ell/2$ doublements et en moyenne $\ell/4$ additions, mais nécessite de stocker P, $\phi(P)$, $P + \phi(P)$ et $P - \phi(P)$. Elle est vulnérable aux attaques simples par canaux auxiliaires (SSCA). La version sécurisée GLV-SAC [FLS15] effectue un doublement et une addition par bit scalaire grâce à une représentation signée à bits alternés (*Sign Aligned Coordinates*), ne stockant que $P + \phi(P)$, mais reste environ 60 % plus coûteuse que la méthode standard.

3.1.2 Arithmétique Co-Z

En coordonnées jacobiennes $(X : Y : Z)$, l'addition de deux points généraux coûte $11M + 5S$ (où M et S désignent le coût d'une multiplication et d'un carré dans $\mathbb{F}_p$). Ce coût descend à $5M + 2S$ lorsque les deux points partagent la même coordonnée Z, en utilisant l'opération ZADD [Mél07] :

$P_1 = (X_1, Y_1, Z), P_2 = (X_2, Y_2, Z)$ et $ZADD(P_1, P_2) = (X_3, Y_3, Z_3)$

$$A = (X_2 - X_1)^2, \quad B = X_1 A, \quad C = X_2 A, \quad D = (Y_2 - Y_1)^2$$

$$\begin{aligned} X_3 &= D - B - C \ , \\ Y_3 &= (Y_2 - Y_1)(B - X_3) - Y_1(C - B) \ , \\ Z_3 &= Z(X_2 - X_1) \ . \end{aligned}$$

L'astuce principale consiste à remarquer que lors du calcul de $P_1 + P_2$, les quantités $X_1 A$ et $Y_1(C - D)$ correspondent aux coordonnées du point P_1 avec pour nouvelle coordonnée Z : $Z_3 = Z(X_1 - X_2)$.

Opération ZADD. Étant donné un couple (P, Q) de points (ordonnés) partageant la même coordonnée Z, l'opération ZADD calcule, au choix, $(Q, P+Q)$ ou $(P, P+Q)$, où les points partagent à nouveau la même coordonnée Z. On appelle *petit pas* le calcul de $(P, Q) \to (P, P + Q)$ et *grand pas* le calcul de $(P, Q) \to (Q, P + Q)$.

Ainsi, partant de P et $2P$ exprimés avec la même coordonnée Z, on maintient l'invariant co-Z tout au long du calcul en alternant petits et grands pas.

3.1.3 Chaînes d'addition euclidiennes et multiplication sur point fixe

Définition 3.1. *Une* chaîne d'addition euclidienne *(EAC) de longueur s est une suite finie $c = (c_i)_{i=1,\ldots,s}$ d'éléments de $\{0, 1\}$. On note $\mathcal{M}$ l'ensemble de toutes les EAC et $\mathcal{M}_s$ l'ensemble des EAC de longueur s. À toute EAC $c \in \mathcal{M}_s$, on associe une suite $(v_i, u_i)_{i=0,\ldots,s}$ de $\mathbb{N}^2$ définie par :*

- $(v_0, u_0) = (1, 2)$,
- *pour $c_i = 1$ (petit pas) :* $(v_i, u_i) = (v_{i-1},\ v_{i-1} + u_{i-1})$,
- *pour $c_i = 0$ (grand pas) :* $(v_i, u_i) = (u_{i-1},\ v_{i-1} + u_{i-1})$.

On note $\chi(c) = v_s + u_s$ l'entier calculé par la chaîne et $\psi(c) = (v_s, u_s)$.

Étant donnée un entier k, il est très facile d'obtenir une EAC le calculant. Il suffit pour

cela de prendre un entier g premier avec k et d'appliquer l'algorithme d'Euclide dans sa version soustractive. Par exemple, on obtient les deux chaînes données au début de ce chapitre en appliquant l'algorithme d'Euclide à 13 et 8 pour la première et 13 et 11 pour la seconde :

$$\begin{aligned} 13 &= 8+5 \\ 8 &= 5+3 \\ 5 &= 3+2 \\ 3 &= 2+1 \end{aligned}$$

donne la chaîne

$$(1,2) \to (2,3) \to (3,5) \to (5,8)$$

représentée sous la forme

$$c = (0,0,0,0)$$

vérifiant $\chi(c) = 5 + 8 = 13$.

$$\begin{aligned} 13 &= 11+2 \\ 11 &= 9+2 \\ 9 &= 7+2 \\ 7 &= 5+2 \\ 5 &= 3+2 \\ 3 &= 2+1 \end{aligned}$$

donne la chaîne

$$(1,2) \to (2,3) \to (2,5) \to (2,7) \to (2,9) \to (2,11)$$

représentée sous la forme

$$c = (0,1,1,1,1)$$

vérifiant $\chi(c) = 2 + 11 = 13$.

Algorithm 3 Multiplication scalaire par EAC (point fixe)

Require: P et $2P$ (Co-Z), une EAC $c = (c_1, \ldots, c_s)$
Ensure: $Q = \chi(c) \cdot P$

```
(U1, U2) ← (P, 2P)
for i = 1 ... s do
    if c_i = 0 then
        (U1, U2) ← ZADD(U2, U1)        ▷ grand pas
    else
        (U1, U2) ← ZADD(U1, U2)        ▷ petit pas
    end if
end for
return U1 + U2
```

L'algorithme 3 permet, à partir d'une EAC d'effectuer une multiplication scalaire très simplement : chaque itération effectue exactement une opération ZADD quelle que

soit la valeur de c_i et l'efficacité dépend directement de la longueur de la chaîne. Cependant, pour un entier k donné, il est difficile de trouver une chaîne courte le calculant : cela revient à tester tous les entiers g premiers avec k jusqu'à trouver une chaîne satisfaisante.

Une idée naturelle consiste à générer directement des chaînes d'additions d'une taille fixée au lieu de nombres entiers. Il se trouve que, même restreinte à des chaînes de taille fixe, l'application χ n'est pas injective et la distribution des entiers produits n'est de plus pas uniforme. Autrement dit, on ne peut pas générer des points kP à partir d'une EAC tirée aléatoirement.

Dans [Her+10], nous avons montré que les 2^n chaînes de $\mathcal{M}_n$ calculent 2^n points distincts en partant de $(F_{n+2}P, F_{n+3}P)$ au lieu de $P, 2P$, où (F_n) est la suite de Fibonacci. Ce résultat permet un schéma de multiplication scalaire randomisé sécurisé et efficace dans le cas où le point P est fixe dans la mesure où le couple $(F_{n+2}P, F_{n+3}P)$ peut être pré-calculé. L'enjeu de ce chapitre est d'étendre cette propriété au cas d'un point P variable.

3.2 Injectivité depuis une paire (aP, bP) quelconque

Afin de généraliser l'approche précédente, on étend tout d'abord les fonctions χ et ψ à une paire initiale $(a, b) \in \mathbb{N}^2$ quelconque.

Définition 3.2. *Soient $(a, b) \in \mathbb{N}^2$, $s \in \mathbb{N}^*$ et $c \in \mathcal{M}_s$. On note $S_0 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ et $S_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$ les matrices correspondant respectivement au grand pas et au petit pas. On définit*

$$\psi_{a,b}(c) = (a,\ b) \prod_{i=1}^{s} S_{c_i}, \qquad \chi_{a,b}(c) = (a,\ b) \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Le cas $(a, b) = (1, 2)$ redonne les définitions précédentes : $\psi_{1,2}(c) = \psi(c)$ et $\chi_{1,2}(c) = \chi(c)$. Pour un point P, l'entier $\chi_{a,b}(c)$ est tel que $\chi_{a,b}(c) \cdot P$ est le point obtenu en appliquant l'Algorithme 3 depuis (aP, bP) plutôt que $(P, 2P)$.

Proposition 3.3 ([Dos+18, Prop. 1])**.** *Soit $s \in \mathbb{N}^*$. L'application $\mu : \mathcal{M}_s \to \mathbb{N}^2$ définie par $\mu(c) = \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ est injective.*

Proposition 3.4 ([Dos+18, Prop. 2])**.** *Soient n, a et b trois entiers positifs tels que $\gcd(a, b) = 1$ et $a > F_{n+2}$ ou $b > F_{n+2}$. Alors, pour tout $(c, c') \in \mathcal{M}_n^2$, on a $\chi_{a,b}(c) = \chi_{a,b}(c')$ si et seulement si $c = c'$.*

Corollaire 3.5 ([Dos+18, Cor. 1]). *Soit E une courbe elliptique, $P \in E$ un point d'ordre N, et n, a, b trois entiers positifs vérifiant :*

— $\gcd(a, b) = 1$ *et* ($a > F_{n+2}$ *ou* $b > F_{n+2}$),

— $aF_{n+1} + bF_{n+2} < N$ *et* $aF_{n+2} + bF_{n+1} < N$.

Alors les 2^n chaînes de $\mathcal{M}_n$ calculent 2^n points distincts de E en appliquant l'Algorithme 3 depuis (aP, bP).

Exemple 3.6. *Le cas $a = F_{n+2}$, $b = F_{n+3}$ redonne le résultat du point fixe de [Her+10] : les 2^n chaînes de $\mathcal{M}_n$ calculent 2^n points distincts depuis $(F_{n+2}P, F_{n+3}P)$.*

3.3 Extension à la multiplication scalaire d'un point variable

3.3.1 Résultat principal d'injectivité

Soit E une courbe elliptique d'ordre $N \cdot h$ admettant un endomorphisme efficacement calculable ϕ de polynôme caractéristique $X^2 + rX + s$. Pour un point P quelconque on a $\phi(P) = \lambda P$, on peut poser $b = \lambda$: le point $bP = \phi(P)$ est alors calculé par une simple évaluation de ϕ, sans aucune multiplication scalaire. Il reste à vérifier que les hypothèses du Corollaire 3.5 sont satisfaites pour $a = 1$ et $b = \lambda$.

La proposition suivante donne une borne inférieure sur les composantes de la décomposition.

Proposition 3.7 ([Dos+18, Prop. 3]). *Soient $(k_1, k_2) \in \mathbb{Z}^2 \setminus \{(0, 0)\}$. Si $k_1 + k_2\lambda \equiv 0 \pmod N$, alors*

$$\max(|k_1|, |k_2|) \geqslant \sqrt{\frac{N}{1 + |r| + s}}.$$

Proposition 3.8 ([Dos+18, Prop. 4]). *Si $N > F_{n+2}^2(1 + |r| + s)$, alors les 2^n chaînes de $\mathcal{M}_n$ calculent 2^n points distincts de E lorsqu'on applique l'Algorithme 4 depuis $(P, \phi(P))$.*

Démonstration. Soient $c, c' \in \mathcal{M}_n$ deux chaînes donnant le même point depuis $(P, \phi(P))$. Posons $\begin{pmatrix} k_1 \\ k_2 \end{pmatrix} = \mu(c)$ et $\begin{pmatrix} k'_1 \\ k'_2 \end{pmatrix} = \mu(c')$. On a $(k_1 - k'_1) + (k_2 - k'_2)\lambda \equiv 0 \pmod N$. Si $(k_1, k_2) \neq (k'_1, k'_2)$, la Proposition 3.7 donne $\max(|k_1 - k'_1|, |k_2 - k'_2|) \geqslant \sqrt{N/(1 + |r| + s)}$. Or les composantes de $\mu(c)$ sont bornées par F_{n+2}, donc $|k_i - k'_i| < F_{n+2}$ pour $i \in \{1, 2\}$, ce qui contredit la borne inférieure dès que $N > F_{n+2}^2(1 + |r| + s)$. On obtient donc $\mu(c) = \mu(c')$, et la Proposition 3.3 conclut $c = c'$. □

Remarque 3.9. *La condition* $N > F_{n+2}^2(1+|r|+s)$ *impose que la taille du corps soit supérieure au minimum requis par le niveau de sécurité cible. Comme* $F_n \approx \gamma^n/\sqrt{5}$ *avec* $\gamma = (1+\sqrt{5})/2$*, on doit travailler sur un corps de taille approximativement* $2n\log_2\gamma \approx 1{,}39n$ *bits. Le Tableau 3.1 résume les tailles requises pour les niveaux de sécurité standards, en prenant pour exemple* $(r,s) = (0,1)$*.*

Niveau de sécurité (bits)	Taille du corps (bits)
96	269
128	358
192	536

TABLE 3.1 – Taille du corps $\mathbb{F}_p$ requise pour garantir l'injectivité de la méthode EAC, pour un endomorphisme ϕ satisfaisant $\phi^2 + s = 0$ avec $s = 1$.

3.3.2 Algorithme de multiplication scalaire

L'Algorithme 4 décrit la méthode de multiplication scalaire résultante. Il prend en entrée un point P et une EAC c choisie aléatoirement, calcule $\phi(P)$ directement par évaluation de l'endomorphisme, puis applique la procédure ZADDb itérativement.

Algorithm 4 PointFromEAC(c) : multiplication scalaire sur courbe à endomorphisme efficace

Require: $P = (X, Y, 1)$ et $Q = \phi(P) = (X', Y', 1)$, une EAC $c = (c_1, \ldots, c_l)$
Ensure: Le point $\chi_{1,\lambda}(c) \cdot P$
1: **for** $i = 1 \ldots l$ **do**
2: ZADDb(c_i) ▷ petit pas si $c_i = 1$, grand pas si $c_i = 0$
3: **end for**
4: ZADD(1) ▷ dernière addition différentielle
5: **return** Q

Le schéma global fonctionne comme suit : pour obtenir un scalaire de n bits de sécurité, on génère aléatoirement une EAC $c \in \mathcal{M}_n$ puis on calcule $Q = \chi_{1,\lambda}(c) \cdot P$ via l'Algorithme 4. La Proposition 3.8 garantit que les 2^n chaînes possibles produisent 2^n points distincts, assurant ainsi n bits de sécurité. Seuls les points P et $\phi(P)$ doivent être stockés pour démarrer le calcul, ce qui constitue un avantage mémoire significatif.

3.4 Coût théorique et performances pratiques

3.4.1 Analyse théorique

Soit M_t (resp. S_t) le coût d'une multiplication (resp. d'un carré) dans $\mathbb{F}_p$ pour un premier de t bits. Une EAC de longueur ℓ requiert ℓ opérations ZADDb, chacune de coût $5M_t + 2S_t$ en coordonnées de Weierstrass, plus une addition finale ZADD, soit un coût total de $(\ell+1)(5M_t + 2S_t)$. Pour atteindre un niveau de sécurité de $\ell/2$ bits, la longueur t doit être choisie voisine de $1{,}4\ell$ (d'après la Proposition 3.8).

On compare dans le Tableau 3.2 ce coût à celui de la méthode GLV standard (W-GLV) et de sa version sécurisée (W-GLV-SAC).

Méthode	Opérations de corps
EAC	$(\ell+1)(5M_t + 2S_t)$
W-GLV	$\ell/2 \times (5{,}5M_t + 7S_t)$
W-GLV-SAC	$\ell/2 \times (9M_t + 9S_t)$

TABLE 3.2 – Coût théorique des méthodes de multiplication scalaire pour un niveau de sécurité de $\ell/2$ bits ($t \simeq 1{,}4\ell$, modèle de Weierstrass).

Le coût de la méthode EAC dépend de M_t, le coût d'une multiplication sur des entiers de taille $t \simeq 1{,}4\ell$, tandis que les méthodes GLV opèrent sur le corps $\mathbb{F}_p$ de taille ℓ. Notre méthode est théoriquement compétitive avec W-GLV-SAC dès lors que

$$M_t < 1{,}29\, M_{2\ell'},$$

où $\ell' = \ell/2$. Cette condition dépend du comportement de l'algorithme de multiplication utilisé en pratique. Le Tableau 3.3 donne les coûts numériques pour les niveaux de sécurité standards.

Méthode	96 bits	128 bits	192 bits
EAC	$965M_{269} + 386S_{269}$	$1285M_{358} + 514S_{358}$	$1925M_{536} + 770S_{536}$
W-GLV	$528M_{192} + 672S_{192}$	$704M_{256} + 896S_{256}$	$1056M_{384} + 1344S_{384}$
W-GLV-SAC	$864M_{192} + 864S_{192}$	$1152M_{256} + 1152S_{256}$	$1728M_{384} + 1728S_{384}$

TABLE 3.3 – Coût théorique de la multiplication scalaire (forme de Weierstrass) pour différents niveaux de sécurité.

3.4.2 Coût pratique de la multiplication modulaire

L'argument théorique repose sur la condition $M_t < 1{,}29\, M_{2\ell'}$. Pour la vérifier en pratique, nous avons mesuré le rapport $\tilde{M}_t/\tilde{M}_{1,4t}$ entre le coût d'une multiplication modulaire pour des entiers de t bits et pour des entiers de $1{,}4t$ bits, en utilisant trois bibliothèques multi-précision standard : BigInteger d'Android (basée sur la bibliothèque C BIGNUM d'OpenSSL), BigInteger de Java SE (arithmétique en Java pur) et Gnu MP (sur x64). Ces bibliothèques sont représentatives de nombreux contextes applicatifs réels (GnuTLS, OpenSSL, Bouncy Castle utilisent ces primitives).

t (bits)	$\simeq 1{,}4t$	$\tau_\times$	$\tau_\%$	τ_*
BigInteger Android				
192	269	1,756	1,647	1,669
256	358	1,832	1,805	1,811
384	538	1,755	1,891	1,859
BigInteger Java SE				
192	269	1,157	1,323	**1,259**
256	358	1,170	1,322	**1,266**
384	538	1,371	1,552	1,486
Gnu MP, x64				
192	269	1,041	1,053	**1,049**
256	358	1,004	1,085	**1,059**
384	538	1,917	1,891	1,364

TABLE 3.4 – Rapport d'exécution entre opérations sur des entiers de $1{,}4t$ bits et de t bits. $\tau_\times$: multiplication, $\tau_\%$: réduction modulaire, τ_* : multiplication modulaire complète. Les cas favorables à la méthode EAC ($\tau_* < 1{,}29$) sont en gras.

Les résultats du Tableau 3.4 révèlent un écart important entre le rapport théorique $(1{,}4)^2 \approx 1{,}96$ et le rapport observé. Cet écart s'explique par le coût de la gestion mémoire (allocation et initialisation des objets BigInteger), dont le temps est quasi-indépendant de la taille des entiers. Sur Android, ce surcoût mémoire représente jusqu'à 54 % du temps total d'une multiplication modulaire, ramenant le rapport effectif bien en dessous de 1,29 pour $t \in \{192, 256\}$. La condition de compétitivité est donc vérifiée en pratique pour ces tailles, ce qui rend notre méthode pertinente dans des contextes utilisant des bibliothèques génériques.

3.4.3 Résultats expérimentaux

Nous avons implanté les méthodes EAC, W-GLV et W-GLV-SAC sur deux plateformes : un smartphone Android et un ordinateur x64 sous Gnu MP, pour un niveau de sécurité de 128 bits. Le Tableau 3.5 donne les rapports de temps d'exécution (un

rapport < 1 signifie que la méthode EAC est plus rapide).

Méthode / Plateforme	Android	Gnu MP, x64
EAC/W-GLV (sans encodage)	1,03	1,38
EAC/W-GLV (avec encodage)	**0,98**	1,32
EAC/W-GLV-SAC (sans enc.)	**0,75**	**0,98**
EAC/W-GLV-SAC (avec enc.)	**0,73**	**0,96**

TABLE 3.5 – Rapport d'exécution entre la méthode EAC et les méthodes GLV pour un niveau de sécurité de 128 bits. Un rapport < 1 indique que la méthode EAC est plus rapide.

Sur Android, notre méthode est au moins 25 % plus rapide que W-GLV-SAC et comparable à la version non sécurisée W-GLV lorsque l'encodage du scalaire k est pris en compte. Ce gain provient directement du fait que notre méthode n'a pas besoin d'encodage : la chaîne EAC est générée aléatoirement sans connaissance préalable de k. Sur x64 avec Gnu MP, notre méthode est compétitive avec W-GLV-SAC pour les niveaux de sécurité 96 et 128 bits.

4. Arithmétique des corps finis

La multiplication scalaire sur une courbe elliptique définie sur $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ peut être ramenée à une séquence d'opérations sur ce corps fini. Là encore la question de la représentation des objets manipulés est cruciale et conditionne le type d'algorithmes qui peuvent être utilisés. Traditionnellement, les éléments du corps $\mathbb{F}_p$ sont représentés sous forme d'entiers, et des réductions modulaires (autrement dit des divisions euclidiennes par p) sont utilisées pour maintenir la cohérence de la représentation après chaque opérations de corps (addition, multiplication etc).

Le *Polynomial Modular Number System* (PMNS), introduit par Bajard, Imbert et Plantard [BIP04], propose une représentation alternative dans laquelle les éléments de $\mathbb{F}_p$ sont représentés par des polynômes à coefficients bornés. L'arithmétique de corps est alors transposée dans un anneau polynomial $\mathbb{Z}[X]/(E)$ où E est un polynôme choisit de façon à rendre la réduction polynomiale très efficace, même pour un premier p sans forme particulière. La difficulté propre au PMNS est la *réduction interne* : ramener les coefficients d'un polynôme dans un certain intervalle. La grande majorité des travaux antérieurs utilisent pour cette étape une variante de l'algorithme de Montgomery [NP08].

Ce chapitre présente les travaux de [Mél22]. Après un rappel du formalisme PMNS et de la méthode de Montgomery pour la réduction interne, nous verrons deux nouvelles familles d'algorithmes fondées sur les procédés de Babai pour le problème du vecteur le plus proche dans un réseau euclidien [Bab86]. En pratique il s'agit d'une généralisation des approches de Barrett et Montgomery à un réseau quelconque. Une analyse de complexité et des résultats expérimentaux montrent que ces nouvelles approches sont compétitives pour les tailles d'entiers utilisés en cryptographie basée sur les courbes elliptiques.

4.1 Le système de représentation PMNS

4.1.1 Définition

Soit p un nombre premier et $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ le corps fini à p éléments. L'idée centrale du PMNS est de représenter chaque élément de $\mathbb{F}_p$ par un polynôme de degré au plus $n-1$ dont les coefficients restent bornés en valeur absolue.

Définition 4.1. *Un* système de représentation polynomiale modulaire *(PMNS) est un sextuplet $\mathcal{B} = (p, n, \gamma, \rho, E, l)$ où p est un nombre premier, n un entier, γ un entier vérifiant $E(\gamma) \equiv 0 \pmod{p}$, ρ un réel et $E \in \mathbb{Z}[X]$ un polynôme de degré n, tels que pour tout entier $0 \leqslant x < p$ il existe un polynôme $V(X) = \sum_{i=0}^{n-1} v_i X^i$ vérifiant*

$$V(\gamma) \equiv x \pmod{p} \quad \text{et} \quad \|V\|_l < \rho.$$

Un tel polynôme (ou vecteur $(v_0, \ldots, v_{n-1})$) est appelé *représentant* de x dans $\mathcal{B}$, noté $V \equiv_{\mathcal{B}} x$. Un même entier peut admettre plusieurs représentants.

Remarque 4.2. *La norme utilisée classiquement est la norme infinie, ce qui revient à demander $|v_i| < \rho$ pour tout i. Dans [Mél22], les normes 1 et 2 sont également considérées pour affiner l'analyse des algorithmes de Babai.*

Exemple 4.3. *Soit $p = 1\,048\,573$, $n = 5$, $\gamma = 238\,019$, $\rho = 36$, $E = X^5 - 2$. On vérifie que $E(\gamma) \equiv 0 \pmod{p}$, et $\mathcal{B} = (p, 5, \gamma, 36, X^5 - 2, \infty)$ est un PMNS. La multiplication de $A = -9X^4 + 13X^3 - 35X^2 - 25$ et de $B = 24X^4 - 32X^3 + 21X^2 - 2X - 16$ se déroule en trois étapes.*

1. ***Multiplication polynomiale*** *(degré $2n-2$ au plus) :*

$$\begin{aligned} AB = &-216X^8 + 600X^7 - 1445X^6 + 1411X^5 \\ &-1217X^4 + 662X^3 + 35X^2 + 50X + 400. \end{aligned}$$

2. ***Réduction externe*** *(réduction polynomiale modulo E) :*

$$C = AB \bmod E = -1217X^4 + 230X^3 + 1235X^2 - 2840X + 3222.$$

3. ***Réduction interne*** *(réduction des coefficients) :*

$$C \equiv_{\mathcal{B}} C' = 3X^4 + 2X^3 + 8X^2 - X - 1.$$

On vérifie que $A(\gamma)B(\gamma) \equiv C'(\gamma) \pmod{p}$ et $\|C'\|_\infty = 8 < 36 = \rho$.

Les deux premières étapes font appel à des algorithmes standards. C'est la troisième, la réduction interne, qui est l'opération critique et l'objet principal de ce chapitre.

4.1.2 Le réseau associé au PMNS

La réduction interne revient à trouver, parmi tous les polynômes équivalents au polynôme résultant C, un représentant à petits coefficients. Deux polynômes V et W sont équivalents si et seulement si $V(\gamma) \equiv W(\gamma) \pmod{p}$, c'est-à-dire si leur différence appartient au réseau suivant.

Définition 4.4. *Le* réseau associé au PMNS $\mathcal{B} = (p, n, \gamma, \rho, E, l)$ *est l'ensemble*

$$\mathcal{L}(\mathcal{B}) = \left\{(x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} x_i \gamma^i \equiv 0 \pmod{p}\right\}.$$

Ce réseau est de dimension n et admet pour base les lignes de la matrice

$$G = \begin{pmatrix} p & 0 & 0 & \cdots & 0 \\ -\gamma & 1 & 0 & \cdots & 0 \\ -\gamma^2 & 0 & 1 & \cdots & 0 \\ \vdots & & \ddots & & \vdots \\ -\gamma^{n-1} & 0 & \cdots & 0 & 1 \end{pmatrix}.$$

Le *problème de réduction interne du PMNS* consiste alors, étant donné un vecteur $v \in \mathbb{Z}^n$, à trouver $s \in \mathcal{L}(\mathcal{B})$ tel que $\|v - s\|_l < \rho$. C'est une variante du *Closest Vector Problem* (CVP). Dans le cas général, ce problème est NP-dur. Heureusement dans notre cas la petite dimension n et la tolérance sur la norme (on ne cherche pas *le* vecteur le plus proche mais un vecteur proche) rend le problème beaucoup plus facile à résoudre.

4.1.3 Réduction interne par l'algorithme de Montgomery

La méthode classique de réduction interne, proposée par Nègre et Plantard [NP08] et utilisée dans tous les travaux ultérieurs, est une adaptation de la multiplication modulaire de Montgomery [Mon85]. Elle requiert l'existence d'un polynôme $M \in \mathcal{B}$ tel que $M(\gamma) \equiv 0 \pmod{p}$, et d'un entier ϕ ; on calcule $M' = -M^{-1} \bmod (E, \phi)$.

Algorithm 5 RedCoeff [NP08]

Require: $\mathcal{B} = (p, n, \gamma, \rho, E, \infty)$, $V \in \mathbb{Z}_n[X]$, $M \in \mathcal{B}$, $\phi \in \mathbb{N}^*$, $M' = -M^{-1} \bmod (E, \phi)$
Ensure: $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$
1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: **return** S

Le coût de cet algorithme est de $2n^2$ multiplications de mots machine, $(3n^2 - n)$ additions et n décalages. Il constitue la référence par rapport à laquelle nous mesurons l'efficacité des nouvelles approches.

4.2 Réduction interne par les algorithmes de Babai

Les algorithmes de Babai [Bab86] constituent des heuristiques efficaces pour le CVP. Ils ne s'appliquent pas directement à la réduction interne du PMNS, car leurs coefficients font intervenir les coordonnées de la base de Gram-Schmidt ou les coefficients de l'inverse de la matrice du réseau, dont les numérateurs et dénominateurs peuvent être aussi grands que le premier p. Cette section décrit comment les adapter au problème de la réduction interne [Mél22].

4.2.1 Préliminaires : orthogonalisation de Gram-Schmidt

Soit $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ une base du réseau $\mathcal{L}(\mathcal{B})$. Son *orthogonalisation de Gram-Schmidt* (OGS) est la famille orthogonale $\tilde{\mathbf{B}} = (\tilde{\mathbf{b}}_0, \ldots, \tilde{\mathbf{b}}_{n-1})$ définie par $\tilde{\mathbf{b}}_0 = \mathbf{b}_0$ et, pour $i \geqslant 1$,

$$\tilde{\mathbf{b}}_i = \mathbf{b}_i - \sum_{j=0}^{i-1} \mu_{ij}\, \tilde{\mathbf{b}}_j, \quad \mu_{ij} = \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_j \rangle}{\|\tilde{\mathbf{b}}_j\|_2^2}.$$

On rappelle que $\tilde{\mathbf{b}}_i \perp \operatorname{span}(\mathbf{b}_0, \ldots, \mathbf{b}_{i-1})$, et que $\det(\mathcal{L}) = \prod_{i=0}^{n-1} \|\tilde{\mathbf{b}}_i\|_2 = p$.

Une base $\mathbf{B}$ est dite *LLL-réduite* [LLL82] si ses vecteurs sont courts et quasi-orthogonaux. En petite dimension, l'algorithme LLL fournit des bases dont les vecteurs ont une norme de l'ordre de $p^{1/n}$.

4.2.2 Algorithme du plan le plus proche

L'algorithme de Babai par *plan le plus proche* (*Nearest Plane*) repose tout d'abord sur l'observation qu'il est facile de réduire un vecteur dans une base orthogonale : il est

possible de réduire coordonnée par coordonnée.

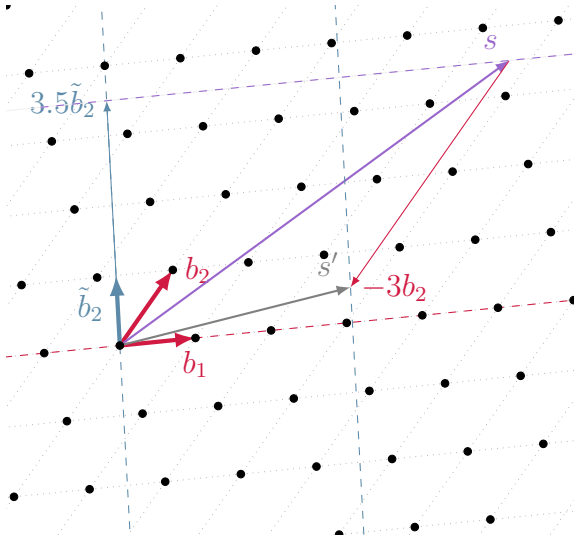


FIGURE 4.1 – Etape de réduction de l'algorithme par *plan le plus proche*

La Figure 4.1 montre comment l'algorithme l'algorithme réduit successivement chaque coordonnée du vecteur v dans la base de départ en utilisant la base OGS pour obtenir une approximation du coefficient à appliquer. Dans cet exemple, on cherche à réduire la norme du vecteur s en retranchant le vecteur b_2. Pour ce faire, on projette s sur l'espace engendré par $\tilde{b}_2$ (intersection des lignes en pointillés bleues et violettes) et on évalue la norme du projeté par rapport à celle de $\tilde{b}_2$ (ici 3.5). On peut ainsi retrancher $3b_2$ au vecteur s pour obtenir s'. Dans la base OGS, $s' = s'_1\tilde{b}_1 + s'_2\tilde{b}_2$ vérifie alors $|s'_2| = 0.5$.

Dans le cas général, à l'étape i, on calcule l'entier c qui minimise la projection de $(s - c\,\mathbf{b}_i)$ sur $\tilde{\mathbf{b}}_i$, à savoir

$$c_i = \left\lfloor \frac{\langle s, \tilde{\mathbf{b}}_i \rangle}{\|\tilde{\mathbf{b}}_i\|_2^2} \right\rceil .$$

Algorithm 6 Algorithme de Babai – Plan le plus proche [Bab86]

Require: Réseau $\mathcal{L}$, vecteur $v \in \mathbb{Z}^n$, base $\mathbf{B}$ et OGS $\tilde{\mathbf{B}}$

Ensure: $s \in \mathbb{Z}^n$ tel que $v - s \in \mathcal{L}$ et $v - s$ est proche de v

1: $s \leftarrow v$
2: **for** $i = n - 1$ **à** 0 **do**
3: $\quad c \leftarrow \left\lfloor \langle s, \tilde{\mathbf{b}}_i \rangle / \|\tilde{\mathbf{b}}_i\|_2^2 \right\rceil$
4: $\quad s \leftarrow s - c\,\mathbf{b}_i$
5: **end for**
6: **return** s

La sortie s vérifie $s = \sum_{i=0}^{n-1} s_i\,\tilde{\mathbf{b}}_i$ avec $|s_i| \leqslant 1/2$, ce qui donne la borne $\|s\|_2 \leqslant$

$\frac{1}{2}\sqrt{\sum_i \|\tilde{\mathbf{b}}_i\|_2^2}$.

Adaptation au PMNS. Pour utiliser l'Algorithme 6 en pratique, il faut calculer $c_i = \lfloor \langle s, \tilde{\mathbf{b}}_i \rangle / \|\tilde{\mathbf{b}}_i\|_2^2 \rceil$. Les coefficients $\tilde{\mathbf{b}}_{i,j}/\|\tilde{\mathbf{b}}_i\|_2^2$ sont des rationnels de grandes tailles (celle de p). L'idée est donc de les approximer par des entiers tenant dans un registre machine de w bits.

Soient h_1 et h_2 deux entiers tels que

$$G_{i,j} = \left\lfloor \frac{2^{h_1}\,\tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|_2^2} \right\rceil \quad \text{et} \quad \left\lfloor \frac{s_j}{2^{h_2}} \right\rceil$$

soient des entiers de taille w bits pour tous les i, j. On peut alors approcher c_i par

$$R_i = \left\lfloor \frac{\sum_j \left\lfloor \frac{s_j}{2^{h_2}} \right\rceil \times G_{i,j}}{2^{h_1-h_2}} \right\rceil.$$

La matrice $(G_{i,j})$ ne dépend pas du vecteur d'entrée et peut être précalculée.

Algorithm 7 Réduction PMNS - Nearest Plane [Mél22]

Require: Réseau $\mathcal{L}$, base $\mathbf{B}$, OGS $\widetilde{\mathbf{B}}$, entiers précalculés $G_{i,j} = \lfloor 2^{h_1}\tilde{\mathbf{b}}_{i,j}/\|\tilde{\mathbf{b}}_i\|_2^2 \rceil$, paramètres h_1, h_2

Ensure: s équivalent à v avec de petits coefficients

```
s ← v
for i = n − 1 à 0 do
    c ← 0
    for j = 0 à n − 1 do
        r ← s_j ≫ h_2
        c ← c + r × G_{i,j}
    end for
    r ← c ≫ (h_1 − h_2)
    s ← s − r b_i
end for
return s
```

Analyse du coût. Les quantités $G_{i,j}$ sont précalculées. À chaque itération de la boucle principale, la boucle intérieure effectue n décalages, n multiplications w-bits et $(n-1)$ additions $2w$-bits. En notant mul_w, add_w et shft les coûts respectifs, le coût total de l'Algorithme 7 est

$$2n^2\,\mathsf{mul}_w + (4n^2 - 3n)\,\mathsf{add}_w + (n^2 + n)\,\mathsf{shft}.$$

Analyse de la réduction des coefficients. La sortie s de l'algorithme vérifie $s = \sum_i s_i \tilde{\mathbf{b}}_i$, où chaque s_i dépend de l'erreur d'approximation commise lors du calcul de R_i.

Lemme 4.5 ([Mél22, Lem. 1]). *Soit $s^{(i)}$ la valeur de la variable s au début de l'itération $n-1-i$ de l'Algorithme 7. La sortie S satisfait $S = \sum_i S_i \tilde{\mathbf{b}}_i$ avec, pour tout i,*

$$|S_i| \leqslant e_i = \frac{\|s^{(i)}\|_1}{2^{h_1+1}} + K_i,$$

où $K_i = \frac{2^{h_2}}{\|\tilde{\mathbf{b}}_i\|_2^2} |\sum_j \tilde{\mathbf{b}}_{i,j}| + \frac{n}{2^{h_1-h_2+1}} + 1$ est une constante indépendante de v.

Démonstration. Puisque $\tilde{\mathbf{b}}_i \perp \mathrm{span}(\mathbf{b}_0, \ldots, \mathbf{b}_{i-1})$, l'instruction $s \leftarrow s - r\,\mathbf{b}_i$ ne modifie pas S_k pour $k > i$; ainsi S_i ne dépend que de $s^{(i)}$. En développant

$$R_i = \frac{1}{2^{h_1-h_2}} \sum_j \left(\frac{s_j^{(i)}}{2^{h_2}} - \varepsilon_j \right) \left(\frac{2^{h_1} \tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|_2^2} - \varepsilon'_j \right) + \varepsilon$$

avec $0 \leqslant \varepsilon, \varepsilon_j < 1$ et $-1/2 \leqslant \varepsilon'_j < 1/2$, on obtient

$$e_i = \left| \sum_j s_j^{(i)} \frac{\tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|_2^2} - R_i \right| \leqslant \frac{\|s^{(i)}\|_1}{2^{h_1+1}} + \frac{2^{h_2}}{\|\tilde{\mathbf{b}}_i\|_2^2} \left| \sum_j \tilde{\mathbf{b}}_{i,j} \right| + \frac{n}{2^{h_1-h_2+1}} + 1. \qquad \square$$

Proposition 4.6 ([Mél22, Prop. 1]). *Posons $\mu_2 = \sqrt{\sum_i \|\tilde{\mathbf{b}}_i\|_2^2}$. Il existe des constantes α et K telles que la sortie S de l'Algorithme 7 satisfait*

$$\|S\|_2 \leqslant \left(\frac{\alpha \|v\|_1}{2^{h_1+1}} + K \right) \mu_2.$$

Corollaire 4.7 ([Mél22, Cor. 1]). *Soit $(p, n, \gamma, \rho, E, 2)$ un PMNS. Si v vérifie*

$$\|v\|_1 \leqslant \frac{2^{h_1+1}}{\alpha} \left(\frac{\rho}{\mu_2} - K \right),$$

alors l'Algorithme 7 résout le problème de réduction interne du PMNS.

4.2.3 Algorithme par arrondi

L'algorithme de Babai par *arrondi* (*Rounding*) est plus direct que le précédent. Il consiste à exprimer v dans la base $\mathbf{B}$ à l'aide de la matrice inverse $\mathbf{B}^{-1}$, à arrondir chaque coordonnée rationnelle à l'entier le plus proche, puis à en déduire un vecteur proche :

$$s = v - \lfloor v\,\mathbf{B}^{-1} \rceil\, \mathbf{B}.$$

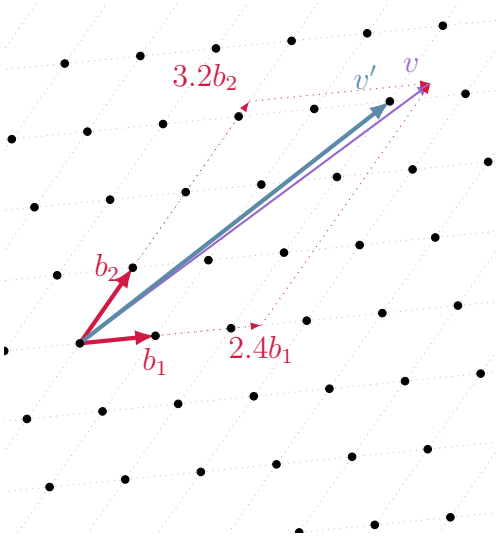


FIGURE 4.2 – Réduction de vecteur par l'algorithme Babai *Rounding*

La Figure 4.2 montre comment l'algorithme fonctionne en dimension 2. Le vecteur de départ v est projeté dans la base (b_1, b_2). On obtient ainsi $v = 2.4b_1 + 3.2b_2$. On en déduit un vecteur proche du réseau par arrondi des coordonnées : $v' = 2b_1 + 3b_2$.

Algorithm 8 Algorithme de Babai – Arrondi [Bab86]

Require: Réseau $\mathcal{L}$, vecteur $v \in \mathbb{Z}^n$, base $\mathbf{B}$, inverse $\mathbf{B}^{-1} = (\mathbf{b}'_0, \ldots, \mathbf{b}'_{n-1})$

Ensure: $s \in \mathbb{Z}^n$ tel que $v - s \in \mathcal{L}$ et $v - s$ est proche de v

1: $s \leftarrow v$
2: **for** $i = 0$ **à** $n - 1$ **do**
3: $\quad r \leftarrow \lfloor \langle v, \mathbf{b}'_i \rangle \rceil$
4: $\quad s \leftarrow s - r\,\mathbf{b}_i$
5: **end for**
6: **return** s

La sortie vérifie $s = \sum_i s_i\,\mathbf{b}_i$ avec $|s_i| \leqslant 1/2$, d'où $\|s\|_1 \leqslant \frac{1}{2}\sum_i \|\mathbf{b}_i\|_1$.

Adaptation au PMNS. De même que précédemment, les coefficients de $\mathbf{B}^{-1}$ sont des rationnels de grandes tailles. On les approxime par des entiers $B'_{i,j} = \lfloor 2^{h_1}\,\mathbf{b}'_{i,j} \rceil$ de taille w bits, et le vecteur résultant est calculé par

$$s = v - \left\lfloor \frac{\lfloor v/2^{h_2} \rfloor \, \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1 - h_2}} \right\rfloor \mathbf{B}.$$

Algorithm 9 Réduction PMNS – Arrondi [Mél22]

Require: Réseau $\mathcal{L}$, base $\mathbf{B}$ et inverse approximé $B'_{i,j} = \lfloor 2^{h_1} \mathbf{b}'_{i,j} \rceil$, paramètres h_1, h_2
Ensure: s équivalent à v avec de petits coefficients

$s \leftarrow v$
$v' \leftarrow (v_0 \gg h_2, \ldots, v_{n-1} \gg h_2)$
for $i = 0$ **à** $n - 1$ **do**
    $r \leftarrow 0$
    for $j = 0$ **à** $n - 1$ **do**
        $r \leftarrow r + v'_j \times B'_{i,j}$
    end for
    $r \leftarrow r \gg (h_1 - h_2)$
    $s \leftarrow s - r\,\mathbf{b}_i$
end for
return s

Analyse du coût. L'algorithme effectue des produits scalaires. Son coût est

$$2n^2\,\mathsf{mul}_w + (3n^2 - 2n)\,\mathsf{add}_w + 2n\,\mathsf{shft}.$$

Proposition 4.8 ([Mél22, Prop. 2]). *Posons $\mu_1 = \sum_i \|\mathbf{b}_i\|_1$. Il existe une constante K telle que*

$$\|S\|_1 \leqslant \left(\frac{\|v\|_1}{2^{h_1+1}} + K\right)\mu_1.$$

Corollaire 4.9 ([Mél22, Cor. 2]). *Soit $(p, n, \gamma, \rho, E, 1)$ un PMNS. Si v vérifie*

$$\|v\|_1 \leqslant 2^{h_1+1}\left(\frac{\rho}{\mu_1} - K\right),$$

alors l'Algorithme 9 résout le problème de réduction interne du PMNS.

4.2.4 Versions optimisées et compromis erreur-vitesse

Les deux algorithmes peuvent être accélérés au prix d'une légère augmentation de l'erreur d'arrondi. L'idée est de remplacer l'approximation

$$\sum_j a_j \approx \left\lfloor \frac{\sum_j \lfloor a_j\, 2^h \rfloor}{2^h} \right\rfloor$$

par l'approximation plus grossière

$$\sum_j a_j \approx \sum_j \left\lfloor \frac{\lfloor a_j\, 2^h \rfloor}{2^h} \right\rfloor.$$

La seconde forme effectue les additions sur des entiers w-bits (au lieu de $2w$-bits), les décalages étant du type *shift* de w bits, virtuellement gratuit sur une architecture w-bits. En notant shft_w ces décalages, les algorithmes optimisés (Algorithmes 10 et 11) ont pour complexité respective :

$$2n^2\,\mathsf{mul}_w + (3n^2 - 2n)\,\mathsf{add}_w + n^2\,\mathsf{shft} + n^2\,\mathsf{shft}_w$$

et

$$2n^2\,\mathsf{mul}_w + (2n^2 - n)\,\mathsf{add}_w + n\,\mathsf{shft} + n^2\,\mathsf{shft}_w.$$

Algorithm 10 Réduction PMNS – Plan le plus proche optimisé [Mél22]

$s \leftarrow v$
for $i = n-1$ **à** 0 **do**
  $c \leftarrow 0$
  for $j = 0$ **à** $n-1$ **do**
    $r \leftarrow s_j \gg h_2$
    $c \leftarrow c + \Big((r \times G_{i,j}) \gg (h_1 - h_2)\Big)$
  end for
  $s \leftarrow s - c\,\mathbf{b}_i$
end for
return s

Algorithm 11 Réduction PMNS – Arrondi optimisé [Mél22]

$s \leftarrow v$
$v' \leftarrow (v_0 \gg h_2, \ldots, v_{n-1} \gg h_2)$
for $i = 0$ **à** $n-1$ **do**
  $r \leftarrow 0$
  for $j = 0$ **à** $n-1$ **do**
    $r \leftarrow r + \Big((v'_j \times B'_{i,j}) \gg (h_1 - h_2)\Big)$
  end for
  $s \leftarrow s - r\,\mathbf{b}_i$
end for
return s

4.3 Résultats

4.3.1 Condition de validité

Pour que les algorithmes de réduction s'appliquent, le vecteur d'entrée $v = AB \bmod E$ doit satisfaire les conditions des Corollaires 4.7 et 4.9. La borne sur $\|v\|_1$ dépend du choix de E.

Théorème 4.10 ([Mél22, Thm. 1]). *Soit $(p, n, \gamma, \rho, X^n - \lambda, l)$ un PMNS. Pour $A, B \in \mathbb{Z}[X]$ de degré $n-1$ avec $\|A\|_l \leqslant \rho$, $\|B\|_l \leqslant \rho$ et $V = AB \bmod (X^n - \lambda)$, on a les conditions suffisantes suivantes.*

1. *(Arrondi, $l = 1$) : si $|\lambda| K \mu_1^2 \leqslant 2^{h_1 - 1}$, alors l'Algorithme 9 résout le problème de réduction interne.*
2. *(Plan le plus proche, $l = 2$) : si $n\alpha|\lambda| K \mu_2^2 \leqslant 2^{h_1 - 1}$, alors l'Algorithme 7 résout le problème de réduction interne.*

Ce résultat met en évidence le rôle de λ : pour un premier p donné, le polynôme $E = X^n - \lambda$ doit être choisi avec $|\lambda|$ le plus petit possible. En pratique, on cherche λ tel que E admette une racine γ modulo p, avec $|\lambda|$ minimal.

4.3.2 Génération des paramètres

La génération des paramètres pour un premier p donné suit les étapes suivantes.

1. Choisir n tel que $64n \geqslant \lceil \log_2 p \rceil$ (pour une architecture 64 bits).
2. Trouver λ minimal tel que $E = X^n \pm \lambda$ admette une racine γ dans $\mathbb{F}_p$.
3. Calculer le réseau $\mathcal{L}(\mathcal{B})$ et en extraire une base LLL-réduite $\mathbf{B}$.
4. Calculer l'OGS $\widetilde{\mathbf{B}}$ et l'inverse $\mathbf{B}^{-1}$.
5. Pour chaque algorithme, trouver (h_1, h_2) tels que les coefficients précalculés tiennent dans un registre 64 bits et que le Théorème 4.10 soit satisfait.
6. Si aucun paramètre ne convient, incrémenter n ou changer λ.

Un outil de génération automatique de code C a été développé en SageMath dans le cadre de [Mél22] et mis à disposition publiquement.

4.3.3 Comparaison des complexités

Le Tableau 4.1 récapitule les complexités théoriques des différents algorithmes de réduction interne.

Algorithme	mul_w	add_w	shft	shft_w
Montgomery [Did+19]	$2n^2$	$3n^2 - n$	0	n
Plan le plus proche	$2n^2$	$4n^2 - 3n$	n^2	n
Plan le plus proche (opt.)	$2n^2$	$3n^2 - 2n$	n^2	n^2
Arrondi	$2n^2$	$3n^2 - 2n$	n	n
Arrondi (opt.)	$2n^2$	$2n^2 - n$	n	n^2

TABLE 4.1 – Complexités théoriques des algorithmes de réduction interne.

La version optimisée de l'arrondi se distingue par un nombre d'additions nettement inférieur aux autres méthodes : $2n^2 - n$ contre $3n^2 - n$ pour Montgomery, soit une réduction de l'ordre de 33% pour des valeurs typiques de n.

Remarque 4.11. *Il est à noter qu'une optimisation similaire à celle utilisée ici pour les algorithmes de Babai est possible pour l'algorithme de Montgomery mais elle n'avait pas été implantée à l'époque dans [Did+19]. Cette optimisation rend à nouveau les deux approches quasi identiques en termes de coût de calcul.*

4.3.4 Résultats expérimentaux

Les algorithmes ont été implantés en C et mesurés sur un processeur Intel Core i5-10500 @ 3,10 GHz, pour les tailles de premiers utilisées en cryptographie sur courbes elliptiques. Les Tableaux 4.2 et 4.3 reproduisent les résultats obtenus dans [Mél22].

Taille de p (bits)	192	224		256	384		512	
Degré n	4	4	5	5	7	8	10	11
Montgomery [Did+19]	60	63	86	83	155	206	300	357
Plan le plus proche	83	83	143	133	265	331	494	606
Plan le plus proche (opt.)	70	70	108	117	196	248	405	494
Arrondi	50	52	76	74	155	205	321	362
Arrondi (opt.)	**45**	**45**	**64**	**70**	**148**	**199**	**291**	**344**

TABLE 4.2 – Nombre de cycles processeur pour une réduction interne, polynôme $E = X^n - \lambda$.

Taille de p (bits)	192	224		256	384		512	
Degré n	4	4	5	5	7	8	10	11
Montgomery [Did+19]	191	188	255	254	496	649	966	1122
Plan le plus proche	228	214	388	357	793	1021	1624	2014
Plan le plus proche (opt.)	194	199	298	304	538	743	1218	1486
Arrondi	181	178	261	253	468	644	987	1021
Arrondi (opt.)	**159**	**156**	**221**	**220**	**381**	**493**	**735**	**879**

TABLE 4.3 – Nombre d'instructions CPU pour une réduction interne, polynôme $E = X^n - \lambda$.

Les résultats confirment l'analyse théorique. L'algorithme d'arrondi et sa version optimisée sont plus rapides que Montgomery pour la majorité des tailles, avec un gain de 10 à 20% sur le nombre d'instructions.

En revanche, l'algorithme du plan le plus proche est pénalisé par ses nombreux décalages, qui se révèlent coûteux en pratique même si leur coût est souvent négligé dans les analyses théoriques.

5. Arithmétique des ordinateurs

Le PMNS est compétitif avec les bibliothèques multi-précision standard pour des premiers allant jusqu'à environ 1024 bits. Au-delà, deux phénomènes dégradent ses performances. D'une part, la génération des paramètres devient de plus en plus difficile. D'autre part, le degré n des polynômes PMNS croît, pour des raisons arithmétiques, plus rapidement que la valeur optimale $n_{\text{opt}} = \lfloor \log_2 p/k \rfloor + 1$ dictée par la taille des registres, induisant un surcoût difficile à compenser. Par exemple, considérons des entiers de 256-bits à représenter sur une architectures 64-bits. Une représentation optimale en nombre de registre stockerait de tels entiers sur 4 registres. Nous l'avons vu au chapitre précédent, concernant les PMNS, seules des représentations avec $n = 5$ existent. Expérimentalement, l'écart se creuse avec l'augmentation de la tailles des entiers, par exemple, pour des tailles de 3200-bits 51 registres suffisent à stoker de tels entiers mais en pratique on ne trouve pas de PMNS pour n inférieur à 74 [Pal25, page 86].

Ce chapitre présente les travaux de [MPV23], qui proposent des réponses à ces deux problèmes et étendent la portée du PMNS à des entiers de très grande taille à partir des travaux effectués sur les algorithmes basés sur la méthode de réduction de Montgomery. La première contribution est un algorithme de réduction interne généralisé, fondé sur l'algorithme de Montgomery appliqué aux réseaux, qui supprime toute recherche exhaustive dans la génération des paramètres (Section 5.1). La deuxième est une implémentation en arithmétique 128 bits qui réduit significativement le degré effectif des polynômes pour les grands premiers (Section 5.2).

Une fois n'est pas coutume, ce dernier chapitre est plus le fruit de mon co-encadrement (avec Pascal Véron) de la thèse de François Palma [Pal25] que le résultat de travaux personnels. Je présente ici des idées qui se trouvent à la base de certains résultats de cette thèse. Les travaux sur le sujet sont bien plus complet et je renvoie le lecteur aux chapitres 2 et 3 de cette thèse pour plus de détails.

5.1 Génération de paramètres

5.1.1 Limitation de l'approche classique

Rappelons pour commencer l'algorithme de réduction interne vue au chapitre précédent basé sur l'algorithme de Montgomery.

Algorithm 12 RedCoeff [NP08]

Require: $\mathcal{B} = (p, n, \gamma, \rho, E, \infty)$, $V \in \mathbb{Z}_n[X]$, $M \in \mathcal{B}$, $\phi \in \mathbb{N}^*$, $M' = -M^{-1} \bmod (E, \phi)$
Ensure: $S(\gamma) \equiv V(\gamma)\phi^{-1} \pmod{p}$
1: $Q \leftarrow V \times M' \bmod (E, \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: **return** S

La construction d'un PMNS valide permettant l'utilisation de cet algorithme nécessite, en plus des paramètres du PMNS, la production d'un certain polynôme M inversible modulo E et modulo ϕ, où ϕ est en général une puissance de 2 dépendant de l'architecture cible de l'implantation (typiquement sur une architecture 64-bits $\phi = 2^{64}$).

Le processus de génération des paramètres est alors le suivant.

1. Choisir $\phi = 2^k$ et $n > \log_2 p/k$.
2. Trouver un polynôme irréductible de degré n admettant une racine γ modulo p.
3. Construire le réseau $\mathcal{L}$ associé.
4. Appliquer l'algorithme LLL pour obtenir une base réduite de $\mathcal{L}$.
5. Trouver $M(X) \in \mathcal{L}$ à petits coefficients et inversible modulo (E, ϕ).
6. Poser $\rho = 2\|\mathcal{M}\|_1$.
7. Vérifier $\phi \geq 2w\rho$.

L'étape 5 est potentiellement la plus coûteuse. Il est prouvé dans [DRV22] que, dans certaines situations, la complexité en pire cas de cette recherche est exponentielle en n. Pour une architecture 64 bits, le degré minimal d'un polynôme satisfaisant les contraintes précédentes vérifie $n \geq n_{\text{opt}} = \lfloor \log_2 p/64 \rfloor + 1$. Pour un premier de 256 bits, la recherche implique $2^5 = 32$ itérations et reste faisable. Pour 4096 bits, $n_{\text{opt}} = 65$ et la recherche exige au moins 2^{65} itérations.

5.1.2 Algorithme de Montgomery sur les réseaux

Soit $\mathcal{L}$ la matrice dont les lignes forment une base LLL-réduite de $\mathfrak{L}$, et $C = (C_0, \ldots, C_{n-1})$ le vecteur des coefficients du polynôme à réduire. On cherche $q \in \mathbb{Z}^n$ tel que $C + q\mathcal{L} \equiv 0 \pmod{\phi}$, c'est-à-dire $q \equiv -C\mathcal{L}^{-1} \pmod{\phi}$. Comme $\det(\mathcal{L}) = p$ est premier avec $\phi = 2^k$, l'inverse $\mathcal{L}^{-1} \bmod \phi$ existe toujours.

Algorithm 13 Réduction interne généralisée [MPV23]

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$ un PMNS, $C \in \mathbb{Z}_{n-1}[X]$, $\mathcal{L}$ une base réduite du réseau $\mathfrak{L}$, $\phi \in \mathbb{N}^*$, $-\mathcal{L}^{-1} \bmod \phi$

Ensure: $S(\gamma) \equiv C(\gamma)\,\phi^{-1} \pmod{p}$, avec $S \in \mathbb{Z}_{n-1}[X]$

1: $q \leftarrow C\,(-\mathcal{L}^{-1}) \bmod \phi$
2: $S \leftarrow (C + q\mathcal{L})/\phi$
3: **return** S

L'Algorithme 13 ne nécessite aucune recherche de polynôme $M(X)$: la base $\mathcal{L}$ est calculée une fois pour toutes par LLL, et le reste de la procédure est déterministe.

Proposition 5.1 ([MPV23, Prop. 3.1])**.** *Soit $C(X)$ un polynôme de degré au plus $n-1$. Si*

$$\|C\|_\infty \leq \phi\Big(\rho - \|\mathcal{L}\|_1\Big),$$

alors la sortie S de l'Algorithme 13 vérifie $\|S\|_\infty < \rho$, soit $S \in \mathcal{B}$.

La proposition suivante donne les conditions sur ϕ et ρ pour que la sortie d'une multiplication PMNS satisfasse la Proposition 5.1.

Proposition 5.2 ([MPV23, Prop. 3.2])**.** *Soient $A(X), B(X) \in \mathcal{B}$ et $C(X) = A(X)B(X) \bmod E(X)$. Soit $w > 0$ tel que $\|C\|_\infty \leq w\|A\|_\infty\|B\|_\infty$. Si $\phi \geq 2w\rho$ et $\rho \geq 2\|\mathcal{L}\|_1$, alors C satisfait la condition de la Proposition 5.1.*

Remarque 5.3. *La valeur w dépend uniquement de la forme du polynôme $E(X)$. Une borne précise est établie dans [DRV22]. En pratique, pour $\phi = 2^{64}$, on choisit $\phi = 2w\rho$ de façon à minimiser simultanément w et ρ.*

La nouvelle procédure de génération des paramètres se réduit donc à cinq étapes, sans recherche exhaustive.

1. Choisir $n \geq n_{\text{opt}} = \lfloor \log_2 p/64 \rfloor + 1$.
2. Trouver un polynôme irréductible $E(X)$ de degré n admettant une racine γ modulo p.
3. Construire le réseau $\mathfrak{L}$ et appliquer LLL pour obtenir la base réduite $\mathcal{L}$.

4. Poser $\rho = 2\|\mathcal{L}\|_1$ et $\phi = 2w\rho$ (Proposition 5.2).
5. Si les contraintes pratiques ($\rho < 2^{63}$, $\phi = 2^{64}$) ne sont pas satisfaites, recommencer à l'étape 1 ou 2 avec un autre n ou un autre E.

La Table 5.1 montre que les performances de la réduction interne généralisée sont comparables à celles de la réduction de Montgomery classique sur les tailles standard.

Taille de p (bits)	256	512	1024
Montgomery [NP08]	121	432	1709
Ce travail	122	443	1728

TABLE 5.1 – Nombre de cycles processeur pour une multiplication modulaire PMNS, processeur Intel i9-11900KF, gcc 12.1.0.

5.2 PMNS en arithmétique 128 bits

5.2.1 Le problème de la taille des données

Même avec la procédure de génération généralisée, les performances du PMNS se dégradent pour les grands premiers. La cause est structurelle : le degré n des polynômes PMNS doit satisfaire la contrainte

$$p^{1/n} \leq \frac{\phi}{4w},$$

qui découle des Propositions 5.1 et 5.2 combinées aux bornes classiques sur les vecteurs courts d'un réseau. Pour $\phi = 2^{64}$, cette contrainte force n à s'écarter de n_{opt} dès que p est assez grand, ce que quantifie la proposition suivante.

Proposition 5.4 ([MPV23, Prop. 4.1])**.** *Pour tout $\varepsilon > 0$, il existe un entier p tel que*

$$p^{\frac{1}{n_{opt}+\varepsilon}} > \frac{\phi}{4w}.$$

Exemple 5.5. *Pour p de 2048 bits, $n_{opt} = 33$. D'un côté, $p^{1/n_{opt}} \approx 2^{62,1}$; de l'autre, puisque $w \geq n$, $\phi/(4w) \lesssim 2^{56,9}$. La contrainte $p^{1/n} \leq \phi/(4w)$ exige donc $n \geq 37$, soit un surcoût de quatre coefficients par rapport à n_{opt}.*

5.2.2 Solution : $\phi = 2^{128}$

La solution proposée dans [MPV23] est d'augmenter ϕ de 2^{64} à 2^{128} : cela repousse la contrainte $p^{1/n} \leq \phi/(4w)$ et permet de choisir n beaucoup plus proche de n_{opt}.

L'implémentation repose sur le type `__int128` du compilateur gcc, qui manipule les entiers 128 bits à l'aide de deux registres 64 bits.

On note n_{64} (resp. n_{128}) la valeur pratique de n obtenue avec $\phi = 2^{64}$ (resp. $\phi = 2^{128}$).

Taille de p (bits)	256	512	1024	2048	4096	8192
$n_{\text{opt},64}$	5	9	17	33	65	129
n_{64}	5	9	19	40	83	187
$n_{\text{opt},128}$	3	5	9	17	33	65
n_{128}	3	5	9	18	36	72

TABLE 5.2 – Degrés optimaux et pratiques des polynômes PMNS selon la taille de ϕ.

La Table 5.2 illustre le gain : pour un premier de 8192 bits, $n_{64} = 187$ contre $n_{128} = 72$. Comme les opérations de réduction interne sont quadratiques en n, la version 128 bits réduit le nombre d'opérations élémentaires d'un facteur proche de $(187/72)^2 \approx 6{,}7$ en théorie.

En pratique, l'implémentation 128 bits introduit des surcoûts : chaque coefficient nécessite deux registres 64 bits, et la multiplication de deux coefficients demande quatre multiplications 64 bits. La comparaison des versions **Red-64** ($\phi = 2^{64}$) et **Red-128** ($\phi = 2^{128}$) est donnée dans la Table 5.3.

Taille de p (bits)	256	512	1024	2048	4096	8192
Red-64	122	443	1709	7690	33471	169070
Red-128	245	720	2367	10920	39446	**156310**

TABLE 5.3 – Cycles processeur pour une multiplication modulaire PMNS, Red-64 vs Red-128, processeur Intel i9-11900KF.

La version 128 bits ne devient avantageuse qu'à partir de 8192 bits, car le surcoût lié à la propagation des retenues compense le gain en degré pour les tailles intermédiaires.

Mes publications

[MPV26] Nicolas Méloni, François Palma et Pascal Véron. “Equality Tests in the Polynomial Modular Number System”. In : *International Workshop on the Arithmetic of Finite Fields 2026*. Santander, Spain, mai 2026. URL : https://hal.science/hal-05586823.

[Dos+25] Fangan Yssouf Dosso, Nadia El Mrabet, Nicolas Méloni, François Palma et Pascal Véron. “Friendly primes for efficient modular arithmetic using the Polynomial Modular Number System”. In : *Journal of Cryptographic Engineering* 15.3 (sept. 2025), p. 18. DOI : 10.1007/s13389-025-00382-8.

[MPV25] Nicolas Méloni, François Palma et Pascal Véron. “Multi-precision PMNS with CIOS reduction”. In : *Selected Areas in Cryptography*. Toronto (Ontario), Canada : Springer, 2025.

[MPV23] Nicolas Méloni, François Palma et Pascal Véron. “PMNS for Cryptography : A Guided Tour”. In : *Advances in Mathematics of Communications* (2023). DOI : 10.3934/amc.2023033.

[Mra+23] Nadia El Mrabet, Loubna Ghammam, Nicolas Méloni et Emmanuel Fouotsa. “New Versions of Miller-loop Secured against Side-Channel Attacks”. In : *Arithmetic of Finite Fields*. T. 13638. Lecture Notes in Computer Science. Springer International Publishing, 2023, p. 269-287. DOI : 10.1007/978-3-031-22944-2_17.

[Mél22] Nicolas Méloni. “An Alternative Approach to Polynomial Modular Number System Internal Reduction”. In : *IEEE Transactions on Emerging Topics in Computing* (juill. 2022). DOI : 10.1109/TETC.2022.3190368.

[HMV21] Fabien Herbaut, Nicolas Méloni et Pascal Véron. “Compact variable-base ECC scalar multiplication using Euclidean addition chains”. In : *18th International Conference on Security and Cryptography (SECRYPT 2021)*. Proceedings of the 18th International Conference on Security and Cryptography - SECRYPT. Online Event, Italy, 2021, p. 531-539. DOI : 10.5220/0010551705310539.

[Dos+18] Fangan-Yssouf Dosso, Fabien Herbaut, Nicolas Méloni et Pascal Véron. "Euclidean addition chains scalar multiplication on curves with efficient endomorphism". In : *Journal of Cryptographic Engineering* (2018). doi : 10.1007/s13389-018-0190-0.

[MH16] Nicolas Méloni et M. Anwar Hasan. "Random Digit Representation of Integers". In : *23nd IEEE Symposium on Computer Arithmetic, ARITH 2016, Silicon Valley, CA, USA, July 10-13, 2016*. 2016, p. 118-125. doi : 10.1109/ARITH.2016.11.

[MH15] Nicolas Méloni et M. Anwar Hasan. "Efficient Double Bases for Scalar Multiplication". In : *IEEE Trans. Computers* 64.8 (2015), p. 2204-2212. doi : 10.1109/TC.2014.2360539.

[Has+12] M. Anwar Hasan, Nicolas Méloni, Ashkan Hosseinzadeh Namin et Christophe Nègre. "Block Recombination Approach for Subquadratic Space Complexity Binary Field Multiplication Based on Toeplitz Matrix-Vector Product". In : *IEEE Trans. Computers* 61.2 (2012), p. 151-163. doi : 10.1109/TC.2010.276.

[HMN11] M. Anwar Hasan, Nicolas Méloni et Christophe Negre. "High performance GHASH and impacts of a class of unconventional bases". In : *Journal of Cryptographic Engineering* 1.3 (2011), p. 201. issn : 2190-8516. doi : 10.1007/s13389-011-0013-z.

[Her+10] Fabien Herbaut, Pierre-Yvan Liardet, Nicolas Méloni, Yannick Teglia et Pascal Véron. "Random Euclidean Addition Chain Generation and Its Application to Point Multiplication". In : *Progress in Cryptology - INDOCRYPT 2010 - 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings*. 2010, p. 238-261. doi : 10.1007/978-3-642-17401-8_18.

[MNH10] Nicolas Méloni, Christophe Nègre et M. Anwar Hasan. "High Performance GHASH Function for Long Messages". In : *Applied Cryptography and Network Security, 8th International Conference, ACNS 2010, Beijing, China, June 22-25, 2010. Proceedings*. 2010, p. 154-167. doi : 10.1007/978-3-642-13708-2_10.

[MH09] Nicolas Méloni et M. Anwar Hasan. "Elliptic Curve Scalar Multiplication Combining Yao's Algorithm and Double Bases". In : *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. 2009, p. 304-316. doi : 10.1007/978-3-642-04138-9_22.

[Byr+07a] Andrew Byrne, Francis M. Crowe, William P. Marnane, Nicolas Méloni, Arnaud Tisserand et Emanuel M. Popovici. "SPA resistant elliptic curve cryptosystem using addition chains". In : *IJHPSA* 1.2 (2007), p. 133-142. doi : 10.1504/IJHPSA.2007.015399.

[Byr+07b] Andrew Byrne, Nicolas Méloni, Arnaud Tisserand, Emanuel M. Popovici et William P. Marnane. "Comparison of Simple Power Analysis Attack Resistant Algorithms for an Elliptic Curve Cryptosystem". In : *JCP* 2.10 (2007), p. 52-62. doi : 10.4304/jcp.2.10.52-62.

[Mél07] Nicolas Méloni. "New Point Addition Formulae for ECC Applications". In : *Arithmetic of Finite Fields, First International Workshop, WAIFI 2007, Madrid, Spain, June 21-22, 2007, Proceedings*. 2007, p. 189-201. doi : 10.1007/978-3-540-73074-3_15.

[Baj+06] J.-C. Bajard, S. Duquesne, M. Ercegovac et N. Méloni. "Residue systems efficiency for modular products summation : Application to Elliptic Curves Cryptography". In : *Advanced Signal Processing Algorithms, Architectures, and Implementations XVI, part of the SPIE Optics & Photonics 2006 Symposium*. 2006.

[BMP05a] J.-C. Bajard, N. Méloni et T. Plantard. "Efficient RNS Bases for Cryptography". In : *IMACS'2005 World Congress*. 2005.

[BMP05b] J.-C. Bajard, N. Méloni et T. Plantard. "Study of modular inversion in RNS". In : *Advanced Signal Processing Algorithms, Architectures, and Implementations XV, part of the SPIE Optics & Photonics 2005 Symposium*. 2005.

Références bibliographiques

[Pal25] François Palma. “Arithmétique modulaire pour la cryptographie à clé publique”. Thèse de doctorat dirigée par Veron, Pascal Informatique Toulon 2025. Thèse de doct. 2025.

[DRV22] Fangan Yssouf Dosso, Jean-Marc Robert et Pascal Véron. “PMNS for Efficient Arithmetic and Small Memory Cost”. In : *IEEE Transactions on Emerging Topics in Computing* 10.3 (2022), p. 1263-1277. doi : 10.1109/TETC.2021.3091046.

[Did+19] Laurent-Stéphane Didier, Fangan Yssouf Dosso, Pierre-Louis Masson et Pascal Veron. “A new tool to implement modular arithmetic with PMNS”. In : *IEEE Nordic Circuits and Systems Conference (NorCAS)*. 2019. doi : 10.1109/NORCHIP.2019.8906897.

[FLS15] A. Faz-Hernández, P. Longa et A. H. Sánchez. “Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version)”. In : *J. Cryptographic Engineering* 5.1 (2015), p. 31-52. doi : 10.1007/s13389-014-0085-7.

[Pro+15] Julien Proy, Nicolas Veyrat-Charvillon, Arnaud Tisserand et Nicolas Méloni. “Full Hardware Implementation of Short Addition Chains Recoding for ECC Scalar Multiplication”. In : *Compas : Conférence d’informatique en Parallélisme, Architecture et Système*. Lille, France, juin 2015.

[DIM08] Vassil Dimitrov, Laurent Imbert et Pradeep Mishra. “The Double-Base Number System and its Application to Elliptic Curve Cryptography”. In : *Mathematics of Computation* 77.262 (2008), p. 1075-1104.

[NP08] Christophe Nègre et Thomas Plantard. “Efficient Modular Arithmetic in Adapted Modular Number System Using Lagrange Representation”. In : *Information Security and Privacy – ACISP 2008*. T. 5107. Lecture Notes in Computer Science. Springer, 2008, p. 463-477. doi : 10.1007/978-3-540-70500-0_34.

[DI06] Christophe Doche et Laurent Imbert. “Extended Double-Base Number System with Applications to Elliptic Curve Cryptography”. In : *7th International Conference on Cryptology in India*. T. LNCS. Progress in Cryptology - INDOCRYPT 2006 4329. Kolkata, India : Springer, déc. 2006, p. 335-348. doi : 10.1007/11941378_24.

[BIP04] Jean-Claude Bajard, Laurent Imbert et Thomas Plantard. “Modular Number Systems : Beyond the Mersenne Family”. In : *Selected Areas in Cryptography – SAC 2004*. T. 3357. Lecture Notes in Computer Science. Springer, 2004, p. 159-169. doi : 10.1007/978-3-540-30564-4_11.

[Möl04] Bodo Möller. “Fractional windows revisited : improved signed-digit representations for efficient exponentiation”. In : *Proceedings of the 7th International Conference on Information Security and Cryptology*. ICISC'04. Seoul, Korea : Springer-Verlag, 2004, p. 137-153. isbn : 3540262261. doi : 10.1007/11496618_11. url : https://doi.org/10.1007/11496618_11.

[SCQ03] Francesco Sica, Mathieu Ciet et Jean-Jacques Quisquater. “Analysis of the Gallant-Lambert-Vanstone Method Based on Efficient Endomorphisms : Elliptic and Hyperelliptic Curves”. In : *Selected Areas in Cryptography*. Sous la dir. de Kaisa Nyberg et Howard Heys. Berlin, Heidelberg : Springer Berlin Heidelberg, 2003, p. 21-36. isbn : 978-3-540-36492-4.

[Ber02] Daniel J. Bernstein. “Pippenger's Exponentiation Algorithm”. Draft manuscript. 2002.

[Sol99] Jerome A. Solinas. “Generalized Mersenne Numbers”. In : *Technical Report CORR-99-39* (1999).

[Bab86] László Babai. “On Lovász' Lattice Reduction and the Nearest Lattice Point Problem”. In : *Combinatorica* 6.1 (1986), p. 1-13. doi : 10.1007/BF02579403.

[Mon85] Peter L. Montgomery. “Modular Multiplication Without Trial Division”. In : *Mathematics of Computation* 44.170 (1985), p. 519-521. doi : 10.1090/S0025-5718-1985-0777282-X.

[LLL82] Arjen K. Lenstra, Hendrik W. Lenstra et László Lovász. “Factoring Polynomials with Rational Coefficients”. In : *Mathematische Annalen* 261.4 (1982), p. 515-534. doi : 10.1007/BF01457454.

[Bra39] Alfred Brauer. “On addition chains”. In : *Bulletin of the American Mathematical Society* 45.10 (1939), p. 736-739.

Random Digit Representation of Integers

Nicolas Méloni
IMATH
Université de Toulon
Email: nicolas.meloni@univ-tln.fr

M. Anwar Hasan
Department of Electrical
and Computer Engineering
University of Waterloo
Email: ahasan@ece.uwaterloo.ca

***Abstract*—Modular exponentiation, or scalar multiplication, is core to today's main stream public key cryptographic systems. In this article we generalize the classical fractional wNAF method for modular exponentiation - the classical method uses a digit set of the form $\{1, 3, \ldots, m\}$ which is extended here to any set of odd integers of the form $\{1, d_2, \ldots, d_n\}$. We propose a general modular exponentiation algorithm based on a generalization of the frac-wNAF recoding and a new precomputation scheme. We also give general formula for the average density of non-zero therms in these representations, prove that there are infinitely many optimal sets for a given number of digits and show that the asymptotic behavior, when those digits are randomly chosen, is very close to the optimal case.**

I. Introduction

Let $k = (k_{t-1} \ldots k_0)_2$ be an integer and G a group. For $g \in G$, one can always compute g^k with at most $2\log_2(k) = 2t$ group operations using the classical `square-and-multiply` algorithm (or `double-and-add` in its additive form). There are various methods to speed up the exponentiation process, most of them are based on the initial idea from Brauer [1] that uses the 2^w-ary representation of k and performs the exponentiation accordingly. Generally speaking, those methods consider a recoding of k of the form $\sum_{i=0}^{t-1} k_i 2^i$ with k_i in some digit set D and performs the exponentiation with an adapted version of the `square-and-multiply` algorithm. Many improvements have been proposed over the years, from signed digits to sliding and fractional windows [18], [15]; see [3], [8] for a general overview. Common to those improvements is the use of digit sets containing odd integers lower than some fixed bound.

In the present work, we generalize those approaches to any set of digits containing 1. We propose a general recoding algorithm using any digit set containing 1 and give a formulae to compute the average density of non-zero terms of the recoding. In particular, we show that the digit set $\{1, 3, \ldots, 2n-1\}$ used in the fractional wNAF method is optimal among all sets of n digits in terms of average non-zero term density of the recoding, but that there are infinitely many such sets and we prove a simple criteria to determine if a set is optimal or not. We also propose a randomized exponentiation scheme in which the digit set is randomly chosen at the beginning of every exponentiation with a specific precomputation scheme. Surprisingly, we show that the average density of the resulting representations is almost as low as that of optimal recodings, so that the exponentiation computational costs are basically the same.

The rest of this paper is organized as follows: Section 2 is a brief review of the standard fractional window exponentiation method. In Section 3 we describe our new recoding algorithm, give a formulae to compute its average density of non-zero terms for any set of digits and give a criteria to distinguish optimal digit sets. In Section 4 we describe our new randomized exponentiation scheme, study its average density, propose a specific method for the precomputation phase of the exponentiation and compare the overall cost of our method to that of the fractional wNAF. In Section 5 we discuss the security provided by the new scheme and propose some comparisons with previous related works.

II. Preliminaries

Let k, g and G be as defined above. Most standard fast exponentiation schemes fall into a general framework. First find a recoding of $k = (k_{l-1} \ldots k_0)_2$ with $k_i \in \mathcal{D} \cup \{0\}$, for some set $\mathcal{D}$ of positive integers.

Then compute g^k using Algorithm 1. From this perspective, most successive improvements of the `square-and-multiply` algorithm can be viewed as new recoding schemes using different sets of digits and providing sparser and sparser recoding, that is with as few non-zero terms as possible. For instance, the computations $h \leftarrow h^2 \times h^{2k}$ and $h \leftarrow (h \times h^k)^2$ being equivalent, it is possible to restrict $\mathcal{D}$ to odd integers. When inversion is cheap, one can extend the recoding scheme using the set $\mathcal{D}$ to negative digits, thus considering the larger set $\overline{\mathcal{D}} = \mathcal{D} \cup \{-d_1, \ldots, -d_l\}$.

Finally it was shown that one is not bound to consider the 2^w-ary representation of k for a fixed w.

All put together, the various improvements give the *fractional windows* or *frac w-NAF* proposed by Möller [15]. In that case, $\mathcal{D} = \{1, 3, \ldots, 2n-1\}$. Möller proved that the average density of non-zero terms of this representation is $\frac{1}{a+1}$, where $a = W_n + \frac{2n}{2^{W_n}}$ and $W_n = \lfloor \log_2 2n - 1 \rfloor$. In this work we generalize this approach to any set $\mathcal{D}$ containing 1.

Algorithm 1 Computation of g^k

Require: An integer $k = (k_{l-1} \ldots k_0)_2$, an element g and a set of integers $\mathcal{D}$
Ensure: g^k
1: $h \leftarrow 1$
2: **for** $d \in \mathcal{D}$ **do**
3: $\quad G_d \leftarrow g^d$
4: **end for**
5: **for** $i = l-1 \ldots 0$ **do**
6: $\quad h \leftarrow h^2$
7: $\quad$ **if** $k_i \neq 0$ **then**
8: $\quad\quad h \leftarrow h \times G_{k_i}$
9: $\quad$ **end if**
10: **end for**
11: **return** h

Remark 1. *In certain contexts, it has been proven to be more efficient to consider recoding using a different base than 2. For instance, fast cubing can lead one to consider ternary or hybrid binary/ternary representations [4] and fast group endomorphism have been used in producing complex representations such as the τNAF on Koblitz curves [13].*

III. Random digit representation (RDR)

Let $\mathcal{D} = \{d_1, \ldots, d_l\}$ be a set of odd integers. Let $\overline{\mathcal{D}} = \mathcal{D} \cup \{-d_1, \ldots, -d_l\}$. We call $\mathcal{D}$-representation of k any recoding of the form $k = \sum k_i 2^i$ with $k_i \in \overline{\mathcal{D}} \cup \{0\}$. We define $N(\mathcal{D})$ as the set of all integers for which there exists a $\mathcal{D}$-representation. It is clear that any integer in $N(\mathcal{D})$ is a multiple of the gcd of $\mathcal{D}$. Thus, in order to have $N(\mathcal{D}) = \mathbb{Z}$ we must have $\gcd(\mathcal{D}) = 1$. However, the reverse does not necessarily hold. Indeed, with $\mathcal{D} = \{5, 13\}$, 1 does not have a $\mathcal{D}$-representation. On the other hand, as long as 1 belongs to $\mathcal{D}$ we are guaranteed that $N(\mathcal{D}) = \mathbb{Z}$ since its binary representation is a $\mathcal{D}$-representation for any k. In the rest of the paper, we will only consider sets on the form $\{1, d_2, \ldots, d_n\}$.

A. The recoding algorithm

Let us start with a few notations. Let $w > 0$ be an integer. For any integer x we define $p_w(x) := x \mod 2^w$. We then set $\mathcal{D}_w = p_w(\mathcal{D})$ and $\overline{\mathcal{D}}_w = \mathcal{D}_w \cup \{2^w - d : d \in \mathcal{D}_w\}$. Finally, we define $W_n = \lfloor \log_2(\max_i(d_i)) \rfloor$.

In order to define the recoding map, we first need to define, for all odd integers k, $w_{max}(k)$ as the largest integer $w \leq W_n + 2$ such that there exists a digit $d_i \in \mathcal{D}$ satisfying the following two conditions:

1) $d_i < k$,
2) $p_w(k) \in \overline{\mathcal{D}}_w$.

Finally, let the mapping $digit_{\mathcal{D}} : \mathbb{N} \to \overline{\mathcal{D}} \cup \{0\}$ be defined as follows:

- if k is even: $digit_{\mathcal{D}}(k) = 0$,
- if k is odd:
  - $W_{max} = w_{max}(k)$
  - if $p_{W_{max}}(k) \in \mathcal{D}_{W_{max}}$, $digit_{\mathcal{D}}(k) = d$ with any integer $d < k$ such that $p_w(k) = p_w(d)$
  - if $2^{W_{max}} - p_{W_{max}}(k) \in \mathcal{D}_{W_{max}}$, $digit_{\mathcal{D}}(k) = -d$ with any integer $d < k$ such that $2^w - p_w(k) = p_w(d)$.

It is worth mentioning that the map is well defined, that is to say that $digit_{\mathcal{D}}(k)$ exists for all odd k. Indeed $1 \in \mathcal{D}$ which implies that $1 \in \mathcal{D}_w$ for any w so that $W_{max} \geq 2$. The following algorithm uses the $digit_{\mathcal{D}}$ map to compute the $\mathcal{D}$-representation of any given k.

Algorithm 2 Random Digit Representation of integer k

Require: An integer k and a set $\mathcal{D} = \{1, d_2, \ldots, d_n\}$
Ensure: $k = (k_t k_{t-1} \ldots k_1 k_0)_2, k_i \in \overline{\mathcal{D}} \cup \{0\}$
1: $i = 0$
2: **while** $k \neq 0$ **do**
3: $\quad k_i = digit_{\mathcal{D}}(k)$
4: $\quad k = \frac{k - k_i}{2}$
5: $\quad i = i + 1$
6: **end while**
7: **return** $(k_{i-1} \ldots k_0)$

Remark 2. *If $\mathcal{D} = \{1, 3, \ldots, 2n-1\}$ we obtain exactly the* fractional windows *recoding.*

Example 1. *Let $k = 31415$ and $\mathcal{D} = \{1, 3, 23, 27\}$. We have $W_n = 4$, $\mathcal{D}_2 = \{1, 3\}$, $\mathcal{D}_3 = \{1, 3, 7\}$, $\mathcal{D}_4 = \{1, 3, 7, 11\}$ and $\mathcal{D}_5 = \mathcal{D}_6 = \mathcal{D}$. Algorithm 2 applied to k gives:*

1) *k is odd, $k \mod 2^6 \equiv 55 \equiv -19$ and $k \mod 2^5 \equiv 23$, so $W_{\max} = 5$ and $digit_{\mathcal{D}}(k) = 23$,*
2) *$k_0 = 23$ and $k \leftarrow \frac{k-23}{2} = 15696$.*
3) *$k_1 = k_2 = k_3 = k_4 = 0$ and $k \leftarrow \frac{k}{2^4} = 981$.*

4) k *is odd,* $k \mod 2^6 \equiv 21 \equiv -43$*,* $k \mod 2^5 \equiv 21 \equiv -11$ *and* $k \mod 2^4 \equiv 5 \equiv -11$ *thus* $W_{\max} = 4$ *and* $digit_{\mathcal{D}}(k) = -27$ *as* $p_4(27) = 11$*,*
5) $k_5 = -27$ *and* $k \leftarrow \frac{k+27}{2} = 504$*,*
6) $k_6 = k_7 = k_8 = 0$ *and* $k \leftarrow k/8 = 63$*,*
7) $k \mod 2^6 \equiv -1$ *so* $W_{max} = 6$*,* $k_9 = -1$ *and* $k \leftarrow \frac{k+1}{2}$
8) $k_{10} = k_{11} = k_{12} = k_{13} = k_{14} = 0$*,* $k \leftarrow \frac{k}{2^5}$
9) $k = k_{15} = 1$.

Finally we obtain $k = (1, 0, 0, 0, 0, 0, -1, 0, 0, 0, -27, 0, 0, 0, 0, 23)_2$.

B. Average density

Theorem 1. *Let* k *be an integer and* $\mathcal{D} = \{1, d_2, \ldots, d_n\}$ *a set of digits. For all* $w \geq 2$ *we define* $D(w) = \frac{\#\overline{\mathcal{D}}_w}{2^{w-1}}$*. Then the asymptotic average density of non-zero terms achieved by the random digit representation is* $\frac{1}{a_{\mathcal{D}}+1}$*, where*

$$a_{\mathcal{D}} = 2D(W_n + 2) + \sum_{w=2}^{W_n+1} D(w).$$

Proof. Let k be an odd integer greater than d_n and $d = digit_{\mathcal{D}}(k)$. We first want to evaluate the probability $P(w)$ that $w_{max}(k) = W_{max} = w$ for every $w \leq W_n + 2$. As given in Theorem 1, $D(w)$ is the probability that the residue of a given odd integer modulo 2^w lies in $\overline{\mathcal{D}}_w$. By construction we have that $P(W_n+2) = D(W_n+2)$. For lower values of w, we must evaluate the probability that a residue is in $\overline{\mathcal{D}}_w$ but not in $\overline{\mathcal{D}}_{w+1}$. It is clear that $(d \mod 2^{w+1} \in \overline{\mathcal{D}}_{w+1}) \Rightarrow (d \mod 2^w \in \overline{\mathcal{D}}_w)$ which implies that, for $w < W_n+2$, $P(w) = D(w)-D(w+1)$.

Now, from the definition of W_{max}, we have $(k-d) \equiv 0 \mod 2^{W_{max}}$. In other words, k can be written in the form

$$k = (k'_{t'} \ldots k'_{W_{max}} 0 \ldots 0 d)_2, k_i \in \{0, 1\}.$$

If $W_{max} < W_n + 2$, still by definition, $k'_{W_{max}} \neq 0$. However, if $W_{max} = W_n$ it is necessary to estimate the average number of consecutive zeros starting from $k'_{W_{max}}$. Classically, for an arbitrary long sequence of random bits, this number is 1. As a consequence, we can finally write the value of $a_{\mathcal{D}}$, the average number of zeros following a non-zero digit in the RDR of an integer in terms of $D(w)$:

$$\begin{aligned} a &= (W_n + 2)P(W_n + 2) \\ &\quad + W_n P(W_n + 1) + \cdots + 2P(3) + P(2) \\ &= (W_n + 2)D(W_n + 2) \\ &\quad + W_n(D(W_n + 1) - D(W_n + 2)) + \ldots \\ &\quad + (D(2) - D(3)) \\ &= 2D(W_n + 2) + D(W_n + 1) + \cdots + D(3) + D(2) \end{aligned}$$

□

Example 2. *For set* $B_n = \{1, 3, \ldots, 2n-1\}$ *we have* $W_n = \lfloor \log_2(2n-1) \rfloor$*. It is easy to see that, on one hand* $D(W_n + 1) = D(W_n) = \cdots = D(2) = 1$ *and on the other hand* $D(W_n + 2) = (2n)/2^{W_n+1}$*, so that*

$$a_{\mathcal{D}} = W_n + \frac{2n}{2^{W_n}},$$

which corresponds to the standard result on the average density of the frac-w*NAF representation.*

Example 3. *With* $\mathcal{D} = \{1, 3, 23, 27\}$*, we have* $\overline{\mathcal{D}}_2 = \{1, 3\}$*,* $\overline{\mathcal{D}}_3 = \{1, 3, 5, 7\}$*,* $\overline{\mathcal{D}}_4 = \{1, 3, 5, 7, 9, 11, 13, 15\}$*,* $\overline{\mathcal{D}}_5 = \{1, 3, 5, 9, 23, 27, 29, 31\}$*. From Theorem 1 we obtain that*

$$a_{\mathcal{D}} = 2 \times \frac{1}{4} + \frac{1}{2} + 1 + 1 + 1 = 4.$$

In this case the RDR has a density of $\frac{1}{5}$*, the same as the* 4*-NAF representation, where* $\mathcal{D} = \{1, 3, 5, 7\}$*. In other words, we see that we can achieve the same density with different sets of digits of the same cardinal.*

C. Optimal digit sets

The random digit representation obtained from Algorithm 2 is a generalization of the frac-wNAF recoding. Now we show that, for a given number of digits n, the lowest asymptotic density of non-zero terms is that of the corresponding frac-wNAF recoding but can be obtained with infinitely many digit sets.

Definition 1. *Let* $n > 0$ *and* $\mathbb{D}_n$ *be the set of all sets of odd integers of the form* $\{1, d_2, \ldots, d_n\}$*. Then* $\mathcal{D} \in \mathbb{D}_n$ *is an* ***optimal digit set*** *if*

$$a_{\mathcal{D}} = \max_{\mathcal{D}' \in \mathbb{D}_n} (a_{\mathcal{D}'}).$$

Theorem 2. *Let* $n > 0$*,* $w_n = \lfloor \log_2 n \rfloor$ *and* $\mathcal{D} \in \mathbb{D}_n$*. If* $\mathcal{D}$ *is an optimal digit then*

$$a_{\mathcal{D}} = w_n + \frac{n}{2^{w_n}} + 1.$$

Proof. From Theorem 1 we know that

$$a_{\mathcal{D}} = \underbrace{2D(W_n+2)}_{A} + \underbrace{\sum_{i=w_n+3}^{W_n+1} D(i)}_{B} + \underbrace{\sum_{i=2}^{w_n+2} D(i)}_{C}.$$

First we remark that $\forall w \geq 2, \#\overline{\mathcal{D}}_w \leq \min(2^{w-1}, 2n)$. By definition of w_n, we have $2n < 2^{w_n+2}$, thus $\#\overline{\mathcal{D}}_w \leq 2n$ for all $w \geq w_n + 3$.

So we have

$$A = 2\frac{2n}{2^{W_n+1}} = \frac{n}{2^{W_n-1}},$$

$$\begin{aligned} B &\leq \sum_{i=w_n+3}^{W_n+1} \frac{2n}{2^{i-1}} \\ &\leq \frac{4n}{2^{w_n+2}} \times (1 - \frac{1}{2^{W_n-w_n-1}}) \\ &\leq \frac{n}{2^{w_n}} - \frac{n}{2^{W_n-1}}, \end{aligned}$$

$$C \leq \sum_{i=2}^{w_n+2} 1 \leq w_n + 1.$$

So we have shown that $a_{\mathcal{D}} \leq w_n + \frac{n}{2^{w_n}} + 1$. To conclude we just need to remark that the inequality becomes an equality for $\mathcal{D} = \{1, 3, \ldots, 2n-1\}$. Indeed if n is a power of two then $n = 2^{w_n}$, $W_n = w_n$ and $a_{\mathcal{D}} = w_n + 2 = w_n + \frac{n}{2^{w_n}} + 1$. If n is not a power of two then $W_n = \lfloor \log_2(2n-1) \rfloor = w_n + 1$ and $2^{W_n} < 2n - 1 < 2^{W_n+1}$ so that $D(2) = \cdots = D(W_n + 1) = 1$ and $2D(W_n + 2) = \frac{4n}{2^{W_n+1}}$, which leads to $a_{\mathcal{D}} = W_n + \frac{n}{2^{W_n-1}} = w_n + 1 + \frac{n}{2^{w_n}}$.

□

Corolary 1. *Let $n > 0$, $w_n = \lfloor \log_2 n \rfloor$ and $\mathcal{D} \in \mathbb{D}_n$. Then $\mathcal{D}$ is an optimal digit set if and only if:*

$$\#\overline{\mathcal{D}}_{w_n+3} = 2n \text{ and } \#\overline{\mathcal{D}}_{w_n+2} = 1.$$

Proof. From the proof of Theorem 2, we know that $\mathcal{D}$ is optimal if and only if for all $i \geq w_n + 3, \#\overline{\mathcal{D}}_i = 2n$ and for all $2 \leq i \leq w_n + 2, D(i) = 1$ i.e. $\#\overline{\mathcal{D}}_i = 2^{i-1}$. To prove this corollary it is enough to prove that, on the one hand, $\#\overline{\mathcal{D}}_{w_n+3} = 2n \Rightarrow \#\overline{\mathcal{D}}_w = 2n$ for all $w \geq w_n + 3$ and that, on the other hand, for all $2 < i$, $(D(i) = 1) \Rightarrow (D(i-1) = 1)$.

Both properties result from the fact that $\forall w \geq 2$, $\#\overline{\mathcal{D}}_{w+1} \leq 2\left(\#\overline{\mathcal{D}}_w\right)$. Indeed, any $d \in \#\overline{\mathcal{D}}_{w+1}$ can be written as $d' + b2^w$ with $b \in \{0, 1\}$ and $d' \in \overline{\mathcal{D}}_w$ so that there are at most twice as many elements in $\overline{\mathcal{D}}_{w+1}$ than in $\overline{\mathcal{D}}_w$. It implies that $D(w+1) = \frac{\#\overline{\mathcal{D}}_{w+1}}{2^w} \leq 2\frac{\#\overline{\mathcal{D}}_w}{2^w} \leq D(w)$ which in turn implies that $(D(w) = 1) \Rightarrow (D(w-1) = 1)$. Also, $\forall w \leq w_n + 3$, $2n \leq \#\overline{\mathcal{D}}_{w_n+3} \leq \#\overline{\mathcal{D}}_w \leq \min(2n, 2^w) \leq 2n$, which concludes the proof. □

Example 4. *Using the previous corollary it is easy to show that the previously seen set $\mathcal{D} = \{1, 3, 23, 27\}$ is optimal. We have $n = 4$ and $w_n = 2$, so we just need to check that $\#\overline{\mathcal{D}}_4 = \#\{1, 3, 5, 7, 9, 11, 13, 15\} = 2^3$ and $\#\overline{\mathcal{D}}_5 = \#\{1, 3, 5, 9, 23, 27, 29, 31\} = 2n$.*

Remark 3. *Corollary 1 directly implies that, for all n, there are infinitely many optimal digit sets. For instance, all sets of the form*

$$\mathcal{D} = \{1, 3, \ldots, 2n-3\} \cup \{2n - 1 + 2^w\}$$

with $w \geq w_n + 3$ are optimal. More generally, for a given number n, all sets of the form

$$\{1, 3 + t_2 2^{w_n+3}, \ldots, 2n - 1 + t_n 2^{w_n+3}\}$$

with $(t_2, \ldots, t_n) \in \mathbb{N}^{n-1}$ are optimal.

IV. Randomized exponentiation scheme

Algorithm 2 allows us to compute the random digit representation (RDR) of an integer k using any set of digits $\mathcal{D}$ as long as 1 belongs to it. We can now integrate this algorithm into a general randomized exponentiation scheme. Let g be a group element, k an exponent and m and l two integers satisfying $l \leq (m+1)/2$. One can compute g^k using the following scheme:

1) randomly choose $l-1$ odd integers $\{d_2, \ldots, d_l\}$ among $\{3, \ldots, m\}$,
2) compute RDR of k using Algorithm 2 and set $\mathcal{D} = \{1, d_2, \ldots, d_l\}$,
3) compute g^k using Algorithm 1.

One obvious advantage of such a scheme is that it provides an added resistance to differential power analysis alongside with other schemes such as point blinding. Indeed, from one exponentiation to the other, it ensures that the operation flow will be completely different.

A. Average case and the urn problem

From Theorem 1 we can compute the asymptotic density of the RDR for a given set of digits $\mathcal{D}$. One natural question is what is the average behavior of this density when the digits are chosen randomly. Let m be a parameter and let us consider $B_m = \{1, 3, \ldots, m\}$. We want to evaluate the average density of the RDR when we randomly choose l integers from B_m. To apply our

theorem, we need to compute the value of $D(w)$ for all needed w and thus the cardinal of $\overline{\mathcal{D}_w}$.

First, let $N = \frac{m+1}{2}$ and $W_N = \lfloor \log_2 N \rfloor$, we note that by definition all $d \in \mathcal{D}$ are smaller than 2^{W_N+1}, which implies that $(2^{W_N+2} - d) \neq d$ and thus $\overline{\mathcal{D}_{W_N+2}} = 2l$. Evaluating $D(w)$ for smaller value of w becomes a harder problem. If an integer d is in $\mathcal{D}$, then all integers of the form $d + i2^w$ and $i2^w - (d \mod 2^w)$ do not add up anything to the cardinal of $\overline{\mathcal{D}_w}$. In short, we need to evaluate the number of equivalence classes of the set $\mathcal{D}$ with respect to the relation $\mathcal{R}_w$ define for all $w \leq W_N + 1$ by:

$$x \mathcal{R}_w y \Leftrightarrow x \equiv y \mod 2^w, \text{ or } x \equiv -y \mod 2^w.$$

A simple way to consider this problem is to see it as an urn problem. Let us consider N balls corresponding to each integer of our initial integer set B_m. For a given w, define $C_w = 2^{w-2}$ as the number of equivalence classes with respect to $\mathcal{R}_w$, each pair $(i \mod 2^w, 2^w - (i \mod 2^w))$ for i in $\{1, 3, \ldots, 2^{w-1} - 1\}$ being a representative of one of them. Finally, define E_w^i as the number of representatives of each of those classes. Our problem consists of drawing l balls (without replacement) in an urn containing N balls of C_w different colors and evaluate the average number of different colors obtained. Let $M(l, c, N)$ be the number of different drawings, without replacement, of l balls among N having exactly c different colors. Then the probability that we obtain exactly c colors is

$$P[X = c] = \frac{M(l, c, N)}{\binom{N}{l}},$$

where X is a random variable corresponding to the number of drawn colors. A theorem from Walton [21] shows that $M(l, c, N)$ can be computed by developing the polynomial

$$F_w(X, Y) = \prod_{c=1}^{C_w} \left(Y\{(1 + X)^{E_w^i} - 1\} + 1 \right).$$

Indeed, he proves that

$$F_w(X, Y) = \sum_c \sum_l M(l, c, N) X^l Y^c.$$

It is then possible to compute $D(w)$ for practical values of w and finally obtain the density of the RDR representation for a given number of drawings. In this work we have computed it for $w \leq W \leq 10$. Results are summarized in Table I. In order to maximize the number of possible digit sets, we fix $t = \left\lfloor \frac{1+m}{4} \right\rfloor$ as the number of drawn balls. The wNAF column corresponds to the (optimal) density of the wNAF representation using as many digits as the RDR. We observe that the difference between the two methods is relatively small. The general loss in terms of density is less than half a bit.

m	$\#\mathcal{D}$	RDR	wNAF
7	2	3.833	4
15	4	4.771	5
31	8	5.728	6
63	16	6.706	7
127	32	7.695	8
255	64	8.689	9
511	128	9.686	10
1023	256	10.69	11

TABLE I
INVERSES OF THE DENSITY OF THE RDR AND wNAF USING $\left\lfloor \frac{m+1}{4} \right\rfloor$ DIGITS

B. Precomputation scheme

The first step of our exponentiation scheme consists of computing $g^{\pm d_i}$ for $d_i \in \mathcal{D}$. Finding an efficient way to perform this computation is somehow equivalent to finding a short addition chain computing the set $\mathcal{D}$. The problem is trivial when $\mathcal{D} = B_m$ as the chain $(1, 2, 3, 5, 7, 9, \ldots, m)$ is the shortest possible. However when the d_i's are randomly chosen it is harder to find an optimal chain. The naive approach consists of using the previous chain and only keep the needed elements. It requires the computation of $m/2$ integers when only $m/4$ could be needed in the best case. Here we propose a method to find shorter addition chains than the naive approach, inspired by Pippinger algorithm [17]. It is a very general algorithm that allows the computation of multiple powers of a group element. Our case however does not require such a general method. In particular, we know that

- we need to compute the g^{d_i}'s for small values of d_i,
- all d_i's are odd,
- the cardinal of $\mathcal{D}$ is fixed to $\lfloor \frac{m+1}{4} \rfloor$.

Thus we can use a more simple method described next. Let $0 < b < W$ be a parameter and define $q = \lfloor \frac{m}{2^b} \rfloor$:

1) compute $X = \{g, g^2, g^3, g^5, g^7, \ldots, g^{2^b-1}\}$,
2) compute $Y = \{g^{2^b}, g^{2 \cdot 2^b}, , g^{3 \cdot 2^b}, \ldots, g^{q \cdot 2^b}\}$,
3) for all $d_i \notin X$, compute $g^{d_i} = xy, (x, y) \in X \times Y$.

The computation cost is 2^{b-1} group operations to compute X, q group operations to compute Y and at most $\#\mathcal{D}$ group operations to obtain the g^{d_i}'s. The total cost is thus bounded by $2^{b-1} + \lfloor \frac{m}{2^b} \rfloor + \#\mathcal{D}$ group operations. In the end, we save many operations in the later stage depending on parameter b. Indeed, the proportion of integer d_i in X is given by $\frac{2^b}{m+1}$. So for instance, with $\frac{m+1}{4}$ randomly chosen numbers, our method saves on average 2^{b-2} group operations.

In order to give a clearer view of the computational cost of our generalized recoding using our precomputation scheme we have evaluated the number of group operations and the overall density of non-zero terms of both the RDR and the frac-wNAF in various situations. To do so, for different values of the number of digits n we have:

1) randomly generated up to 1000 random digit sets of size n,
2) randomly generated up to 1000 random optimal digit sets of size n,
3) randomly generated 1000 1024-bit integers,
4) computed the inverse of the average density $a_{\mathcal{D}}+1$ and the number of group operations of the RDR for each digit set in terms of squarings (S) and multiplications (M)
5) done the same for the frac-wNAF using digit set $\mathcal{D} = \{1, 3, \ldots, 2n-1\}$.

The results of our experiments are summarized in Table II. It clearly shows that the overhead cost caused by the use of a randomized digit set is negligible compared to the cost of the frac-wNAF recoding. Also, as expected, restricting the RDR to optimal digit sets leads to even faster exponentiations with recodings with the same density of non-zero terms as those obtained with the frac-wNAF method.

Digit set size	Method	$a_{\mathcal{D}}+1$	group operation count
8	RDR	5.701	1023S+191M
	Opt. RDR	5.970	1024S+183M
	frac-wNAF	5.997	1023S+178M
16	RDR	6.666	1023S+175M
	Opt. RDR	6.960	1023S+169M
	frac-wNAF	6.962	1022S+161M
24	RDR	7.209	1023S+175M
	Opt. RDR	7.454	1023S+167M
	frac-wNAF	7.454	1023S+160M
32	RDR	7.634	1023S+175M
	Opt. RDR	7.940	1023S+170M
	frac-wNAF	7.950	1022S+160M
48	RDR	8.178	1023S+190M
	Opt. RDR	8.434	1023S+180M
	frac-wNAF	8.440	1022S+168M
64	RDR	8.692	1023S+207M
	Opt. RDR	8.922	1023S+196M
	frac-wNAF	8.940	1022S+177M

TABLE II
DENSITY AND GROUP OPERATION COUNTS OF THE RANDOM DIGIT REPRESENTATION AND FRACTIONAL wNAF RECODINGS FOR 1024-BIT EXPONENTS.

V. SIDE-CHANNEL SECURITY

The main interest of randomizing the exponentiation process is to provide an added resistance to side-channel attacks via algorithmic countermeasures. In this section we discuss the security of our method against differential and simple side channel attacks.

A. Differential attacks

Differential power analysis aim at finding the secret key by analyzing power traces of several executions of the same computation, depending on that secret. Recent works have proven to be able to defeat various randomization methods such as the Binary Signed Digit randomization [5] or Liardet-Smart randomized algorithm [19] for instance. The main weakness of those methods is little randomness they actually provide despite the apparent variety of recoding they provide. In particular, Fouque et al. stress that such randomization techniques fail because they do not provide a sufficiently large number of possible local internal states and transitions from that states, making them vulnerable to collision attacks. Another important remark is that those attacks use the facts that the set of digits is known in advance. For instance, the hidden Markov model cryptanalysis used against the Oswald-Aigner randomized exponentiation makes a direct use of the knowledge of the three possible digits $0, 1$ and -1 to produce the probabilistic state machines used in the cryptanalysis [12].

From that perspective, our method is the first to provide two levels of protection against those attacks. First, the fact that the digit set is randomly chosen prevents a traditional attackers to mount any attack previously mentioned as they directly use the fact that the digit set is known in advance. In order to mount an attack, all possible digit sets must be considered and dealt with in parallel. For that matter, the size of the set can be seen as a security parameter. For instance, using an eight digit recoding (seven of them randomly chosen from $\{3, \ldots, 31\}$), we obtain a total of 6435 possible digit

sets and more than 3×10^8 for a sixteen digit recoding. On top of that, the recoding algorithm itself provides randomness as when several digits satisfy the appropriate congruence one is chosen at random. It means that for a given digit set, any integer can have many different recodings.

B. Simple power analysis attacks

Simple power analysis attacks aim at obtaining information on the secret key using a single trace. Typically, being able to distinguish squarings from multiplications allows an adversary to recover the secret exponent of any exponentiation using the `square-and-multiply` algorithm. From that perspective, a randomization process does not, by itself, provide any protection. To ensure that an algorithm is secure against such attacks, the standard way is to make the computation as regular as possible. It can be done at the algorithm level, using the Montgomery ladder for instance, or at the group algorithmic level, for example by using unified formulae in the context of elliptic curve cryptography or using block atomicity [2].

Our algorithm clearly will not have a regular behavior, however that does not signifies that it is vulnerable to simple power analysis attacks. Indeed, one obviously implements it using one of the previously mentioned arithmetic level countermeasures, but even without them, being able to distinguish between squaring and multiplication does not provide much information on the secret key. Even if the sequence of multiplications and squarings performed by the algorithm is given, one still has to guess which digit has been used at each step. For instance, for a 128-bit security, and therefor a 256-bit exponent, and an eight digit recoding (that is with parameter $m = 31$ in Table 1), there will be on average 44.6 non-zero digits in the recoding corresponding to so many multiplications in the trace. As there are 8 possible choices for each of these multiplications the total number of combinations is roughly $8^{44.6} \sim 2^{133.8}$. This has to be multiplied by the number of possible digits sets (6435). Trying to recover the original key from an exhaustive search would be more difficult than attacking the system itself.

C. Related works

Randomization is a standard way to provide security against differential side-channel attacks. In particular, several randomized exponentiation algorithms have been proposed [7], [16], [11], [20], [9] but the security offered by those methods remains in general uncertain. For example, randomized recodings proposed by Ha and Moon [7] or Oswald and Aigner [16] have been defeated due to little local variation of the data. It was exploited through collision detection [5] or more generally using the hidden Markov model cryptanalysis [12]. In a similar way, some randomization techniques focus on the management of the window in sliding window algorithm [10], [14] and successful attacks have been mounted against some of them [19]. More generally, the hidden Markov model attack seems to be a threat to all of them. Finally, very recently Guérini, Imbert and Winterhalter proposed a new recoding method based on exact covering systems of congruences [6]. In some way it is the closest approach to ours as it provides several possible digits, however fixed in advance, at every step of the recoding and seem to provide more randomness and security than the previous approaches. It is also interesting to remark that both methods can be combined as they rely on different aspects of the exponent recoding.

VI. Conclusions

In this work we have proposed a generalization of the traditional fractional wNAF recoding. Our algorithm allows the computation of the representation of an integer using a set of any digits that has 1 in it. We also have given a general formulae to compute the average density of non-zero terms of such representations. Two important results that we have shown are that the optimal density can be achieved by infinitely many sets and that there is a simple criteria to distinguish them. We also studied the average density obtained when digits are chosen at random from a given set. The surprising result is that recodings obtained from random digit sets almost have the same density of non-zero terms as optimal ones. Combined with our new precomputation scheme we have proposed a randomized exponentiation scheme and suggested that it could be used to provide some additional protection against differential power analysis attacks for almost no additional cost.

References

[1] A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.

[2] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[3] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Cryptography*. Chapman & Hall, 2006.

[4] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computations*, 77, 2008.

[5] Pierre-Alain Fouque, Frédéric Muller, Guillaume Poupard, and Frédéric Valette. Defeating countermeasures based on randomized bsd representations. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *Lecture Notes in Computer Science*, pages 312–327. Springer Berlin Heidelberg, 2004.

[6] Eleonora Guerrini, Laurent Imbert, and Théo Winterhalter. Randomizing scalar multiplication using exact covering systems of congruences. Cryptology ePrint Archive, Report 2015/475, 2015.

[7] Jae-Cheol Ha and Sang-Jae Moon. Randomized signed-scalar multiplication of ecc to resist power attacks. In *Revised Papers from the 4th International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '02, pages 551–563. Springer-Verlag, 2003.

[8] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[9] M.A. Hasan. Power analysis attacks and algorithmic approaches to their countermeasures for koblitz curve cryptosystems. *Computers, IEEE Transactions on*, 50(10):1071–1083, Oct 2001.

[10] Kouichi Itoh, Jun Yajima, Masahiko Takenaka, and Naoya Torii. Dpa countermeasures by improving the window method. In *Cryptographic Hardware and Embedded Systems - CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 303–317. Springer Berlin Heidelberg, 2003.

[11] Tetsuya Izu and Tsuyoshi Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In *Public Key Cryptography*, volume 2274 of *Lecture Notes in Computer Science*, pages 280–296. Springer Berlin Heidelberg, 2002.

[12] Chris Karlof and David Wagner. Hidden markov model cryptanalysis. In *Cryptographic Hardware and Embedded Systems - CHES 2003*, volume 2779 of *Lecture Notes in Computer Science*, pages 17–34. Springer Berlin Heidelberg, 2003.

[13] N. Koblitz. Cm-curves with good cryptographic properties. In *Advances in Cryptology - CRYPTO*, volume 576 of *LNCS*, page 279. Springer, February 1992.

[14] P.-Y. Liardet and N. P. Smart. Preventing SPA/DPA in ECC systems using the Jacobi form. In *Cryptographic Hardware and Embedded Systems - CHES*, pages 391–401. Springer-Verlag, 2001.

[15] B. Möller. Improved techniques for fast exponentiation. In Springer Berlin / Heidelberg, editor, *Information Security and Cryptology — ICISC 2002*, volume 2587 of *LNCS*, pages 298–312, 2003.

[16] Elisabeth Oswald and Manfred Aigner. Randomized addition-subtraction chains as a countermeasure against power attacks. In *Proceedings of the Third International Workshop on Cryptographic Hardware and Embedded Systems*, CHES '01, pages 39–50. Springer-Verlag, 2001.

[17] N. Pippenger. On the evaluation of powers and related problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 258–263, Oct 1976.

[18] E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.

[19] Colin D. Walter. Breaking the liardet-smart randomized exponentiation algorithm. In *Proceedings of the 5th Conference on Smart Card Research and Advanced Application Conference - Volume 5*, CARDIS'02, pages 7–7, Berkeley, CA, USA, 2002. USENIX Association.

[20] ColinD. Walter. Mist: An efficient, randomized exponentiation algorithm for resisting power analysis. In *Topics in Cryptology — CT-RSA 2002*, volume 2271 of *Lecture Notes in Computer Science*, pages 53–66. Springer Berlin Heidelberg, 2002.

[21] Gerald S. Walton. The number of observed classes from a multiple hypergeometric distribution. *Journal of the American Statistical Association*, 81(393):pp. 169–171, 1986.

Noname manuscript No.
(will be inserted by the editor)

Euclidean addition chains scalar multiplication on curves with efficient endomorphism

Yssouf Dosso · Fabien Herbaut · Nicolas Méloni · Pascal Véron

This is a pre-print of an article published in "Journal of Cryptographic Engineering". The final authenticated version is available online at: https://doi.org/10.1007/s13389-018-0190-0.

the date of receipt and acceptance should be inserted later

Abstract Random Euclidean addition chain generation has proven to be an efficient low memory and SPA secure alternative to standard ECC scalar multiplication methods in the context of fixed base point [21]. In this work, we show how to generalize this method to random point scalar multiplication on elliptic curves with an efficiently computable endomorphism. In order to do so we generalize results from [21] on the relation of random Euclidean chains generation and elliptic curve point distribution obtained from those chains. We propose a software implementation of our method on various platforms to illustrate the impact of our approach. For that matter, we provide a comprehensive study of the practical computational cost of the modular multiplication when using Java and C standard libraries developed for the arithmetic over large integers.

Keywords Addition chains · Co-Z arithmetic · scalar multiplication · GLV · Android

1 Introduction

Let p be a prime number. An elliptic curve in short Weierstrass form over a finite prime field $\mathbb{F}_p$ is defined by $E(\mathbb{F}_p) = \{(x, y) \in \mathbb{F}_p \times \mathbb{F}_p | y^2 = x^3 + ax + b\} \cup \mathcal{O}$, with $a, b \in \mathbb{F}_p$ satisfying $4a^3 + 27b^2 \neq 0$ and $\mathcal{O}$ being called the point at infinity. The set $E(\mathbb{F}_p)$ is an abelian group with an efficiently computable group law. The main operation in elliptic curve cryptography is scalar multiplication, that is the computation of kP, where P is a prime order point on a curve and k is an integer. Optimizing this operation is directly linked to the problem of finding a short addition chain computing the integer k. The most common way to find such chains relies on the classical *double-and-add* algorithm and its many variants and improvements [5,34,29,30]. Another approach consists in using a rather different family of chains, the Euclidean Addition Chains (EAC). If $C = (c_1, \ldots, c_l)$ is an EAC computing k, one can compute kP in l differential point additions, that is additions for which the difference of the two summands is already known. On elliptic curves in short Weierstrass form it provides an efficient, low memory and simple side channel attack (SSCA) resistant method to perform scalar multiplication when combined with Co-Z arithmetic, as long as one is capable of finding a chain of small length [28]. However, given a large integer k, it appears to be quite time consuming to find suitable chains. A natural way to bypass that issue is to randomly generate a small EAC and to consider the corresponding integer. However, in that case, one does not fully control the distribution of the corresponding points. To obtain a proven security of n bits, that is to say to be able to guarantee that one can generate 2^n different chains computing 2^n different points, one can work with chains of length n but has to pre-compute the pair $(F_{n+2}P, F_{n+3}P)$, where $\{F_n\}$ is the Fibonacci sequence, and work on larger fields than standard methods. Those constraints limit the use of this approach to the case of fixed base point scalar multiplication [21].

Y. Dosso, N. Méloni, P. Véron
Institut de Mathématiques de Toulon
Université de Toulon, France
E-mail: dosso@univ-tln.fr
E-mail: meloni@univ-tln.fr
E-mail: veron@univ-tln.fr

F. Herbaut
Université de Nice Sophia Antipolis, France
Institut de Mathématiques de Toulon
Université de Toulon, ESPE Nice-Toulon
E-mail: herbaut@unice.fr

In this paper, we propose to generalize that previous work to the case of random base point scalar multiplication on elliptic curves with an efficiently computable endomorphism. Moreover, we want to derive from this generalization a practical implementation which fits the following constraints.

SSCA resistance: we need to design a regular and constant-time algorithm to be protected against simple side channel attacks.

Cache timing attack resistance: the execution flow (sequence of instructions) must be independent from the key used in order to avoid recent cache instructions attacks [1]. Data loaded into cache must also be independent from the key in order to avoid data cache attacks [3].

Low memory: we want to minimize the number of registers needed to store the coordinates of the various points involved in the computation of kP. For resource constrained devices (like IoT devices), it is of utmost importance to design a low memory algorithm with little impact on the performances of the scalar multiplication operation.

Non specific libraries dependencies: in order to manage arithmetic over large integers, we only consider general purpose multi-precision libraries with long term support (BigInteger library for Java-based platforms and GNU Multiple Precision library for other platforms).

Curves with one efficient endomorphism: in order to design an efficient algorithm, we consider curves with one endomorphism. Taking into account our memory usage constraint, we focus on curves with exactly one endomorphism. Indeed, the use of extra endomorphisms leads to extra precomputation stages, and so to extra memory usage. As an example, a secure implementation of a scalar multiplication algorithm using a curve with one endomorphism needs to precompute two points. For a curve, with two endomorphisms, the same implementation needs eight points [12].

Curves over $\mathbb{F}_p$: to be compliant with actual elliptic curve cryptography standards, we focus on the case of an elliptic curve defined over $\mathbb{F}_p$. Moreover, to be more resource friendly, we do not want to have to deal both with arithmetic modulo p and arithmetic modulo p^2.

In the sequel, we first recall the necessary background on EAC, Co-Z arithmetic and elliptic curves with fast endomorphism (Section 2). Then we generalize the results from [21] on the distribution of integers computed by an EAC starting from any pair of points (aP, bP) when P is fixed (Corollary 1). Finally we consider the case of scalar multiplication with a random base point on curves with a fast endomorphism ϕ (Proposition 4). We show that under some assumptions we can guarantee a given security when starting from a pair of points $(P, \phi(P))$. We derive from those results a new scalar multiplication scheme on curves with a fast endomorphism (Section 4). The complexity analysis of this scheme shows that it can be competitive with state of the art methods depending on the relative costs of modular multiplication on fields of various sizes. To illustrate our point, we propose software implementations on various platforms using standard libraries for arithmetic over large integers. We discuss the efficiency of our method from a speed and memory consumption point of view (Section 5, 6, 7 and 8).

2 Background on ECC scalar multiplication

2.1 Curve with efficient endomorphism

In 2001, Gallant, Lambert and Vanstone introduced a new approach to speed up scalar multiplication on elliptic curves with an efficiently computable endomorphism [15], the so called GLV method. Let E be an elliptic curve over $\mathbb{F}_p$ such that $\#E(\mathbb{F}_p) = N \times h$, where N is a large prime and h a small co-factor (i.e. 1, 2 or 4). Let ϕ be a non trivial endomorphism. Then there exists λ such that for all points P of order $N, \phi(P) = \lambda P$. Now let us consider a scalar $k \in [1, N-1]$. It has been proven that one can always find k_1, k_2 such that $k \equiv k_1 + k_2\lambda \mod N$ and $\max\{|k_1|, |k_2|\} \leq c\sqrt{N}$ for some computable constant c [11]. On curves with such an endomorphism, kP can be computed by performing a multi-scalar multiplication saving half the point doublings in exchange of a few point additions. The standard method consists of storing P, $\phi(P)$, $P + \phi(P)$ and $P - \phi(P)$ (in addition of the current point) but is vulnerable to simple side channel attack. To prevent such attacks, the most recent implementations use a combination of Least Significant Bit set representation and sign alignment [12]. It has the advantages to make the scalar multiplication regular (one doubling and one addition per scalar bit) and to reduce the storage requirement as only the points P and $P + \phi(P)$ need to be stored. The GLV method has later been extended to a larger set of curves defined over $\mathbb{F}_{p^2}$ [14,32,19,27] which are endowed by more than one endomorphism. In this case additional performance gains can be achieved, whereas it implies the need of more memory.

2.2 Co-Z arithmetic on elliptic curves

Fast elliptic curve computations has become an important research area over the past years. Many formulas,

coordinate systems or curve shapes have been proposed in order to implement the associated group law. For a comprehensive overview, one can refer to [8,20]. The traditional approach is to consider curves in Jacobian coordinates, a point on such a curve is represented as a triple $(X : Y : Z)$ or any triple $(\alpha^2 X : \alpha^3 Y : \alpha Z)$, with $\alpha \in \mathbb{F}_p^*$. In that case, the formulae given in [28] enable to compute the sum of two points, P and Q, sharing the same Z-coordinate, lowering the computational cost from 11M+5S for a standard point addition to 5M+2S. At the same cost one obtains the coordinates of a point $\tilde{P}$ such that :

- $P + Q$ and $\tilde{P}$ share the same Z-coordinate,
- P and $\tilde{P}$ are in the same equivalent co-set.

This operation is sometimes called ZADD, or ZADDU ([16,18]) to say ZADD with Update. Several works have used those formulae to propose efficient and secure scalar multiplication schemes: one can see [23,2,21] for right-to-left algorithms and [26] for left-to-right applications.

2.3 Euclidean addition chains

Given an ordered pair of points (P, Q) sharing the same Z-coordinate, one can compute, using the ZADD operation, either $(Q, P + Q)$ or $(P, P + Q)$ with the same Z-coordinate. Following notations from [21], the first computation will be called a *big step* (denoted by 0) and the second one will be called a *small step* (denoted by 1). For instance, starting from P and $2P$ (sharing the same Z-coordinate), one can compute $(P, 3P)$ or $(2P, 3P)$. Then, one can obtain $(P, 4P)$ or $(3P, 4P)$ from $(P, 3P)$, and $(2P, 5P)$ or $(3P, 5P)$ from $(2P, 3P)$, and so on. One can thus perform a whole scalar multiplication using Algorithm 1 and the ZADD operation (see Definition 1 below for the definition of $\chi(c)$).

Definition 1

. An Euclidean addition chain (or EAC) of length s is a finite sequence $(c_i)_{i=1\ldots s}$ of elements of $\{0, 1\}$.
. We will denote the set of EAC by $\mathcal{M}$ and the set of EAC of length s by $\mathcal{M}_s$.
. To such a sequence we associate $(v_i, u_i)_{i=0..s}$ a sequence of elements of $\mathbb{N}^2$ defined as follows:
  - $(v_0, u_0) = (1, 2)$,
  - $\forall i \in [\![1, s]\!]$, $(v_i, u_i) = (v_{i-1}, v_{i-1} + u_{i-1})$ if $c_i = 1$ (small step),
  - $\forall i \in [\![1, s]\!]$, $(v_i, u_i) = (u_{i-1}, v_{i-1} + u_{i-1})$ if $c_i = 0$ (big step).
. We will say that the sequence or the EAC $(c_i)_{i=1\ldots s}$ computes the integer $v_s + u_s$ and the pair (v_s, u_s). If $c = (c_i)_{i=1\ldots s}$ we will denote the integer $v_s + u_s$ by $\chi(c)$, and (v_s, u_s) by $\psi(c)$.

Algorithm 1 EAC_Point_Mul(c: an addition chain of length n)

Require: P and $2P$
Ensure: $Q = \chi(c)P$
1: $(U_1, U_2) \leftarrow (P, 2P)$
2: **for** $i = 1 \ldots \text{length}(c)$ **do**
3: **if** $c_i = 0$ **then**
4: $(U_1, U_2) \leftarrow ZADD(U_2, U_1)$ [it corresponds to $(U_2, U_1 + U_2)$]
5: **else**
6: $(U_1, U_2) \leftarrow ZADD(U_1, U_2)$ [it corresponds to $(U_1, U_1 + U_2)$]
7: **end if**
8: **end for**
9: **return** $Q = U_1 + U_2$

We will often denote the sequence $(c_1, ..., c_s)$ by $c_1 c_2 \ldots c_s$ for convenience. Let r and s be two integers, we will denote by cc' the element of $\mathcal{M}_{r+s}$ obtained from the concatenation of $c \in \mathcal{M}_r$ and $c' \in \mathcal{M}_s$, so that, for $n > 0$, c^n is a word of $\mathcal{M}_{nr}$.

Example 1 Let us consider the EAC $c = 00011$ of $\mathcal{M}_5$. It is related to the following sequence of ordered pairs of integers: $(1, 2) \to (2, 3) \to (3, 5) \to (5, 8) \to (5, 13) \to (5, 18)$. So it computes the integer $\chi(c) = 23$, and the couple $\psi(c) = (5, 18)$. Note that $\chi(00011) = \chi(11000) = 23$, so χ is not injective. Actually, the function χ, even restricted to some $\mathcal{M}_s$ for $s \in \mathbb{N}^*$, is never injective as we can prove that for all $c \in \mathcal{M}_s$ we have $\chi(c_1 \ldots c_s) = \chi(c_s \ldots c_1)$.

Small and big steps have an easy interpretation in terms of linear algebra.

Definition 2 Let S_0 and S_1 be the matrices corresponding to the linear maps $(v, u) \mapsto (u, u + v)$ (big step) and $(v, u) \mapsto (v, u + v)$ (small step), namely

$$S_0 = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \text{ and } S_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

For $c = (c_1, \ldots, c_s) \in \mathcal{M}_s$, we have the equalities:

$$\psi(c) = (1, 2) \prod_{i=1}^{s} S_{c_i},$$

and

$$\chi(c) = (1, 2) \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

A remarkable case is that of the EAC involving big steps only. It corresponds to the sequence of pairs of consecutive Fibonacci numbers F_n defined by $F_0 = 0$, $F_1 = 1$ and $\forall n \in \mathbb{N}$, $F_{n+2} = F_n + F_{n+1}$. Indeed,

$S_0^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}$, $\psi(0^n) = (F_{n+2}, F_{n+3})$ and $\chi(0^n) = F_{n+4}$.

There are no known regular and efficient methods to find short EAC computing a fixed integer k. This is the reason why it was suggested in [21] to randomly generate them from well fitted subsets $\mathcal{S}$, such that the restriction of χ to $\mathcal{S}$ be injective. For example, Proposition 3 in [21] states that for $(c, c') \in \mathcal{M}_n^2$, the equality $\chi(0^n c) = \chi(0^n c')$ implies $c = c'$. In other words, starting from $(v_0, u_0) = (F_{n+2}, F_{n+3})$ (instead of $(1, 2)$), one obtains 2^n different integers when computing $\chi(c)$ for all $c \in \mathcal{M}_n$.

3 Euclidean addition chains computing different points

Definition 3 Let $(a, b) \in \mathbb{N}^2$, $s \in \mathbb{N}^*$ and $c \in \mathcal{M}_s$. We define:

$$\psi_{a,b}(c) = (a, b) \prod_{i=1}^{s} S_{c_i},$$

and

$$\chi_{a,b}(c) = (a, b) \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

The case $(a, b) = (1, 2)$ corresponds to Definition 1 as for all $c \in \mathcal{M}_s$ we have $\psi_{1,2}(c) = \psi(c)$ and $\chi_{1,2}(c) = \chi(c)$. Notice that for $c \in \mathcal{M}_s$, the integer $\chi_{a,b}(c)$ is the integer computed as the sum of the two components of the vector obtained from the EAC c when starting from (a, b). In other words, $\chi_{a,b}(c).P$ is the point obtained when applying Algorithm 1 starting from (aP, bP) rather than from $(P, 2P)$. We will need the following Proposition for the two injectivity results presented in this note.

Proposition 1 *Let $s \in \mathbb{N}^*$. The map $\mu : \mathcal{M}_s \to \mathbb{N}^2$ defined by $\mu(c) = \prod_{i=1}^{s} S_{c_i} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ is injective.*

Proof. It is sufficient to prove that $\mu(c) = \mu(c')$ implies $c_1 = c'_1$. Indeed, one could thus conclude left-multiplying by $S_{c_1}^{-1}$ and using induction. To prove this claim, first notice that for all $c \in \mathcal{M}_s$ both components of the vector $\mu(c)$ are positive. Then remark that for any couple of integers (x, y) we have $S_0 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} y \\ x+y \end{pmatrix}$ and $S_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+y \\ y \end{pmatrix}$. So, if $\mu(c) = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$, we have $c_1 = 0$ if and only if $\beta > \alpha$. □

Proposition 2 *Let n, a and b be three positive integers such that a and b are co-prime and such that $a > F_{n+2}$ or $b > F_{n+2}$. Then, for all $(c, c') \in \mathcal{M}_n^2$ we have $\chi_{a,b}(c) = \chi_{a,b}(c')$ if and only if $c = c'$.*

To prove this proposition we will make use of the following lemma which follows by an easy induction.

Lemma 1 *Let n be a non-negative integer, $c \in \mathcal{M}_n$ and $\mu(c) = \begin{pmatrix} x \\ y \end{pmatrix}$. Then*

$$\left(x \leq F_{n+2} \text{ and } y \leq F_{n+1}\right) \text{ or } \left(x \leq F_{n+1} \text{ and } y \leq F_{n+2}\right).$$

As $S_0^n \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n+1} \\ F_{n+2} \end{pmatrix}$ and $S_1 S_0^{n-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} F_{n+2} \\ F_{n+1} \end{pmatrix}$ the bound is sharp.

Proof of Proposition 2. Let $(c, c') \in \mathcal{M}_n^2$ such that $\chi_{a,b}(c) = \chi_{a,b}(c')$. By definition,

$$\chi_{a,b}(c) = (a, b)\mu(c) \text{ and } \chi_{a,b}(c') = (a, b)\mu(c'). \tag{1}$$

Let us set $\begin{pmatrix} x \\ y \end{pmatrix} = \mu(c)$ and $\begin{pmatrix} x' \\ y' \end{pmatrix} = \mu(c')$, so we have $a(x - x') = b(y' - y)$. Since a and b are co-prime, Gauss lemma implies that $a \mid y' - y$ and $b \mid x - x'$. From Lemma 1 , we have that $|y' - y| \leq F_{n+2}$, thus if $a > F_{n+2}$ we deduce that $y' - y = 0$ and therefore $x - x' = 0$. In the case where $b > F_{n+2}$ we obtain in the same way that $(x, y) = (x', y')$. In both cases it enables to prove that $c = c'$, using Proposition 1. □

Example 2 With $a = F_{n+2}$ and $b = F_{n+3}$ both conditions of Proposition 2 are satisfied. We recover the case of Proposition 3 in [21], as $\psi_{1,2}(0^n) = (F_{n+2}, F_{n+3})$.

Corollary 1 *Let E be an elliptic curve and a point $P \in E$ of order N. Let n, a and b be three positive integers such that*
- a and b are co-prime,
- $a > F_{n+2}$ or $b > F_{n+2}$,
- $aF_{n+1} + bF_{n+2} < N$ and $aF_{n+2} + bF_{n+1} < N$.
Then the 2^n chains $c \in \mathcal{M}_n$ compute 2^n different points when applying Algorithm 1 starting from (aP, bP) rather than from $(P, 2P)$.

Proof. Let us consider two different elements c and c' of $\mathcal{M}_n$ as well as the two points $\chi_{a,b}(c)P$ and $\chi_{a,b}(c')P$ in E obtained from Algorithm 1. These two points are the same if and only if $\chi_{a,b}(c)$ is congruent to $\chi_{a,b}(c')$ modulo N. The precedent proposition and the first two conditions of the corollary ensure that $\chi_{a,b}(c) \neq \chi_{a,b}(c')$. Since $\chi_{a,b}(c) = (a, b)\mu(c)$ and $\chi_{a,b}(c') = (a, b)\mu(c')$, Lemma 1 and the third condition of the corollary imply that $\chi_{a,b}(c) < N$ and $\chi_{a,b}(c') < N$. Thus $\chi_{a,b}(c)P \neq \chi_{a,b}(c')P$ in E. □

Example 3 Let E be an elliptic curve, and P a point of order $N > F_{2n+4}$. We have $F_{n+2}F_{n+1} + F_{n+3}F_{n+2} = F_{n+2}^2 + F_{n+1}^2 = F_{2n+4}$, so starting from $(F_{n+2}P, F_{n+3}P)$ and applying Algorithm 1 with the 2^n chains of $\mathcal{M}_n$ enable us to compute 2^n different points of E. It corresponds to Method 1 described in [21].

Example 4 Another possibility is to start from (P, bP), where $b > F_{n+2}$ and the order of the point P is greater than $F_{n+1}+bF_{n+2}$. It requires to precompute the points P and bP with the same Z-coordinate unless bP can be efficiently computed on the fly.

4 An EAC-based scalar multiplication algorithm for curves with an efficient endomorphism

The first method proposed in [21] requires to start from a pre-computed couple of points $(F_{n+2}P, F_{n+3}P)$. Results from the previous section show that it can be extended to any pair of points (aP, bP) when a and b satisfy the hypotheses of Corollary 1. Now our concern is to adapt these methods to the variable-base scalar multiplication case. Example 4 gives food for thought: in this case we just need P and bP. If the curve is endowed by an endomorphism ϕ, we can obtain $bP = \phi(P)$ without precomputation. However the integer b given in such a way has no reason to verify the hypotheses of Corollary 1. Fortunately, we prove in this section injectivity results when starting from $(P, \phi(P))$.

From now on, we will consider the context of an elliptic curve E endowed with a non trivial endomorphism ϕ defined over $\mathbb{F}_p$. We follow the notation adopted by [11]: we fix $P \in E$ a point of prime order N such that $\#(E)/N \leq 4$, and $X^2 + rX + s$ the characteristic polynomial of ϕ. We will denote by λ the unique element of $[0, N-1]$ such that $\phi(P) = \lambda P$. The following result is established in section 2.1 of [11] and in Lemma 6 of [27].

Proposition 3 *Let $(k_1, k_2) \in \mathbb{Z}^2 \setminus \{(0,0)\}$. If $k_1 + k_2\lambda \equiv 0 \mod N$ then*

$$\max(|k_1|, |k_2|) \geq \sqrt{\frac{N}{1+|r|+s}}.$$

Combined with Proposition 1 it enables us to prove the following injectivity result.

Proposition 4 *Under the assumptions above and if $N > F_{n+2}^2(1+|r|+s)$, then the 2^n chains $c \in \mathcal{M}_n$ compute 2^n different points when applying Algorithm 1 starting from $(P, \phi(P))$ rather than from $(P, 2P)$.*

Proof. Starting from $(P, \phi(P))$ and applying Algorithm 1 with an EAC $c \in \mathcal{M}_n$, one computes $k_1P + k_2\lambda P$, where $\binom{k_1}{k_2} = \prod_{i=1}^{n} S_{c_i}\binom{1}{1}$, that is $\binom{k_1}{k_2} = \mu(c)$. Let $(c, c') \in \mathcal{M}_n^2$ such that c and c' compute the same point starting from $(P, \phi(P))$. Therefore

$$k_1 + k_2\lambda \equiv k_1' + k_2'\lambda \ (N),$$

where $\binom{k_1'}{k_2'} = \mu(c')$. We deduce

$$(k_1 - k_1') + (k_2 - k_2')\lambda \equiv 0 \ (N).$$

But we know that both components of $\mu(c)$ and $\mu(c')$ are less or equal to F_{n+2}, we thus have

$$|k_i - k_i'| < \sqrt{\frac{N}{1+|r|+s}} \qquad \text{for} \quad i \in \{1,2\}.$$

Use Proposition 3 to obtain $k_i = k_i'$ for $i \in \{1,2\}$, and Proposition 1 to conclude $c = c'$. □

Based on this result, we propose an alternative way to the classical cryptographic primitive which, starting from a point P, maps a random n-bit integer k to a random point in the group $< P >$. First randomly generate an EAC $c \in \mathcal{M}_n$. Then, starting from the couple $(P, Q) = (P, \phi(P))$, apply the ZADD addition procedure to obtain, whether the current bit of c is 0 or 1, a new ordered pair of points $(Q, P+Q)$, or $(P, P+Q)$ (see Algorithm 2). Notice that Algorithm 2 uses a slightly different version of ZADD called ZADDb (see Algorithm 3 in the Appendix B). In this version, for each iteration, the coordinates of the two starting points P and Q are used in order to store some intermediate results and are then replaced by the coordinates of the new current couple of points $(P, P+Q)$ or $(Q, P+Q)$. The algorithm takes the current bit of the addition chain as a parameter.

Algorithm 2 PointFromEAC(EAC c)

Require: $P(X, Y, 1)$ and $Q = \phi(P) = (X', Y', 1)$
Ensure: Update Q with the point computed from $(P, \phi(P))$ and the Euclidean addition chain c.

```
for i = 1 ... length(c) do
    ZADDb(c_i)
end for
ZADDb(1)
return Q
```

This way, we obtain a method which maps a random EAC chain c to a point. From a practical point of view, in order to guarantee that we compute 2^n distinct points, it is sufficient to satisfy the inequality of Proposition 4. As $F_n = \frac{\gamma^n - \overline{\gamma}^n}{\sqrt{5}}$ where $\gamma = \frac{1+\sqrt{5}}{2}$ and $\overline{\gamma} = \frac{1-\sqrt{5}}{2}$, it is sufficient to choose $N > \gamma^{2n+4}\frac{1+|r|+s}{5}$. The size of the right hand side is equivalent to $2n\log_2(\gamma)$, which is between $1.388n$ and $1.389n$. It amounts to choosing a larger base field, as the size of $E(\mathbb{F}_p)$ is close to p by Hasse-Weil bounds. For convenience, we sum up in Table 1 the size of the field necessary to guarantee the injectivity.

Security level	96	128	192
Field size	269	358	536

Table 1: Field size required for a given security level when ϕ satisfies $\phi^2 + r\phi + s = 0$ and $(r, s) = (0, 1)/(1, 1)/(-1, 2)$.

5 Implementation and performances (Weierstrass model)

In this section we analyze the computational cost of our method in comparison to the GLV method when using the classical Weierstrass model. We also propose various implementations and we consider the specific context of mobile device to illustrate the relevance of our approach. All our source codes and collected results are available on GitHub:
`https://github.com/eacElliptic`.
The detailed characteristics of the two platforms we used (an Android smartphone and an x64 based computer), as well as the various auxiliary tools are listed in Appendix A.

5.1 Theoretical cost comparisons

Let us define M_t as the cost of a modular multiplication of two t-bit integers modulo a t-bit prime number and S_t that of a squaring. Our scalar multiplication scheme using an EAC of length l consists of l calls to the ZADD procedure ($5M_t + 2S_t$) plus the initial computation of $\phi(P)$. The cost of this computation will not be taken into account neither for GLV nor for our method.

Important Remark : From the previous section, in order to guarantee a $\ell/2$-bit security level, t must be chosen close to 1.4ℓ and such that there exists a curve over a field of this size with an efficient endomorphism.

If we omit the computational cost of finding a suitable decomposition for the scalar k in the form $k_1 + k_2\lambda$, the standard GLV method requires the computation of the points $\phi(P), \phi(P) + P$ and $\phi(P) - P$, the recoding of k_1 and k_2 in joint sparse form and finally (for a $\ell/2$ bit security level), $\ell/2$ point doublings and on average $\ell/4$ point additions. Each doubling has a cost of $2M_\ell + 5S_\ell$ and each addition (provided that the stored points are in affine coordinates) costs $7M_\ell + 4S_\ell$. However, this method is not secure against Simple Side Channel analysis, that is why we also consider in our comparisons a secure version of the GLV method: GLV-SAC [12]. In that case one point doubling and one point addition is performed for each scalar bit but only the point $\phi(P)+P$ needs to be stored. We summarize the different costs in Table 2. The costs of the classical and secure versions of GLV directly depend on the size of the integer k. The cost of our method depends on the length of the EAC used in Algorithm 2. We provide numbers for some specific security levels in Table 3. In any case, the standard GLV method should be faster than the EAC approach but in the case of SPA resistant methods, we can expect our method to be competitive.

Indeed, let us consider the context where multiplication and squaring have the same cost. From Table 2, to obtain a ℓ'-bit security level, the EAC scalar multiplication algorithm needs $14\ell' + 7$ field multiplications over t-bit integers (where t is greater than $2.8\ell'$). The same computation involves $18\ell'$ field multiplications over $2\ell'$-bit integers for the protected version of GLV. Our method should be efficient as soon as

$$M_t < \frac{18\ell'}{14\ell'} M_{2\ell'} \,. \tag{2}$$

Hence , if $M_t < 1.29M_{2\ell'}$, we have an efficient alternative to the GLV protected method.

5.2 From theory to practice

From a theoretical point of view, when t is about $2.8\ell'$, the ratio $M_t/M_{2\ell'}$ should be closed to $(1.4)^2$ since the

Method	field operations
EAC	$(\ell + 1)(5M_t + 2S_t)$
W-GLV	$\ell/2 \times (5.5M_\ell + 7S_\ell)$
W-GLV-SAC	$\ell/2 \times (9M_\ell + 9S_\ell)$

Table 2: Cost analysis of different scalar multiplication methods for a $\ell/2$-bit security level.

Method	96-bit security
EAC	$965M_{269} + 386S_{269}$
W-GLV	$528M_{192} + 672S_{192}$
W-GLV-SAC	$864M_{192} + 864S_{192}$
Method	128-bit security
EAC	$1285M_{358} + 514S_{358}$
W-GLV	$704M_{256} + 896S_{256}$
W-GLV-SAC	$1152M_{256} + 1152S_{256}$
Method	192-bit security
EAC	$1925M_{536} + 770S_{536}$
W-GLV	$1056M_{384} + 1344S_{384}$
W-GLV-SAC	$1728M_{384} + 1728S_{384}$

Table 3: Theoretical cost of scalar multiplication for a given security level.

classical multiplication algorithm is quadratic. Hence, we do not expect the condition (2) to be verified generally. However, we would like to convince the reader that there exist some real-life contexts in which (2) holds. The aim of this part is to show that it is the case when using general purpose multi-precision libraries (whereas the manipulated data can be stored in few 64-bit words, as already noticed in [13]). There do exist many real-life cryptographic contexts where such libraries are used:

- the popular OpenSSL cryptographic software library relies on the BIGNUM library,
- the secure communication library GnuTLS included in the Synology Diskstation operating system relies on the GNU Multiple Precision arithmetic library,
- the Spongy Castle cryptographic library provided in the Android operating system relies on the Java BigInteger library.

From a practical point of view, the way the arbitrary precision library manages large integers as well as the real amount of memory used to store such integers must be taken into account. Generally speaking, each multiprecision library provides a multiplication procedure which is twofold: it executes the actual computation of the multiplication and also makes some extra operations needed to manage the large integers involved in this computation. So, let us denote by $\tilde{M}_t$ the cost of such a procedure for two t-bit integers. Our method will be efficient as soon as

$$\tilde{M}_t < 1.29\tilde{M}_{2\ell'} \,. \tag{3}$$

To identify some contexts where this inequality holds, we have made several benchmarks for modular multiplications over integers of size 192, 269 ($\simeq 192 \times 1.4$), 256, 358 ($\simeq 256 \times 1.4$), 384, 538 ($\simeq 384 \times 1.4$). For these benchmarks we have used the BigInteger Java library provided in the Android Software Development Kit, the BigInteger Java library provided by Oracle in Java SE for x64 platforms and the GNU Multiple Precision arithmetic library. Table 4 sums up the various execution time ratios for 2^{21} modular multiplications between t-bit integers and $1.4t$-bit integers. Favorable cases for our method appear in boldface. The ratio between 538-bit integers and 384-bit integers is about 1.9 for the multiplication procedure on our Android platform. This is due to the fact that the BigInteger library differently manages integers whose bit-length is less than 512 and integers whose bit-length is greater or equal than 512.

Remark 1 Notice that for Gnu MP these experimental results should be carefully considered. Indeed, we are near from the 1.29 bound of relation (3). If we only consider the multiplication procedure, the ratio is far enough from 1.29 for $t \in \{192, 256\}$. The problem comes from the additional cost of the modular reduction. Depending on the architecture and the processor used, it should happen that we obtained a ratio for the modular multiplication greater than 1.29 (for $t \in \{192, 256\}$) because of the modular reduction.

t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1,04095	1,05315	**1,04927**
256	358	1,00435	1,08456	**1,05934**
384	538	1,91747	1,12702	1,36432
Big Integer Android				
t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1,75618	1,64688	1,66882
256	358	1,83208	1,80546	1,81148
384	538	1,75453	1,89138	1,85883
Big Integer Java SE				
t (bits)	$\simeq 1.4t$	$\tau_\times$	$\tau_\%$	τ_*
192	269	1.15566	1.32298	**1.25854**
256	358	1.17041	1.32226	**1.26592**
384	538	1.37065	1.55186	1.48633
Gnu MP				

Table 4: Execution time ratio between operations over $1.4t$-bit integers and t-bit integers. $\tau_\times$: ratio for multiplication over integers, $\tau_\%$: ratio for modular reduction, τ_* : ratio for modular multiplication.

Remark 2 We have chosen to consider the Android platform because it is widely used in everyday connected objects like smartphones, tablet computers, GPS, car PC, and so on... More generally, there are several Java based platforms and this technology is considered by most of the developers as a de facto standard for developing embedded applications.

Since 2014, the Android operating system includes the Spongy Castle library, a cut-down version of Bouncy Castle Java library. This latter provides a lightweight cryptography API for Java. Bouncy Castle and Spongy castle both use the BigInteger Java library to operate on large integers. This library is also a part of the Java SE provided by Oracle but it differs from the version provided in Android when concerning low-level arithmetic computations. The former is written in pure Java while the latter makes calls to the BIGNUM library written in C and used in OpenSSL.

None of the libraries we used provide a specific method for squaring, so we fit the hypothesis made in the previous subsection about the identical cost of multiplication and squaring.

5.2.1 Collected results

To reach the same level of security as the classical GLV method, the conclusion of section 3 suggests that we have to work with integers which are about 1.4 times larger. As shown in Table 4, when using the above mentioned libraries a modular multiplication between $1.4t$-bit integers and t-bit integers is:

- about 1.05 times slower on Android for $t \in \{192, 256\}$,
- about 1.26 times slower on an x64 platform with GnuMP for $t \in \{192, 256\}$,
- at least 1.7 times slower on an x64 platform with Java SE for $t \in \{192, 256, 384\}$.

Let us briefly explain these results. When using multiple precision libraries, two factors determine the cost of a modular multiplication: arithmetic (whose cost grows quadratically) and memory management (whose cost grows linearly). In order to understand the total cost of such an operation, we have to consider two cases: both parts are implemented using the same language or each of them is programmed in a specific one.

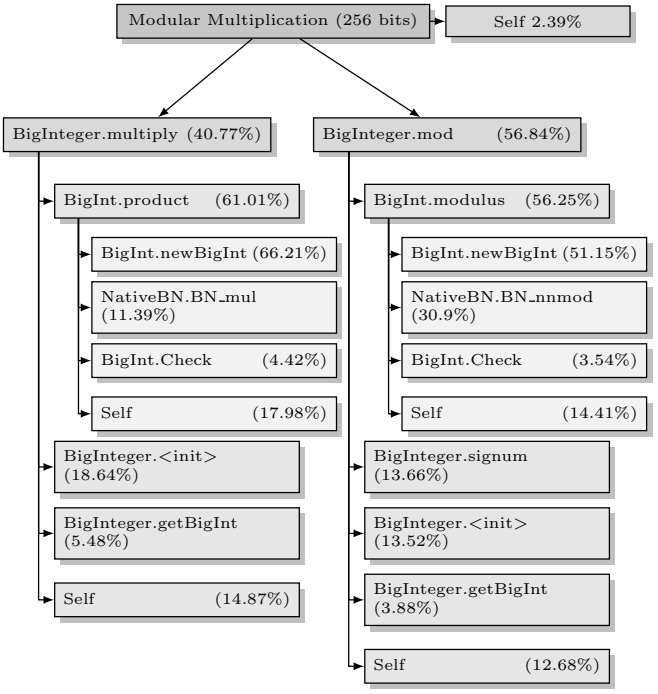
In the first case, the cost strongly depends on the relative cost of arithmetic and memory allocation in the given language. In the second case, the relative speed of the programming language used may have more impact on the overall performances than the cost of the operations themselves. For instance, in the BigInteger Java library provided by Android, arithmetic is performed using a C library (BIGNUM). Even though, the latter is supposed to be quadratic, the fact that Java is much more slower than C makes in this case the memory management be the costliest operation. With regards to the Big Integer Java library provided by Oracle for an x64 platform, arithmetic and memory are both managed in Java. For the size of the manipulated data, memory management performs better than multiplication. For Gnu MP, arithmetic performs better than memory management for the key sizes used in elliptic curve cryptography.

As a concrete illustration, we have used the profiler included in the Android software development kit in order to obtain the *anatomy* of a modular multiplication between two 256-bit integers using the BigInteger Java library (see Figure 1). The code we have developed computes 2^{17} products of two random 256-bit integers modulo a random 256-bit prime integer. To compute such a product, our Java method makes a call to the *multiply* and *mod* methods of the BigInteger library which in turn invoke other internal methods. For each of them, the ratio of the execution time of each call is given as well as the ratio of the execution time spent to execute the instructions of the method itself (denoted as "Self" in the figure).

It turns out that the actual product (*NativeBN.BN_mul*) and modular (*NativeBN.BN_nnmod*) operations represent respectively about 2.8% and 9.9% of the whole process. The rest of the time is spent between memory allocation (*BigInt.newBigInt*) and the management of the Java objects used to store large integers (see Figure 2 where we have isolated the time spent for allocation and actual execution from the time spent for other treatments). When the size of the integers is between 192 bits and 384 bits, the time spent for the actual computation of a modular multiplication takes at most 14% of the total execution time. This grows up to at most 20% for 512 to 717-bit integers. Hence for typical cryptographic key sizes, most of the time is spent between memory allocation and management of the Java objects involved in the computation process. From our experimental results, this time is almost identical for t-bit integers and $1.4t$-bit integers (see Table 5). This explains the important difference between the theoretical ratio of $(1.4)^2$ and the observed one.

Using the GNU C profiler and the Java Netbeans profiler, we performed a similar study to obtain the anatomy of a modular multiplication for an x64 platform (see Figures 3 and 4 in Appendix C). It turns out that for a Java implementation, we cannot expect to obtain

Fig. 1: Anatomy of a Java method computing a modular multiplication over 256-bit integers using Big Integer Java library provided with Android operating system (obtained from Android SDK profiler).



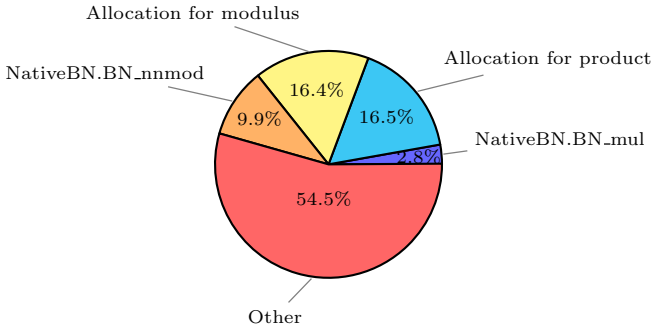


Fig. 2: Distribution of the execution time of various operations involved in the modular multiplication of 256-bit integers on an Android platform.

t (bits)	$\simeq 1.4t$	Execution time ratio for memory and object management
192	269	1,00564
256	358	0,99838
384	538	1,00485

Table 5: Execution time ratio for memory allocation and management between t-bit BigInteger objects and $1.4t$-bit BigInteger objects.

Method/Platform	Android
EAC/W-GLV (without encoding)	1.03
EAC/W-GLV (with encoding)	**0.98**
EAC/W-GLV-SAC (without enc.)	**0.75**
EAC/W-GLV-SAC (with encoding)	**0.73**

Method/Platform	Gnu MP, x64
EAC/W-GLV (without encoding)	1.38
EAC/W-GLV (with encoding)	1.32
EAC/W-GLV-SAC (without enc.)	**0.98**
EAC/W-GLV-SAC (with encoding)	**0.96**

Method/Platform	Android (x-only)
EAC/W-GLV (without encoding)	**0.91**
EAC/W-GLV (with encoding)	**0.87**
EAC/W-GLV-SAC (without enc.)	**0.69**
EAC/W-GLV-SAC (with encoding)	**0.67**

Method/Platform	Gnu MP, x64 (x-only)
EAC/W-GLV (without encoding)	1.23
EAC/W-GLV (with encoding)	1.18
EAC/W-GLV-SAC (without enc.)	**0.88**
EAC/W-GLV-SAC (with encoding)	**0.86**

Table 6: Execution time ratio between EAC and GLV schemes for a 128-bit level of security.

a competitive algorithm. Indeed, the actual computation of the multiplication is written in pure Java and its execution time is not a negligible part of the `multiply` procedure.

To sum up, the results of Table 4 show that there exist some practical contexts where the execution time of a modular multiplication between integers used in our EAC scheme satisfies the condition (3).

5.2.2 Practical implementations

To illustrate this purpose, we have implemented on an Android device and on an x64 platform (using Gnu MP) the classical GLV method, the protected version of GLV and our method.

First, notice that the EAC scalar multiplication algorithm needs only few lines of code (see Algorithm 2 and Algorithm 3 in Appendix B). Jacobian coordinates are used to represent all points and only the ZADD addition formula has to be implemented. For GLV and GLV-SAC two initial steps have to be implemented: a precomputation step which splits k into k_1 and k_2 (Algorithms 4 and 5, Appendix B) and an encoding process to put k_1 and k_2 in JSF form [33] or SAC form (Algorithm 8, Appendix B). Moreover, we need to implement addition formulas for mixed affine-Jacobian coordinates, as well as doubling formulas.

Although the classical GLV version is included in Spongy castle, we decided to make our own implementation. This way, we aim to make a fair comparison between our method, the protected version of GLV and the classical one. Indeed, Spongy Castle library uses generic Java objects and includes several tests so that the end-user can make his choice between different kinds of curves and scalar multiplication algorithms. This will certainly slow down the execution time as compared to a specific version dealing only with the GLV method.

We have chosen to implement GLV (Algorithm 6, Appendix B) and GLV-SAC (Algorithm 7, Appendix B) for the curve $E(\mathbb{F}_p) : y^2 = x^3 + 5$ with $p = 2^{256} - 1539$, and our EAC scheme for the curve $E(\mathbb{F}_p) : y^2 = x^3 + 17$ with $p = 2^{358} - 36855$. These two sets of parameters guarantee a 128-bit security level (see Table 1). The corresponding endomorphism ϕ satisfies $\phi^2 + \phi + 1 = 0$. It corresponds to the map $(x, y) \mapsto (\beta x, y)$ where β is an element of order 3.

In order to compare these methods, we have recorded the average execution time of 2^{17} scalar multiplications. Table 6 gives the corresponding ratios. The favorable cases for our method appear in boldface. Notice that for the GLV methods, the computation of kP (for various points P) requires to decompose k into two smallest integers k_1 and k_2. This encoding process is often over-

looked in the computational cost of kP. However, in some contexts, k and P can both change and thus this encoding process can slow down the whole GLV execution. The encoding process for GLV is done by Algorithms 4 and 5 (Appendix B) as described in [31]. The running time of this decomposition is better than those of the original encoding process proposed by Gallant, Lambert and Vanstone. For GLV, the integers k_1 and k_2 must then be put into Joint Sparse Form [33]. For GLV-SAC, k_1 and k_2 must be post-processed by Algorithm 8 (Appendix B, adapted from [12] to fit in our context).

Instead of randomly choosing the integer k, another way is to randomly generate k_1 and k_2 as suggested and justified in [9]. Notice that in this case the *recomposition* of k from k_1 and k_2 is no more injective. In Table 6, we made comparisons with two versions of GLV: one with the encoding of k, and the other one without encoding.

It turns out that on an Android platform, the EAC scalar multiplication algorithm performs well as compared to the classical GLV algorithm (only 3% slower) and can even be faster (2%) in the context where k and P both vary. The most significant gain is obtained when taking into account the resistance to side channel attacks since our scheme is at least 25% faster than the protected version of GLV. For the x64 platform, our execution time is 2% faster than the secure version of GLV (this result should be carefully considered as explained in Remark 1). We obtain a more significant gain if we target x-coordinate-only system (see Section 9 and [9]).

Remark 3 We insist on the fact that once we have chosen a general multi-precision library, our constraint is to use the primitives offered by this latter. Hence, we do not consider any optimized modular reduction.

Let us focus on the classical multiplication operation over integers. For 96-bit and 128-bit security levels, Table 4 (column $\tau_\times$) shows that the EAC method should always perform better than GLV-SAC on an Android platform (or an x64 platform using GNU MP). Hence, if we consider both for GLV-SAC and our algorithm some specific primes p (like in our examples), using fast modulo reduction techniques will certainly optimize numbers given in the column $\tau_\%$. Thereby, compared to GLV-SAC, the performances of the EAC algorithm should be better.

Remark 4 The fact that our method is only 3% slower than the classical GLV method (without encoding) may seem curious. The results of Table 3 show that there are 1.12 as many multiplications in the EAC based algorithm than in the GLV method. Since, from Table 4, a modular multiplication between two 358-bit integers is about 1.06 times slower than a multiplication between two 256-bit integers, this should lead us to an execution time ratio of $1.12 \times 1.06 \simeq 1.187$ rather than 1.03. Now, let us isolate in our experimental results the time spent for modular multiplications during the scalar multiplication algorithm. We observe a ratio of about 1.19. This result confirms the theoretical analysis of subsection 5.1. This gives us the opportunity to emphasize the need for caution when only considering the cost of the modular multiplication in order to state the complexity of a scalar multiplication algorithm. Indeed, if we take into account the whole computation process, including modular additions, modular multiplications by constants which appear in point addition formulas and the management of the different auxiliary Java objects used in all these computations, the theoretical ratio of 1.19 boils down to a practical ratio of 1.03. Addition formulas used for GLV [4] involve some field multiplications by a constant. Even if those multiplications can be done using the `BigInteger.shiftleft()` method of the BigInteger library, their cost is non-negligible.

6 Implementation and performances (extended twisted Edwards curves model)

Ted127-glv4 [12] and FourQ [10] are the two actual most efficient (and secure) scalar multiplication algorithms. They rely on the four-dimensional GLV algorithm and take advantage of two efficient endomorphisms and the extended twisted Edwards coordinates [22], which offer the fastest known curves addition formulas over large prime characteristic fields. Nevertheless, some precomputed points need to be stored for table lookups. As an example, 512 bytes of memory are needed to store 8 points in Ted127-glv4.
A raw comparison of our approach with these methods is complicated. On the one hand, without a doubt, methods using two endomorphisms are faster, on the other hand our implementation context is quite different (arithmetic over $\mathbb{F}_p$, low memory usage and curves with one endomorphism). However, it seems fair to take advantage of the twisted Edwards coordinates system, to compare our approach to the 2-dimensional version of GLV-SAC, which matches our context.

6.1 Theoretical cost comparisons with GLV-SAC

In extended twisted Edwards coordinates, a point is represented by four coordinates (X, Y, T, Z) with $T = XY/Z$. Using Algorithm 7, each point doubling is followed by a point addition. The best performances for

this sequence of operations is obtained by mixing standard twisted Edwards coordinates with extended twisted Edwards coordinates (as explained in [22]). This leads to $10M_\ell + 4S_\ell$ for a $\ell/2$-bit security level (see Table 7 and 8). It appears from this first theoretical approach, that we cannot outperform GLV-SAC when using extended twisted Edwards coordinates.

Once again, experimental results (see Table 9) reveal that we have to carefully take into account every computations in order to evaluate the real cost of a scalar multiplication algorithm. Results from Table 9 have been obtained by implementing GLV-SAC on the twisted Edwards curve $E(\mathbb{F}_p) : -x^2 + y^2 = 1 + x^2y^2$ with $p = 2^{256} - 43443$. The order of the curve is $8.h$ for some prime h. The corresponding endomorphism ϕ satisfies $\phi^2+1 = 0$. It corresponds to the map $(x, y) \mapsto (\alpha x, 1/y)$ where α is an element of order 4 [25]. Parameters for our EAC scheme are those defined in section 5.2.2. In order to compare these two methods, we have recorded the average execution time of 2^{17} scalar multiplications.

Let us explain the results obtained on the Android platform. First, from Table 4, we see that the computational cost of a 358-bit modular multiplication barely exceeds that of a 256-bit one. The ZADDb procedure involves precisely 7 modular multiplications (assuming the same cost for squaring and multiplying operations) and 7 modular additions. The mixed coordinate addition for the twisted Edward curves [22] involves precisely 14 multiplications, 14 additions, 1 multiplication by 2 and on average the computation of a modular opposite:

1. a doubling requires 8 multiplications, 6 additions and a multiplication by the curve constant a, which in our case is equal to -1,
2. an addition requires 6 multiplications, 8 additions and two multiplications by 2 [22],
3. half the time the opposite of one of the two affine points P or $P + \phi(P)$ has to be computed.

The multiplication by 2 can be implemented using either the `BigInteger.shiftleft()` method or using the `BigInteger.add()` method. Experimental results show that these two methods have almost the same execution time. Indeed, these two methods allocate a new BigInteger object, and as it has already been discussed in section 5.2.1, the most important part in the execution time of an operation comes from the memory allocation.

Similarly, the modular opposite can be implemented using either the `BigInteger.neg()` method or the `BigInteger.subtract()` method. For the reasons mentioned above, these two operations are almost equivalent.

Table 10 summarizes the cost of those two methods for a ℓ-bit security level (D_ℓ denotes the multiplication by 2 of an ℓ-bit integer and N_ℓ the modular opposite of an ℓ-bit integer). As soon as computing $\ell(3D_\ell/2 + N_\ell)$ is slower than computing $7M_t + 7A_t$ our approach becomes more efficient. In other words, we have to compare the allocation time of 14 BigInteger objects used to compute $7M_t + 7A_t$, with the allocation time of $5\ell/2$ BigInteger objects used to compute $\ell(3D_\ell/2 + N_\ell)$.

Let T_{EAC} be the execution time of our EAC method. Considering that the execution time of a 256-bit and a

Method	field operations
EAC	$(\ell + 1)(5M_t + 2S_t)$
TED-GLV-SAC	$\ell/2 \times (10M_\ell + 4S_\ell)$

Table 7: Cost analysis of EAC and TED-GLV-SAC for a $\ell/2$-bit security level ($t \simeq 1.4\ell$).

Method	96-bit security
EAC	$965M_{269} + 386S_{269}$
TED-GLV-SAC	$960M_{192} + 384S_{192}$
Method	**128-bit security**
EAC	$1285M_{358} + 514S_{358}$
TED-GLV-SAC	$1280M_{256} + 512S_{256}$
Method	**192-bit security**
EAC	$1925M_{536} + 770S_{536}$
TED-GLV-SAC	$1920M_{384} + 768S_{384}$

Table 8: Theoretical cost of scalar multiplication for a given security level.

Method/Platform	Android
EAC/TED-GLV-SAC (without enc.)	**0.95**
EAC/TED-GLV-SAC (with enc.)	**0.93**

Method/Platform	Gnu MP, x64
EAC/TED-GLV-SAC (without enc.)	1.41
EAC/TED-GLV-SAC (with enc.)	1.39

Method/Platform	Android (x-only)
EAC/TED-GLV-SAC (without enc.)	**0.84**
EAC/TED-GLV-SAC (with enc.)	**0.82**

Method/Platform	Gnu MP, x64 (x-only)
EAC/TED-GLV-SAC (without enc.)	1.21
EAC/TED-GLV-SAC (with enc.)	1.19

Table 9: Execution time ratio between EAC and TED-GLV-SAC schemes for a 128-bit level of security.

Method	field operations
EAC	$(\ell+1)(7M_t+7A_t)$
TED-GLV-SAC	$\ell/2\times(14M_\ell+14A_\ell+3D_\ell+2N_\ell)$

Table 10: Cost analysis of EAC and TED-GLV-SAC for a $\ell/2$-bit security level ($t\simeq 1.4\ell$).

358-bit modular multiplication (or addition) is roughthly the same, the computational cost of TED-GLV-SAC should be $T_{\text{EAC}}+\delta(\frac{5\times 256}{2}-14)$ where δ is the execution time of memory allocation. Those allocations are done through calls to the `BigInt.newBigInt()` method. Experimentation has shown that its execution time is about 50 microseconds which leads to a ratio of

$$\frac{T_{\text{EAC}}}{T_{\text{EAC}}+31300}\,.$$

Performing 2^{17} scalar multiplications, we have obtained an average execution time of 693943 microseconds for T_{EAC}, which gives a ratio of about 0.96, in accordance with the results of row 1 in Table 9. Notice that the same analysis cannot be done for the x64 platform: the difference between the execution times of modular multiplications is too large (see Table 4).

It turns out that on an Android platform, the EAC scalar multiplication algorithm performs well as compared to the 2-dimensional GLV-SAC algorithm even in the context of extended twisted Edwards coordinates. Using the x-coordinate-only system makes the comparison even more favorable.

On an x64 platform, on the one hand the x-coordinate-only version of EAC is 1.2 times slower than TED-GLV-SAC, but on the other hand TED-GLV-SAC needs 1.4 times more memory (see section 8). Our method achieves a reasonable trade off between execution time and data storage.

7 Security

Our algorithm is based on the constant time `ZADDb` procedure making it intrinsically immune to simple side channel attacks like power consumption measurement. Moreover, the procedure is composed of a fixed sequence of instructions (mixing multiplications and subtractions over the prime field) which does not depend on the scalar k. This is enough to thwart many instruction cache timing attacks [1,3,35]. Yet, some additional precaution must be taken in order to resist to the attack proposed in [7]. Indeed, considering the table built from the precomputed points, each bit of the key leads to a table lookup to select one of these points. The corresponding index is a scalar digit which can be revealed thanks to a cache timing trace. Since the selected index is linked to the key, the attacker can derive the value of the key. GLV-SAC is vulnerable to this attack. As a possible countermeasure, the author of [24] proposes to transverse the entire table for each table lookup and to use some functions to compute the correct index. That means that the entire table is loaded into the cache. In the `ZADDb` procedure, we have a table of two points and, at each iteration, these two points are used. Consequently, their addresses should always be loaded into the cache. However, as it can be seen in the `ZADDb` procedure, some operations depend on the address of the point that is used. As an example, in line 4, the CPU will either load X_0 or X_1 in order to perform the multiplication $X_b.C$. An attacker could actually fill the cache with values to observe timing differences depending on where and how many of the operands X_0 and X_1 are used. In our context, the countermeasure described in [24] can be simplified because we only need to transverse two points. We describe in Algorithm 10 a safe permutation that returns a pair of points (P,Q) or (Q,P) depending on bit b. We then adapt the `ZADDb` procedure and our EAC scheme to take into account this safe permutation (see Algorithms 9 and 11). We will see in the next section why this countermeasure cannot be applied as such for TED-GLV-SAC.

8 Memory usage

Table 11 sums up (for a 128-bit level of security) the number of registers needed to store the coordinates of the various points involved in the computation of kP as well as the number of auxiliary registers used for the addition and doubling formulas (see Algorithm 3 in Appendix B and [4]).

Notice that for the EAC algorithm, the two initial input points are represented using Jacobian coordinates and share a common coordinate Z. Moreover, since the input points are used to store intermediate results as well as the coordinates of the computed point (Algorithm 2, Appendix B), the `ZADDb` procedure can be implemented with only two additional registers.
For GLV (resp. GLV-SAC), in the Weierstrass model, the four (resp. two) input points are in affine coordinates, while intermediate points and the computed point are in Jacobian coordinates. For TED-GLV-SAC, the two input points are in extended affine coordinates $(x,y,xy,1)$, and the intermediate point Q is either represented in extended or standard twisted Edwards coordinates. In both cases, intermediate results cannot be stored into the coordinates of the input points because

Algorithms 6 and 7 in Appendix B use these points at each iteration of their main loop. Four (resp. two) additional registers are needed for the whole computation of GLV and GLV-SAC (resp. TED-GLV-SAC).

Method	EAC	EAC x-only
Field size	358	358
Registers for points	5	4
Auxiliary registers	2	2
Mem. size (bits)	2506	2148
Mem. size (bytes)	314	269
Mem. size (32-bit)	79	68
Mem. size (64-bit)	40	34

Method	GLV	GLV-SAC	TED-GLV-SAC
Field size	256	256	256
Registers for points	11	7	10
Auxiliary registers	4	4	2
Mem. size (bits)	3840	2816	3072
Mem. size (bytes)	480	352	384
Mem. size (32-bit)	120	88	96
Mem. size (64-bit)	60	44	48

Table 11: Memory usage in bits/8-bit words/32-bit words/64-bit words for a 128-bit level of security.

Remark 5 Our countermeasure (see previous section) cannot be applied as such for GLV-SAC and TED-GLV-SAC even though two precomputed points are used. The points P or $P+\phi(P)$ can be safely selected. However, half the time the points $-P$ or $-(P+\phi(P))$ have to be computed, leading to possible leakage. To prevent this, the four points P, $P+\phi(P)$, $-P$ and $-(P+\phi(P))$ must be stored and the table lookup must be implemented as suggested in [24].

Remark 6 A faster version of GLV-SAC for 2-dimensional GLV is described in section 3.2 of [12]. The drawback is that eight points must be stored instead of two. The same remark can be made for the two fastest scalar multiplication algorithms yet, namely Ted127-glv4 and FourQ. Both algorithms each requires 512 bytes to store the precomputed points when ours only requires 224 bytes.

Compared to the methods using at least one endomorphism, the EAC scalar multiplication algorithm is less memory consuming. Hence it seems well suited for memory constrained devices using for example a Java virtual machine. Now, on memory constrained devices where the cryptographic operations are performed by a specific library located in ROM or EEPROM/Flash that directly accesses the RAM , our algorithm will be less efficient because in this context, equation (3) will certainly not be satisfied.

9 Other Related works

In [9] the authors use a curve and its quadratic twist (over $\mathbb{F}_{p^2}$) to obtain a fast x-coordinate-only scalar multiplication algorithm using 256 bytes of memory for precomputations. Our method can also be derived in an x-coordinate-only scheme as described in [28] (see Table 6) and only uses 179 bytes of memory. Moreover, contrary to the method proposed in [9], our algorithm guarantees that for two distinct inputs, two distinct points are output.

An implementation of GLV (standard and secure version) for OpenSSL is discussed in [6]. For a 128-bit security level, the proposed secure version is about 10% slower than the standard GLV version. We cannot make a practical comparison with this proposal because we did not investigate the OpenSSL library. Now, OpenSSL and Android both rely on the BIGNUM C-library for arithmetic over large integers. We can just notice that on Android, the performances of both classical GLV and our method are very close to one another (and 10% faster for an x-only-coordinate system). On the other hand, on an x64 platform with Gnu MP, we have an additional cost of about 40% (and almost no additional cost for an x-coordinate-only system).

Notice that another way to naturally obtain an x-only-coordinate system for our method is to use the work described in [17]. In this paper the authors describe a modified version of the ZADD operation such that all the computations can be done using only affine coordinates. Since our scalar multiplication algorithm is based on the ZADD primitive, we can benefit from this procedure.

10 Conclusion

In this work we have given a mathematical context which enables to use Euclidean addition chains on a curve with one endomorphism. Therefore we have proposed an efficient and secure algorithm for scalar multiplication which can be very easily implemented. The proposed method does not compete with the fastest ones on curves endowed with two endomorphisms. However, up to our knowledge, ours appears to be the less memory consuming among all endomorphism based methods. For developers using the standard cryptographic library included in the Android operating system, we provide a competitive algorithm as compared to the well known GLV method using one endomorphism, even when using extended twisted Edward coordinates.

Acknowledgements: we would like to thank the referees for their careful reading and their helpful comments.

References

1. Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *Proceedings of CHES 2010*, volume 6225 of *LNCS*, pages 110–124. Springer Berlin Heidelberg, 2010.
2. B. Baldwin, R. R. Goundar, M. Hamilton, and W. P. Marnane. Co-z ECC scalar multiplications for hardware, software and hardware-software co-design on embedded systems. *J. Cryptographic Engineering*, 2(4):221–240, 2012.
3. Naomi Benger, Joop Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In *Proceedings of CHES 2014*, volume 8731, pages 75–92, New York, NY, USA, 2014. Springer-Verlag New York, Inc.
4. D. J. Bernstein and T. Lange. Explicit-Formulas Database. `https://www.hyperelliptic.org/EFD/`.
5. A. Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 1939.
6. Billy Bob Brumley. Faster software for fast endomorphisms. In *Constructive Side-Channel Analysis and Secure Design - 6th International Workshop, COSADE 2015*, pages 127–140, 2015.
7. Billy Bob Brumley and Risto M. Hakala. Cache-timing template attacks. In *Advances in Cryptology – ASIACRYPT 2009: 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan*, pages 667–684. Springer Berlin Heidelberg, 2009.
8. H. Cohen, G. Frey, R. M. Avanzi, C. Doche, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Boca Raton, FL: Chapman & Hall/CRC, 2006.
9. C. Costello, H. Hisil, and B. Smith. Faster compact Diffie–Hellman: endomorphisms on the x-line. In *Advances in Cryptology–EUROCRYPT 2014*, pages 183–200. Springer, 2014.
10. Craig Costello and Patrick Longa. Four$\mathbb{Q}$: four-dimensional decompositions on a $\mathbb{Q}$-curve over the Mersenne prime. In *Advances in cryptology – ASIACRYPT 2015. 21st international conference on the theory and application of cryptology and information security, Auckland, New Zealand, November 29 – December 3, 2015. Proceedings. Part I*, pages 214–235. Berlin: Springer, 2015.
11. F. Sica and M. Ciet and J-J. Quisquater. Analysis of the Gallant-Lambert-Vanstone method based on efficient endomorphisms: elliptic and hyperelliptic curves. In *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 21–36. Springer, 2003.
12. A. Faz-Hernández, P. Longa, and A. H. Sánchez. Efficient and secure algorithms for GLV-based scalar multiplication and their implementation on GLV-GLS curves (extended version). *J. Cryptographic Engineering*, 5(1):31–52, 2015.
13. S. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. *Journal of Cryptology*, 24(3):446–469, 2011.
14. S. D. Galbraith, X. Lin, and M. Scott. Endomorphisms for faster elliptic curve cryptography on a large class of curves. In *Advances in Cryptology - EUROCRYPT 2009*, volume 5479 of *LNCS*, pages 518–535. Springer Berlin Heidelberg, 2009.
15. R. P. Gallant, R. J. Lambert, and S. A. Vanstone. Faster point multiplication on elliptic curves with efficient endomorphisms. In *Advances in Cryptology — CRYPTO*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.
16. R. R. Goundar, M. Joye, and A. Miyaji. Co-Z addition formulæ and binary ladders on elliptic curves - (extended abstract). In *Cryptographic Hardware and Embedded Systems, CHES 2010*, pages 65–79, 2010.
17. Raveen R. Goundar and Marc Joye. Inversion-free arithmetic on elliptic curves through isomorphisms. *Journal of Cryptographic Engineering*, pages 1–13, 2016.
18. Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on Weierstraß elliptic curves from co-z arithmetic. *Journal of Cryptographic Engineering*, 1(2):161–176, 2011.
19. A. Guillevic and S. Ionica. Four-dimensional GLV via the weil restriction. In *Advances in Cryptology - ASIACRYPT 2013*, pages 79–96, 2013.
20. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.
21. F. Herbaut, P-Y. Liardet, N. Méloni, Y. Téglia, and P. Véron. Random euclidean addition chain generation and its application to point multiplication. In *Progress in Cryptology - INDOCRYPT 2010*, volume 6498 of *LNCS*, pages 238–261. Springer Berlin / Heidelberg, 2010.
22. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, and Ed Dawson. Twisted Edwards curves revisited. In *Advances in cryptology – ASIACRYPT 2008. 14th international conference on the theory and application of cryptology and information security, Melbourne, Australia, December 7–11, 2008. Proceedings*, pages 326–343. Berlin: Springer, 2008.
23. M. Hutter, M. Joye, and Y. Sierra. Memory-constrained implementations of elliptic curve cryptography in co-Z coordinate representation. In *Progress in Cryptology - AFRICACRYPT 2011*, pages 170–187, 2011.
24. Emilia Käsper. Fast elliptic curve cryptography in openssl. In *Proceedings of the 2011 International Conference on Financial Cryptography and Data Security*, FC'11, pages 27–39. Springer-Verlag, 2012.
25. Zhe Liu, Husen Wang, Johann Großschädl, Zhi Hu, and Ingrid Verbauwhede. Vlsi implementation of double-base scalar multiplication on a twisted edwards curve with an efficiently computable endomorphism. *IACR Cryptology ePrint Archive*, 2015:421, 2015.
26. Patrick Longa and Ali Miri. *New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields*, pages 229–247. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.
27. Patrick Longa and Francesco Sica. Four-dimensional Gallant–Lambert–Vanstone scalar multiplication. *Journal of Cryptology*, 27(2):248–283, 2014.
28. N. Meloni. New point addition formulae for ECC applications. In *Arithmetic of Finite Fields*, volume 4547 of *LNCS*, pages 189–201. Springer Berlin / Heidelberg, 2007.
29. B. Möller. Improved techniques for fast exponentiation. In Springer Berlin / Heidelberg, editor, *Information Security and Cryptology — ICISC 2002*, volume 2587 of *LNCS*, pages 298–312, 2003.
30. P. L. Montgomery. Evaluating Recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, 1983. Available at ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz.

31. Y-H. Park, S. Jeong, C. Kim, and J. Lim. An alternate decomposition of an integer for faster point multiplication on certain elliptic curves. In *Public Key Cryptography*, volume 2274 of *LNCS*, pages 323–334. Springer Berlin Heidelberg, 2002.
32. B. Smith. Families of fast elliptic curves from ıı-curves. In *Advances in Cryptology - ASIACRYPT 2013*, pages 61–78, 2013.
33. J. A. Solinas. Low-weight binary representations for pairs of integers. Technical report, University of Waterloo. Department of Combinatorics and Optimization, 2001.
34. E. G. Thurber. On addition chains $l(mn) \leq l(n) - b$ and lower bounds for $c(r)$. *Duke Mathematical Journal*, 40:907–913, 1973.
35. Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association.

Appendix A Benchmark Platforms

All our source codes and collected results are available on GitHub:
`https://github.com/eacElliptic`.

The characteristics of the platforms we used for our benchmarks are the following:

- Android platform : Wiko Cik Peax 2 phone, with MediaTek MT6589 CPU (4 -core ARM Cortex-A7, 1.21 GHz), Android Version : 4.1.2, API Level 16.
- Java platform : Intel Core I5-4210U 4-core 1.7Ghz, Broadwell technology, JDK 1.7.0_79, Ubuntu 14.04 LTS.
- C platform : Intel Core I5-4210U 4-core 1.7Ghz, Broadwell technology, gcc 5.2.1, gmp 6.1.0, Ubuntu 14.04 LTS.

From various benchmarks, when the CPU models are exactly the same, it seems that on a given task Broadwell technology is better than Haswell technology by 5 percent to 10 percent or so. The Intel Turbo Boost technology has been disabled on the x64 platform so that the frequency of the processor be constant.

To collect the various execution results we have used the following tools:

- Android platform: the `startMethodTracing` and the `stopMethodTracing` of the `Debug` class to generate trace logs, and the `System.currentTimeMillis()` method to measure execution time,
- Java platform: the profiler provided with Netbeans IDE (v. 8.1) and the `System.currentTimeMillis()` method,
- C platform: the `clock.gettime()` system call with `CLOCK_PROCESS_CPUTIME_ID` option, and `taskset` to bind our running process to only one processor.

Appendix B Algorithms

Algorithm 3 ZADDb(bit b)

Require: $P(X_0, Y_0, Z)$ and $Q(X_1, Y_1, Z)$
Ensure: Update X_0, Y_0, X_1, Y_1 and Z such that (X_0, Y_0, Z) and (X_1, Y_1, Z) be the representatives of P and $P + Q$ (or Q and $P + Q$ whether b is 1 or 0).

$C \leftarrow X_{1-b} - X_b$
$Z \leftarrow Z.C$
$C \leftarrow C^2$
$W \leftarrow X_b.C$
$X_0 \leftarrow X_{1-b}.C$
$C \leftarrow Y_{1-b} - Y_b$
$X_1 \leftarrow C^2 - W - X_0$
$W \leftarrow X_0 - W$
$Y_0 \leftarrow Y_{1-b}.W$
$W \leftarrow X_0 - X_1$
$Y_1 \leftarrow C.W - Y_1$

Algorithm 4 Precomputation for GLV

Require: λ root of $\phi^2 + \phi + 1 = 0 \pmod{\#E(\mathbb{F}_p)}$
Ensure: (a, b) is a short vector such that $a + b\lambda \equiv 0 \pmod{\#E(\mathbb{F}_p)}$, $N_\alpha = N_{\mathbb{Z}[\phi]/\mathbb{Z}}(a + b\phi)$.

$u \leftarrow \#E(\mathbb{F}_p)$, $v \leftarrow \lambda$, $q \leftarrow 0$, $y \leftarrow 0$
$x_1 \leftarrow 1$, $y_1 \leftarrow 0$, $x_2 \leftarrow 0$, $y_1 \leftarrow 1$
while $u > \sqrt{\#E(\mathbb{F}_p)}$ **do**
    $q \leftarrow \lfloor v/u \rfloor$, $r \leftarrow v - qu$
    $x \leftarrow x_2 - qx_1$, $y \leftarrow y_2 - qy_1$
    $v \leftarrow u$, $u \leftarrow r$
    $x_2 \leftarrow x_1$, $x_1 \leftarrow x$
    $y_2 \leftarrow y_1$, $y_1 \leftarrow y$
end while
$b \leftarrow -y$
$a \leftarrow u + y$
$N_\alpha \leftarrow u^2 + b^2 - ub$

Algorithm 5 Decompose(k)

Require: a, b, N_α
Ensure: k_1 and k_2 satisfy $kP = k_1P + k_2\phi(P)$
$x_1 \leftarrow k(a+b)$, $x_2 \leftarrow -kb$
$y_1 \leftarrow \lfloor x_1/N_\alpha \rfloor$, $y_2 \leftarrow \lfloor x_2/N_\alpha \rfloor$
$k_1 \leftarrow k - (ay_1 - by_2)$, $k_2 \leftarrow -(ay_2 + b_y 1 + by_2)$
$k_1 \leftarrow k_1 + k_2$
return (k_1, k_2)

Algorithm 6 PointFromGLV(k, PP)

Require: PP is $(P, -P + \phi(P), \phi(P), P + \phi(P))$
Ensure: $Q = kP$
$(k_1, k_2) \leftarrow$ Decompose(k)
$((x_j, \ldots, x_0), (y_j, \ldots, y_0)) \leftarrow$ SJSF(k_1, k_2)
$u \leftarrow x_j + 3.y_j$
$Q \leftarrow (X_{PP[|u|-1]}, \text{sign}(u).Y_{PP[|u|-1]})$
$j \leftarrow j - 1$
while $(j \geqslant 0)$ **do**
    $Q \leftarrow 2Q$
    $u \leftarrow x_j + 3.y_j$
    if $u \neq 0$ **then**
        $Q \leftarrow Q + (X_{PP[|u|-1]}, \text{sign}(u).Y_{PP[|u|-1]})$
    end if
end while
return Q

Algorithm 7 PointFromSGLV(k, PP)

Require: PP is $(P, P + \phi(P))$
Ensure: $Q = kP$
$(k_1, k_2) \leftarrow$ Decompose(k)
if k_1 is even **then**
    $k_1 \leftarrow k_1 - 1$
end if
$((x_j, \ldots, x_0), (y_j, \ldots, y_0)) \leftarrow$ GLV-SAC(k_1, k_2)
$Q \leftarrow (X_{PP[|y_j|]}, \text{sign}(x_j).Y_{PP[|y_j|]})$
$j \leftarrow j - 1$
while $(j \geqslant 0)$ **do**
    $Q \leftarrow 2Q$
    $Q \leftarrow Q + (X_{PP[|y_j|]}, \text{sign}(x_j).Y_{PP[|y_j|]})$
end while
if k_1 is even **then**
    $Q \leftarrow Q + (X_{PP[0]}, Y_{PP[0]})$
end if
return Q

Algorithm 8 GLV-SAC(k_1, k_2)

Require: $k_1 = (k_{\ell-1}^{(1)}, \ldots, k_0^{(1)})$ and $k_2 = (k_{\ell-1}^{(2)}, \ldots, k_0^{(2)})$ are ℓ-bit positive integers. k_1 is odd and $\ell = \#E(\mathbb{F}_p)$.
Ensure: Output $(b_{\ell-1}^{(1)}, \ldots, b_0^{(1)})$ and $(b_{\ell-1}^{(2)}, \ldots, b_0^{(2)})$ such that $\forall i$, $b_i^{(1)} \in \{-1, 1\}$ and $b_i^{(2)} \in \{0, b_i^{(1)}\}$.
$b_{\ell-1}^{(1)} \leftarrow 1$
for $i = 0, \ldots, \ell - 2$ **do**
    $b_i^{(1)} \leftarrow 2k_{i+1}^{(1)} - 1$
    $b_i^{(2)} \leftarrow b_i^{(1)}.k_0^{(2)}$
    $k_2 \leftarrow \lfloor k_2/2 \rfloor - \lfloor b_i^{(2)}/2 \rfloor$
end for
$b_{\ell-1}^{(2)} \leftarrow k_0^{(2)}$
return $(b_{\ell-1}^{(1)}, \ldots, b_0^{(1)})$, $(b_{\ell-1}^{(2)}, \ldots, b_0^{(2)})$

Algorithm 9 ZADDU(P, Q)

Require: $P(X_0, Y_0, Z)$ and $Q(X_1, Y_1, Z)$
Ensure: Update X_0, Y_0, X_1, Y_1 and Z such that (X_0, Y_0, Z) and (X_1, Y_1, Z) be the representatives of P and $P + Q$.
$A \leftarrow X_1 - X_0$
$Z \leftarrow Z.A$
$A \leftarrow A^2$
$X_0 \leftarrow X_0.A$
$A \leftarrow X_1.A$
$Y_1 \leftarrow Y_1 - Y_0$
$B \leftarrow Y_1^2$
$X_1 \leftarrow B - X_0 - A$
$A \leftarrow A - X_0$
$Y_0 \leftarrow Y_0.A$
$B \leftarrow X_0 - X_1$
$Y_1 \leftarrow Y_1.B - Y_0$

Algorithm 10 SafePerm(P, Q, bit bit)

Require: $P(X_0, Y_0, Z)$, $Q(X_1, Y_1, Z)$ and bit
Ensure: Permute safely P and Q if bit is equal to 0.
mask $\leftarrow (bit - 1)$
$X_0 \leftarrow X_0 \oplus X_1$
$X_1 \leftarrow (\text{mask} \,\&\, X_0) \oplus X_1$
$X_0 \leftarrow X_0 \oplus X_1$
$Y_0 \leftarrow Y_0 \oplus Y_1$
$Y_1 \leftarrow (\text{mask} \,\&\, Y_0) \oplus Y_1$
$Y_0 \leftarrow Y_0 \oplus Y_1$

Algorithm 11 PointFromEAC(EAC c)

Require: $P(X, Y, 1)$ and $Q = \phi(P) = (X', Y', 1)$
Ensure: Update Q with the point computed from $(P, \phi(P))$ and the Euclidean addition chain c.
1: **for** $i = 1 \dots \text{length}(c)$ **do**
2: SafePerm(P, Q, c_i)
3: ZADDU(P, Q)
4: **end for**
5: ZADDU(P, Q)
6: **return** Q

Appendix C Anatomy of a modular multiplication

Fig. 3: Anatomy of a C function computing a modular multiplication over 256-bit integers using Gnu MP (obtained from gprof).

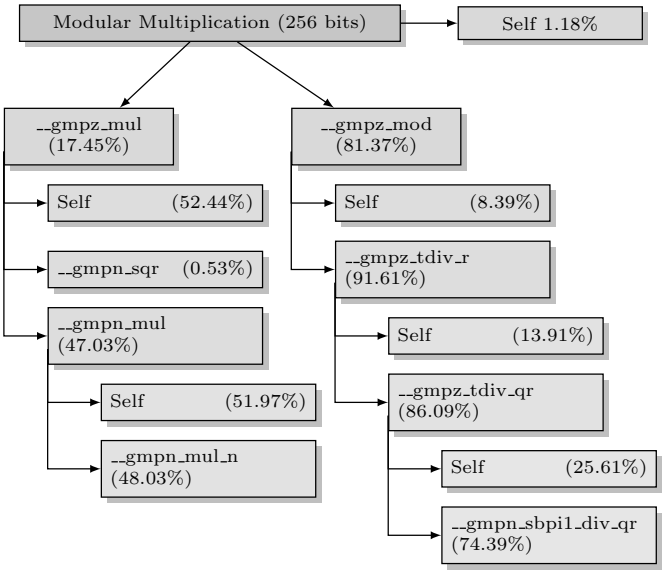
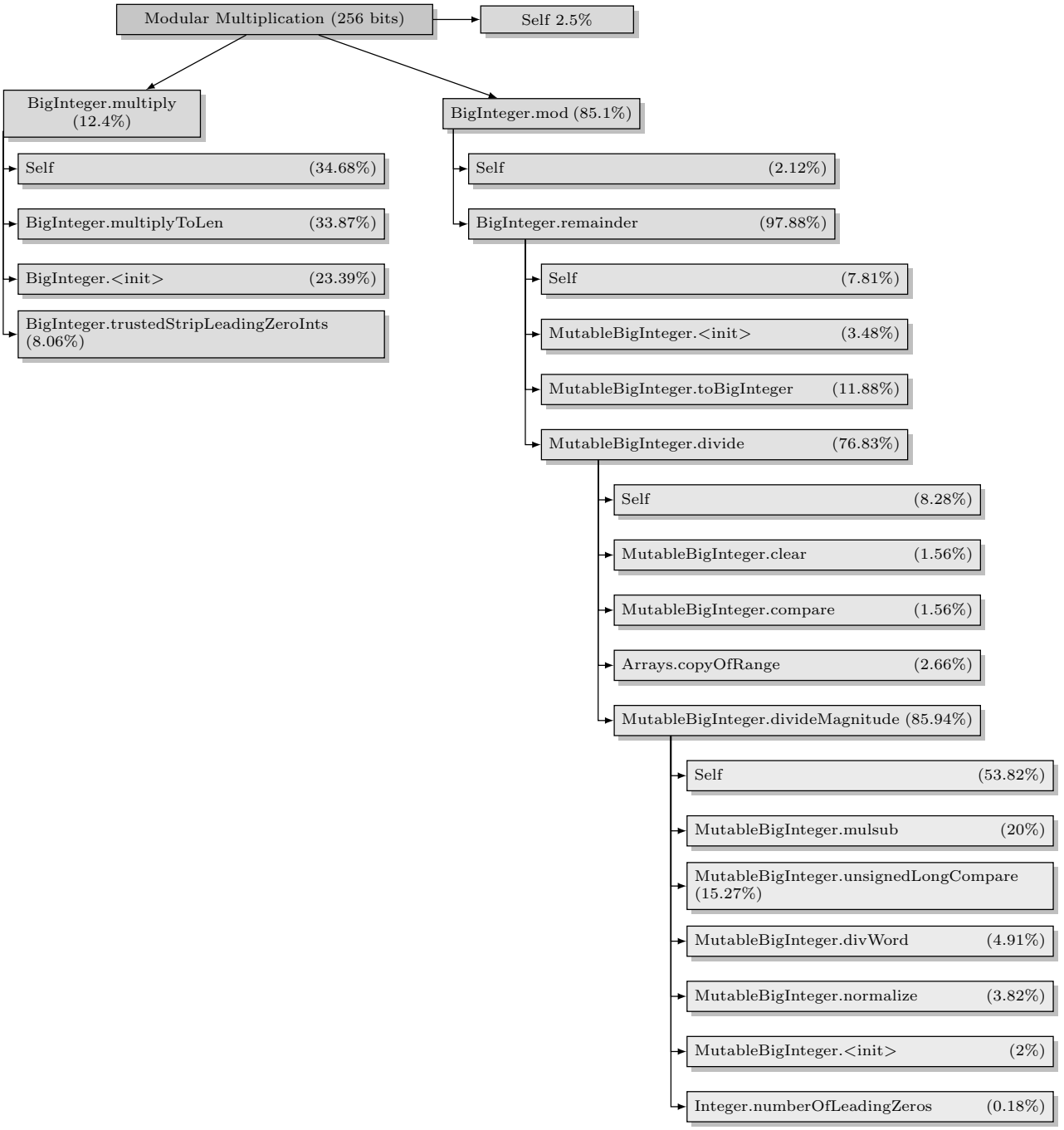


Fig. 4: Anatomy of a Java method computing a modular multiplication over 256-bit integers using Big Integer Java library on an x64 platform (obtained from Netbeans profiler).



A New Approach to Polynomial Modular Number System Internal Reduction

Nicolas Méloni
Université de Toulon
Institut de Mathématiques de Toulon
Toulon, France
nicolas.meloni@univ-tln.fr

***Abstract*—The Polynomial Modular Number System (PMNS) is an alternative to the binary multi-precision representation that allows to transport the arithmetic of a finite field to a polynomial ring. The most important operation in that system is the internal reduction that follows any arithmetic operation. All recent works on the subject use the same algorithm derived from Montgomery's modular multiplications to perform this internal reduction. This paper designs and analyzes two new algorithms to perform the internal reduction, both based on Babai's Closest Vector algorithms. It allows to significantly reduce the number of additions needed to perform this operation. A comprehensive experimental analysis shows that one of those algorithms is also faster in practice. For that matter, a C code generation tool has been developed in order to produce implementations for any prime number field.**

***Index Terms*—Finite field arithmetic, Polynomial Modular Number System, Integer Lattice.**

I. Introduction

Field arithmetic is a key component of many modern cryptosystems. The implementation of prime field in particular has received a lot of attention for the past decades. The most common way to represent a field element is to use a 2^w-ary representation. In such a positional number system, given a prime p, an element a of $\mathbb{Z}/p\mathbb{Z}$ is represented as a $n = \lceil \log(p) \rceil + 1$ array $(a_0, \ldots, a_{n-1})$ corresponding to the integer

$$a = \sum_{i=0}^{n-1} a_i 2^{wi}$$

with $0 \leq a_i \leq 2^w$ or $-2^{w-1} \leq a_i \leq 2^{w-1} - 1$. All arithmetic operations are then performed modulo p. Among those operations modular multiplication has become the main focus of investigation as most cryptosystems heavily rely on it. It is usually performed as a standard integer multiplication followed by a modular reduction. Many of approaches have been proposed to speed up modular reduction modulo a prime number. Two main cases are usually considered, depending on whether the prime can be freely chosen or not. A typical example of the former case is that of a Mersenne prime, that is a prime number of the form $2^k - 1$. In that case a modular reduction can be easily computed as a shift-and-add operation. Generalizations of Mersenne primes have been proposed [7], [15] in order to extend the range of possible candidates. Those methods usually provide extremely fast modular reductions but are limited to a small number of primes. In the latter case, when the cryptosystem does not allow enough freedom to chose a generalized Mersenne number, more generic algorithms must be used [5], [13].

The Polynomial Modular Number System (PMNS) is an alternative to the standard radix 2^w multi-precision representation. It was introduced by Bajard, Imbert and Plantard [2] in order to speed up modular arithmetic when special primes are not available. A field element is represented as a polynomial of bounded degree and field arithmetic is performed using polynomial arithmetic. An element a of $\mathbb{Z}/p\mathbb{Z}$ is represented as a polynomial

$$A(X) = \sum_{i=0}^{n-1} a_i X^i \mod E(X)$$

where E is a degree n polynomial and $|a_i|\rho$ for some ρ. One key point is that, even if p has no special form, the polynomial E can in general be chosen so that polynomial reduction modulo E is fast (for instance if $E = X^n - 1$). In that case, the modular reduction that occurs in positional number system is replaced by a fast polynomial reduction (usually called external reduction) followed by a coefficient reduction (called internal reduction). The latter operation is the critical point. Two approaches have been proposed to perform it based either on Barrett modular reduction algorithm [3] or Montgomery's one [14]. During the past few years improvements have been made on the implementation [10], generation [4], [8], randomization [9] and generalization [6], [11] of PMNS in various contexts. One interesting common feature between all those works is that they all perform the internal reduction step using the Montgomery-like approach.

In this paper we propose two new approaches to perform the internal reduction based on two algorithms initially proposed by Babai to solve the Closest Vector Problem for integer lattices [1]. Those two algorithms have been mentioned before in the literature in order to obtain theoretical bounds on the size of the coefficients [4] but discarded as a practical tools due their high computational complexities [9]. We show that it is actually possible to design competitive algorithms based on Babai's ideas achieving faster internal reduction in many cases.

The rest of the paper is structured as follows. Section II is a review of PMNS and Babai's Closest Vector algorithms. In Section III we propose and study two new internal reduction algorithms and two optimized versions of those. In Section IV we give implementation details about our code generation tool targeting 64-bit architectures and we illustrate the efficiency of our approach with experimental results and comparisons with a similar tool based on Montgomery reduction [10].

II. Mathematical background

A. Lattices

A lattice is a sub-group of $\mathbb{R}^n$. It can be seen as the set of all linear combinations with integer coefficients of any set of linearly independent vectors. Typically, a lattice can be seen as the set

$$\Lambda = \left\{ \sum_{i=0}^{n-1} a_i \mathbf{b}_i \mid a_i \in \mathbb{Z} \right\}$$

where $(\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ is a basis of $\mathbb{R}^n$.

Volume of a lattice.: Any two basis of a lattice Λ can be obtained from one another using a integral change-of-basis matrix of determinant ± 1. Thus the absolute value of the determinant of any basis only depends on Λ. That determinant is called the *volume* of the lattice and is noted $\det(\Lambda)$.

Gram-Schmidt orthogonalization.: Let $\mathbf{B} = (\mathbf{b}_0, \mathbf{b}_1, \ldots, \mathbf{b}_{n-1})$ a basis. Its *Gram-Schmidt orthogonalization* (GSO) $\tilde{\mathbf{B}} = (\tilde{\mathbf{b}}_0, \tilde{\mathbf{b}}_1, \ldots, \tilde{\mathbf{b}}_{n-1})$ is the orthogonal family obtained using the recursive process:

1) $\tilde{\mathbf{b}}_0 = \mathbf{b}_0$
2) $\forall i \geq 1, \tilde{\mathbf{b}}_i = \mathbf{b}_i - \sum_{j=0}^{i-1} \mu_{ij} \tilde{\mathbf{b}}_\mathbf{j}$

where $\mu_{ij} = \frac{\langle \mathbf{b}_i, \tilde{\mathbf{b}}_j \rangle}{\|\tilde{\mathbf{b}}_j\|_2^2}$. Let Λ be the lattice spanned by $\mathbf{B}$, a classical result is that it satisfies

$$\det(\Lambda) = \prod_{i=0}^{n-1} \|\tilde{\mathbf{b}}_i\|_2.$$

LLL algorithm.: The *LLL* algorithm [12] is a lattice basis reduction process that computes an *LLL*-reduced basis in polynomial time. For a given basis $\mathbf{B}$ and a real number $\frac{1}{4} < \delta < 1$, $\mathbf{B}$ is *LLL*-reduced if

- for $0 \leq j \leq i \leq n-1 : |\mu_{ij}| \leq 0.5$
- for $1 \leq i \leq n-1 : \delta \|\tilde{\mathbf{b}}_{i-1}\|_2^2 \leq \|\tilde{\mathbf{b}}_i\|_2^2 + \mu_{i,i-1} \|\tilde{\mathbf{b}}_{i-1}\|_2^2$.

In practice it means that the basis has short vectors and that they are nearly orthogonal.

B. PMNS

Let p be a prime number, E a polynomial of degree n and γ a root of E modulo p. The general idea is to represent integers as elements of the polynomial quotient ring $\mathbb{Z}[X]/(E)$. Any arithmetic operation can then be performed using standard arithmetic over the polynomials followed by a euclidean division by E.

Definition 1: A Polynomial Modular Number System (PMNS) $\mathcal{B}$ is defined by a tuple $(p, n, \gamma, \rho, E, l) \in \mathbb{Z}^4 \times \mathbb{Z}[X] \times \mathbb{Z} \cup \{\infty\}$, such that $E(\gamma) \equiv 0 \mod p$ and for every integer $0 \leq x < p$, there exists a polynomial $V(X)$ in $\mathbb{Z}[X]/(E)$ (or equivalently a vector $V = (v_0, ..., v_{n-1}) \in \mathbb{Z}^n$) such that:

$$V(\gamma) \equiv \sum_{i=0}^{n-1} v_i \gamma^i \equiv x \mod p \text{ and } \|V\|_l < \rho,$$

where $\|.\|_l$ is the l-norm of $\mathbb{R}^n$.

Such a polynomial (or vector) is said to be a representative of x and we notate $V \equiv_\mathcal{B} x$. Their might be several representatives for a given x. If V' is another polynomial such that $V' \equiv_\mathcal{B} x$ we say that V and V' are equivalent an notate $V \equiv_\mathcal{B} V'$.

Remark 1: In all previous work, the only norm considered was the infinity norm. In other words the parameter l was fixed to ∞ so that $\|V\|_l = \|V\|_\infty = \max(|v_i|)$. However in this paper the 1 and 2-norm will also be considered.

Remark 2: In [2] and [3], a naming distinction is made based on the shape of the polynomial E. The name PMNS is reserved to the case where $E(X) = X^n - \alpha X - \lambda$. When $\alpha = 0$ the system is called Adapted Modular Number System (AMNS) and simply MNS otherwise. We make no such distinction in this paper and only use the concept of PMNS.

Example 1: Let $p = 1048573, n = 5, \gamma = 238019, \rho = 36, E = X^5 - 2$ then $\mathcal{B} = (p, n, \gamma, \rho, E, \infty)$ is a PMNS. Let us consider two polynomials $A = -9X^4 + 13X^3 - 35X^2 - 25$ and $B = 24X^4 - 32X^3 + 21X^2 - 2X - 16$, their modular multiplication in PMNS is performed in 3 steps:

1) a standard polynomial multiplication

$$\begin{aligned} AB = &-216X^8 + 600X^7 - 1445X^6 + 1411X^5 \\ &- 1217X^4 + 662X^3 + 35X^2 + 50X + 400 \end{aligned}$$

2) a degree reduction (external reduction)

$$\begin{aligned} C = &AB \mod E \\ = &-1217X^4 + 230X^3 + 1235X^2 - 2840X + 3222 \end{aligned}$$

3) a coefficient reduction (internal reduction) to ensure the coefficients (c'_i) of the results satisfy the bound $|c'_i| < \rho$

$$C \equiv_\mathcal{B} C' = 3X^4 + 2X^3 + 8X^2 - X - 1.$$

One can verify that $A(\gamma)B(\gamma) \equiv C(\gamma) \equiv C'(\gamma) \mod p$. The first two steps are usually performed using standard algorithms and often merge as one operation. The third step is the most difficult from a computational point of view. The general method is to find a polynomial R, close to C (depending on the chosen norm), such that $R(\gamma) \equiv 0 \mod p$, in which case $C' = C - R \equiv_\mathcal{B} C$ is and $\|C'\|$ is supposed to be small. This problem can be interpreted as finding a close vector to the vector C in some integer lattice as shown next.

Definition 2: Let $\mathcal{B} = (p, n, \gamma, \rho, E, l)$ be a PMNS. The lattice $\mathcal{L}(\mathcal{B})$ associated to $\mathcal{B}$ is the set

$$\mathcal{L}(\mathcal{B}) = \left\{ (v_0, \ldots, v_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} v_i \gamma^i \equiv 0 \mod p \right\}.$$

$\mathcal{L}(\mathcal{B})$ can also be seen as the set of all polynomials $V \in \mathbb{Z}[X]$ of degree less than n such that $V(\gamma) \equiv 0 \mod p$. This set is introduced in [2] and it is proven that it is a lattice of dimension n, generated by the row of the matrix

$$G = \begin{pmatrix} p & 0 & 0 & \dots & 0 \\ -\gamma & 1 & 0 & \dots & 0 \\ -\gamma^2 & 0 & 1 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ -\gamma^{n-2} & 0 & \dots & 1 & 0 \\ -\gamma^{n-1} & 0 & \dots & 0 & 1 \end{pmatrix}.$$

Let $(\mathbf{b}_0, \dots, \mathbf{b}_{n-1})$ be any basis of $\mathcal{L}(\mathcal{B})$ and $C = (c_0, \dots, c_{n-1}) \in \mathbb{Z}^n$. The coefficient reduction problem consists in finding a close vector to C that is a linear combination of the $\mathbf{b}_i$'s. We define the PMNS reduction problem as follow.

Definition 3: **PMNS reduction problem**

Given a PMNS $\mathcal{B} = (p, n, \gamma, \rho, E, l)$ and a vector $v \in \mathbb{Z}^n$, find s in $\mathcal{L}(\mathcal{B})$ such that

$$\|v - s\|_l < \rho.$$

This is obviously linked to the Closest Vector Problem (CVP) that is known to be NP-hard for the ∞-norm [17]. Thankfully we have to deal with a much weaker version of that problem: n is small in practice and we only look for a *close enough* vector to v.

C. Montgomery PMNS Reduction

The most efficient method to solve the *PMNS reduction problem* is a Montgomery-like algorithm proposed by Nègre and Plantard [14]. It actually solves a slightly different version of the problem and requires the existence of a polynomial M with certain properties as studied in [10]. The method is summarized by Algorithm 1.

Algorithm 1: RedCoeff ([14])

Data: $\mathcal{B} = (p, n, \gamma, \rho, E, \infty), V \in \mathbb{Z}_n[X], M \in \mathcal{B}$ such that, $M(\gamma) \equiv 0 \mod p$, $\phi \in \mathbb{N} - \{0\}$ and $M' = -M^{-1} \mod (E, \phi)$.
Result: $S(\gamma) = V(\gamma)\phi^{-1} \mod p$
$Q \leftarrow V \times M' \mod (E, \phi)$;
$T \leftarrow Q \times M \mod E$;
$S \leftarrow (V + T)/\phi$;
return S;

D. Babai's Nearest Plane algorithm

Let $\mathbf{B} = (\mathbf{b}_0, \dots, \mathbf{b}_{n-1})$ a lattice basis and $\tilde{\mathbf{B}}$ its GSO. Recall that $\tilde{\mathbf{B}}$ satisfies

$$\tilde{\mathbf{b}}_i \perp \mathbf{span}(\mathbf{b}_0, \dots, \mathbf{b}_{i-1})$$

and

$$\mathbf{span}(\tilde{\mathbf{b}}_0, \dots, \tilde{\mathbf{b}}_i) = \mathbf{span}(\mathbf{b}_0, \dots, \mathbf{b}_i)$$

Babai's nearest hyperplane algorithm [1] can be seen as a way to reduce each coordinate of v in the Gram-Schmidt basis one by one, starting from coefficient $n-1$ to 0.

Suppose we have a semi-reduced vector v' whose coefficient $i+1$ to $n-1$ are reduced in the Gram-Schmidt basis. Reducing coefficient i is done by computing the integer c_i such that $v' - c_i\mathbf{b}_i$ has the smallest i-th coefficient possible in the Gram-Schmidt basis. In other terms c_i must minimize

$$< v', \tilde{\mathbf{b}}_i > - c_i < \mathbf{b}_i, \tilde{\mathbf{b}}_i >$$

which leads to

$$c_i = \left\lfloor \frac{< v', \tilde{\mathbf{b}}_i >}{\|\tilde{\mathbf{b}}_i\|_2^2} \right\rceil.$$

Babai's Nearest Plane algorithm is then straightforward as shown by Algorithm 2.

Algorithm 2: Babai's Nearest Plane algorithm

Data: $\mathcal{L}$ a lattice, $v \in \mathbb{Z}^n$, $B = (\mathbf{b}_0, \dots, \mathbf{b}_{n-1})$ a basis and $\tilde{B} = (\tilde{\mathbf{b}}_0, \tilde{\mathbf{b}}_1, \dots, \tilde{\mathbf{b}}_{n-1})$ its Gram-Schmidt orthogonalization
Result: $s \in \mathbb{Z}^n$ such that $v' = v - s \in \mathcal{L}$ and v' is close to v
$s \leftarrow v$;
for $i = n-1 \dots 0$ **do**
  $c \leftarrow \left\lfloor \frac{<s, \tilde{\mathbf{b}}_i>}{\|\tilde{\mathbf{b}}_i\|_2^2} \right\rceil$;
  $s \leftarrow s - c \times \mathbf{b}_i$
return s;

The output vector s satisfies

$$s = \sum_{i=0}^{n-1} s_i \tilde{\mathbf{b}}_i \text{ with } |s_i| \leq 1/2$$

from which we can deduce that

$$\|s\|_2 \leq \frac{1}{2} \sqrt{\sum_{i=0}^{n-1} \|\tilde{\mathbf{b}}_i\|_2^2}.$$

That last inequality gives, for a given PMNS, a maximum bound for ρ that depends on the basis of $\mathcal{L}(\mathcal{B})$. In particular, the smaller $\sum_{i=0}^{n-1} \|\tilde{\mathbf{b}}_i\|_2^2$ is , the better. As $\det(\mathcal{L}) = \prod \|\tilde{\mathbf{b}}_i\| = p$ is a constant of the lattice, a *good* basis should be one whose vectors satisfies $\|\mathbf{b_i}\| \simeq p^{1/n}$. This is typically the kind of basis that the *LLL* algorithm provides when the dimension n is small.

E. Babai's Rounding algorithm

Let $\mathbf{B} = (\mathbf{b}_0, \dots, \mathbf{b}_{n-1})$ a lattice basis and v a vector. Babai's Rounding algorithm [1] consists of computing the rational coordinates of v in $\mathbf{B}$, so that $v = q_0\mathbf{b}_0 + \dots + q_{n-1}\mathbf{b}_{n-1}$, and choose $c = \lfloor q_0 \rceil \mathbf{b}_0 + \dots + \lfloor q_{n-1} \rceil \mathbf{b}_{n-1}$ as a close vector. The rational coordinates can be obtained by precomputing the inverse matrix $\mathbf{B}^{-1} = (\mathbf{b'}_0, \dots, \mathbf{b'}_{n-1})$ so that

$c = v\mathbf{B}^{-1}$. The whole procedure can be reduced to a simple equation:

$$s = v - \lfloor v\mathbf{B}^{-1} \rceil \mathbf{B}.$$

Babai's Rounding algorithm is described in Algorithm 3.

Algorithm 3: Babai's Rounding algorithm

Data: $\mathcal{L}$ a lattice, $v \in \mathbb{Z}^n$, $\mathbf{B} = (\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$ a basis and $\mathbf{B^{-1}} = (\mathbf{b'}_0, \mathbf{b'}_1, \ldots, \mathbf{b'}_{n-1})$ its inverse
Result: $s \in \mathbb{Z}^n$ such that $v' = v - s \in \mathcal{L}$ and v' is close to v

```
s ← v;
for i = 0 ... n − 1 do
    r ← ⌊< v, b'_i >⌉;
    s ← s − r b_i;
return s;
```

The output vector $s = (q_0 - \lfloor q_0 \rceil)\mathbf{b}_0 + \cdots + (q_{n-1} - \lfloor q_{n-1} \rceil)\mathbf{b}_{n-1}$ satisfies

$$s = \sum_{i=0}^{n-1} s_i \mathbf{b}_i \text{ with } |s_i| \leq 1/2$$

from which we can deduce that

$$\|s\|_1 \leq \frac{1}{2} \sum_{i=0}^{n-1} \|\mathbf{b}_i\|_1.$$

The last inequality gives, for a given PMNS, a maximum bound for ρ that depends on the basis of $\mathcal{L}(\mathcal{B})$. In particular, the smaller $\sum_{i=0}^{n-1} \|\mathbf{b}_i\|_1$ is , the better.

III. Babai PMNS reduction algorithms

Babai's CVP algorithms can not be used as such to perform the internal reduction of a modular reduction because of the size of their operands. The coefficients of the GSO and the inverse matrix are fractional numbers whose numerators and denominators can be as large as the prime number p. This section describes two algorithms based on Babai's Closet Vector algorithms adapted to the PMNS reduction problem.

A. Nearest Plane PMNS internal reduction algorithm

At its core, Algorithm 2 consists of computing a good integer approximation of $\frac{<s, \tilde{\mathbf{b}}_i>}{\|\tilde{\mathbf{b}}_i\|_2^2}$. Our goal is to compute a good enough approximation of these quantities using coefficients that fit in the registers of a target processor.

We first remark that $\left\| \frac{\tilde{\mathbf{b}}_i}{\|\tilde{\mathbf{b}}_i\|_2^2} \right\|_2 = \|\tilde{\mathbf{b}}_i\|_2^{-1}$. If the basis $\mathbf{B}$ is LLL-reduced, it is expected that, for all i, $\|\tilde{\mathbf{b}}_i\|_2 \simeq p^{1/n}$. This justifies that there should be an integer h_1 so that $\left\lfloor \frac{2^{h_1} \tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil$ fit in a w-bit register of all i, j.

Let us suppose that there are two integers h_1 and h_2 such that $\left\lfloor \frac{2^{h_1} \tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil$ and $\lfloor \frac{s_j}{2^{h_2}} \rceil$ are w-bit integers for all i and j. In that case

$$\begin{aligned} < s, \frac{\tilde{b}_i}{\|\tilde{b}_i\|^2} > &= \sum_j s_j \frac{\tilde{b}_{i,j}}{\|\tilde{b}_i\|^2} \\ &= \frac{1}{2^{h_1 - h_2}} \sum_j \frac{s_j}{2^{h_2}} \times \frac{2^{h_1} \tilde{b}_{i,j}}{\|\tilde{b}_i\|^2} \\ &\simeq \left\lfloor \frac{\sum_j \lfloor \frac{s_j}{2^{h_2}} \rceil \times \left\lfloor \frac{2^{h_1} \tilde{b}_{i,j}}{\|\tilde{b}_i\|^2} \right\rceil}{2^{h_1 - h_2}} \right\rfloor \end{aligned}$$

In other words, given h_1 and h_2, it is possible to approximate the coefficients in Babai's Nearest Plane algorithm through register-size integer arithmetic. Algorithm 4 describe a complete reduction procedure.

Algorithm 4: Nearest Plane Reduction Algorithm

Data: $\mathcal{L}$ a lattice defined by a basis $(\mathbf{b}_0, \ldots, \mathbf{b}_{n-1})$, $(\tilde{\mathbf{b}}_0, \ldots, \tilde{\mathbf{b}}_{n-1})$ its GSO, $G_{i,j} = \left\lfloor \frac{2^{h_1} \tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil$ for $0 \leq i, j < n$,
Input: $v \in \mathbb{Z}^n$
Result: s equivalent to v with "*small*" coefficients

```
s ← v;
for i = n − 1 ... 0 do
    c ← 0;
    for j = 0 ... n − 1 do
        r ← s_j ≫ h2;
        c ← c + r × G_ij
    r ← c ≫ (h_1 − h_2);
    s ← s − r × b_i;
return s;
```

The output $s = (s_0, \ldots, s_{n-1})$ satisfies

$$s = \sum_{i=0}^{n-1} s_i \tilde{\mathbf{b}}_i$$

with each s_i satisfying $|s_i| \leq e_i$ for some e_i depending on the approximation error made during the computation of c.

Complexity analysis: One can remark that $\left\lfloor \frac{2^{h_1} \tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil$ does not depend on the input vector v and thus can be pre-computed for all i, j. Let $\mathbf{add}_w$ and $\mathbf{mult}_w$ represent the cost a one addition and one multiplication of two w-bit integers and **shft** that of a shift. For each iteration of the main loop, the inner loop computes n **shft**, n $\mathbf{mult}_w$ and $(n-1)$ $\mathbf{add}_{2w} = (2n-2)$ $\mathbf{add}_w$. It is followed by a addition **shft** and n $\mathbf{mult}_w$ and n $\mathbf{add}_{2w}$. The total computational cost is

$$2n^2 \, \mathbf{mult}_w + (4n^2 - 2n) \, \mathbf{add}_w + (n^2 + n) \, \mathbf{shft}\,.$$

It is possible to save a few additions from the observation that the last vector subtraction of the last iteration ($s \leftarrow s - r \times \mathbf{b_0}$) produces a small vector, that is one with coefficients fitting in one register. It means that the upper part of each subtraction can be discarded, saving n $\mathbf{add}_w$. The complexity of Algorithm 4 thus becomes

$$2n^2\,\mathbf{mult}_w + (4n^2-3n)\,\mathbf{add}_w + (n^2+n)\,\mathbf{shft}\,.$$

Coefficient reduction: We need to prove that Algorithm 4 efficiently solves the PMNS reduction problem. To do so, we must evaluate the error made during the various approximations.

Lemma 1: Let $s^{(n-1)}$ be the initial value of variable s and $s^{(n-k-1)}$ its value at the beginning of the k-th iteration of Algorithm 4. Let $S = \sum_{i=0}^{n-1} S_i \tilde{\mathbf{b}}_i$ be the output of Algorithm 4. Then for all i there exists a constant K_i that does not depend on the input v such that

$$|S_i| \leq e_i = \frac{\|s^{(i)}\|_1}{2^{h_1+1}} + K_i.$$

Proof: First, we know that $\tilde{\mathbf{b}}_i \perp \mathbf{span}(\mathbf{b}_0, \ldots, \mathbf{b}_{i-1})$. It means that, instruction the $s \leftarrow s - r \times \mathbf{b}_i$ do not modify the value of S_k for any $k > i$. In other terms, S_i only depends on the value of the vector $s^{(i)}$.

Set

$$R_i = \left\lfloor \frac{\sum_j \left\lfloor \frac{s_j^{(i)}}{2^{h_2}} \right\rfloor \times \left\lfloor \frac{2^{h_1}\tilde{\mathbf{b}}_{ij}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil}{2^{h_1-h_2}} \right\rceil.$$

Let e_i be the approximation error:

$$e_i = \left| \sum_j s_j^{(i)} \frac{\tilde{\mathbf{b}}_{i,j}}{\|\tilde{\mathbf{b}}_i\|^2} - R_i \right|.$$

We have

$$\begin{aligned} R_i &= \frac{1}{2^{h_1-h_2}} \sum_j \left(\frac{s_j^{(i)}}{2^{h_2}} - \varepsilon_j \right) \left(\frac{2^{h_1}\tilde{\mathbf{b}}_{ij}}{\|\tilde{b}_i\|^2} - \varepsilon'_j \right) + \varepsilon \\ &= \frac{1}{2^{h_1-h_2}} \sum_j \left(\frac{s_j^{(i)}}{2^{h_2}} \times \frac{2^{h_1}\tilde{\mathbf{b}}_{ij}}{\|\tilde{\mathbf{b}}_i\|^2} \right) \\ &\quad - \frac{1}{2^{h_1-h_2}} \sum_j \left(\varepsilon_j \times \frac{2^{h_1}\tilde{\mathbf{b}}_{ij}}{\|\tilde{\mathbf{b}}_i\|^2} \right) \\ &\quad - \frac{1}{2^{h_1-h_2}} \sum_j \left(\varepsilon'_j \times \frac{s_j^{(i)}}{2^{h_2}} \right) \\ &\quad + \frac{1}{2^{h_1-h_2}} \sum_j \varepsilon_j \varepsilon'_j + \varepsilon \end{aligned}$$

with $0 \leq \varepsilon, \varepsilon_j < 1$ and $-1/2 \leq \varepsilon'_j < 1/2$. Remark that

$$\begin{aligned} \left| \sum_j \left(\varepsilon'_j \times \frac{s_j^{(i)}}{2^{h_2}} \right) \right| &\leq \frac{\sum_j |s_j^{(i)}|}{2^{h_2+1}} \\ &\leq \frac{\|s^{(i)}\|_1}{2^{h_2+1}} \end{aligned}$$

and

$$\left| \sum_j \varepsilon_j \varepsilon'_j \right| \leq \frac{n}{2}$$

we obtain

$$e_i \leq \frac{\|s^{(i)}\|_1}{2^{h_1+1}} + \frac{2^{h_2}}{\|\tilde{b}_i\|^2} \left| \sum_j \tilde{b}_{i,j} \right| + \frac{n}{2^{h1-h2+1}} + 1.$$

Setting $K_i = \frac{2^{h_2}}{\|\tilde{\mathbf{b}}_i\|^2} \left| \sum_j \tilde{\mathbf{b}}_{i,j} \right| + \frac{n}{2^{h1-h2+1}} + 1$ gives the expected result. ∎

Despite the fact that the output vector has a small 2-norm, nothing guaranties that the intermediate values of s do not grow. It would be the case if the basis $\mathbf{B}$ was orthogonal as the norm of the vector s would get lower after each iteration.

In our context, the basis is LLL-reduced which means that it is close to being orthogonal and the norm of s is expected not to grow too much. Yet obtaining an effective bound remains an open problem.

For the rest of our analysis we simply remark that, once ρ and $\mathbf{B}$ are fixed there exists an absolute constant α such that $\max_i(\|s^{(i)}\|_1) \leq \alpha\|v\|_1$. All our experiments tend to show that, in our context, $\alpha \leq 2$, however we could not give a formal proof of this result.

Proposition 1: Let $\mu_2 = \sqrt{\sum \|\tilde{\mathbf{b}}_i\|_2^2}$. There exist two constants α and K such that

$$\|S\|_2 \leq \left(\frac{\alpha\|v\|_1}{2^{h_1+1}} + K \right) \mu_2.$$

Proof: We have seen that there exists α such that $\max_i(\|s^{(i)}\|_1) \leq \alpha\|v\|_1$. Then for all i

$$e_i \leq \frac{\alpha\|v\|_1}{2^{h_1+1}} + K.$$

where $K = \max K_i$. As the output satisfies $S = \sum_{i=0}^{n-1} s_i \tilde{\mathbf{b}}_i$ with $|s_i| \leq e_i$ we get that

$$\|S\|_2 \leq \left\| \sum_{i=0}^{n-1} \left(\frac{\alpha\|v\|_1}{2^{h_1+1}} + K \right) \tilde{\mathbf{b}}_i \right\|_2 \leq \left(\frac{\alpha\|v\|_1}{2^{h_1+1}} + K \right) \mu_2.$$

∎

Corollary 1: Let $(p, n, \gamma, \rho, E, 2)$ be a PMNS, α and K be given by Proposition 1. If v satisfies

$$\|v\|_1 \leq \frac{2^{h_1+1}}{\alpha} \left(\frac{\rho}{\mu_2} - K \right)$$

then Algorithm 4 solves the PMNS reduction problem.

Proof: Algorithm 4 solves the PMNS reduction problem if the output S satisfies $\|S\|_2 \leq \rho$. Combining that condition with Proposition 1 gives the expected result. ∎

B. Rounding PMNS reduction algorithm

Our approach is quite similar to that of the previous algorithm. Babai's Rounding method consists of computing an integer approximation of $v\mathbf{B}^{-1}$. The main idea is to approximate the rational coordinates of v in $\mathbf{B}$ using register-size approximations of the coefficients of the inverse matrix $\mathbf{B}^{-1} = (\mathbf{b}'_0, \ldots, \mathbf{b}'_{n-1})$. We introduce the same set of parameters, h_1 and h_2, so that $\lfloor 2^{h_1}\mathbf{b}'_{i,j} \rceil$ and $\lfloor \frac{v_j}{2^{h_2}} \rfloor$ are register-size integers. It is then possible to compute a short vector s based on the following equation

$$s = v - \left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor \mathbf{B}.$$

Algorithm 5 describes the complete procedure.

Algorithm 5: Rounding Reduction Algorithm

Data: $\mathcal{L}$ a lattice, $\mathbf{B}$ a basis saw as a row matrix, $\mathbf{B}^{-1}$ its inverse and for $0 \leq i, j < n$, $B'_{ij} = \lfloor 2^{h_1}\mathbf{b'}_{ij} \rceil$
Input: $v \in \mathbb{Z}^n$
Result: s equivalent to v with "*small*" coefficients

```
s ← v;
v' ← (v_0 ≫ h_2, ..., v_{n-1} ≫ h_2);
for i = 0 ... n − 1 do
    r ← 0;
    for j = 0 ... n − 1 do
        r ← r + v'_j × B'_{i,j}
    r ← r ≫ (h_1 − h_2);
    s ← s − r × b_i;
return s;
```

Complexity analysis: The algorithm performs a succession of dot products. A similar analysis to that of Algorithm 4 shows that the computational cost is

$$2n^2\,\mathbf{mult}_w + (4n^2 - 2n)\,\mathbf{add}_w + 2n\,\mathbf{shft}\,.$$

It is possible to save many additions from the observation the coefficients of final result are supposed to fit in a single register. The instruction $s \leftarrow s - r \times \mathbf{b}_i$ can be replaced by $s \leftarrow (s - r \times \mathbf{b}_i) \mod 2^w$ saving n additions per iteration. Algorithm 5 can thus be performed in

$$2n^2\,\mathbf{mult}_w + (3n^2 - 2n)\,\mathbf{add}_w + 2n\,\mathbf{shft}\,.$$

Lemma 2: Let $S = \sum_{i=0}^{n-1} S_i\mathbf{b}_i$ be the output of Algorithm 5 then for all $0 \leq i \leq n-1$ there exists a constant K_i that does not depend on the input v such that

$$|S_i| \leq \frac{\|v\|_1}{2^{h_1+1}} + K_i.$$

Proof: The output S satisfies

$$\begin{aligned} S &= v - \left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor \mathbf{B} \\ &= v\mathbf{B}^{-1}\mathbf{B} - \left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor \mathbf{B} \\ &= \left(v\mathbf{B}^{-1} - \left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor \right) \mathbf{B} \end{aligned}$$

On top of that, for a given i, consider the i-th coordinate R_i of the vector $\left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor$

$$\begin{aligned} R_i &= \left(\left\lfloor \frac{\lfloor \frac{v}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{B}^{-1} \rceil}{2^{h_1-h_2}} \right\rfloor \right)_i \\ &= \left\lfloor \frac{< \lfloor \frac{v}{2^{h_2}} \rfloor, \lfloor 2^{h_1}\mathbf{b}'_\mathbf{i} \rceil >}{2^{h_1-h_2}} \right\rfloor \\ &= \left\lfloor \frac{\sum_j^{n-1} \lfloor \frac{v_j}{2^{h_2}} \rfloor \lfloor 2^{h_1}\mathbf{b'}_{i,j} \rceil}{2^{h_1-h_2}} \right\rfloor \\ &= \frac{1}{2^{h_1-h_2}} \sum_j^{n-1} \left\lfloor \frac{v_j}{2^{h_2}} \right\rfloor \lfloor 2^{h_1}\mathbf{b'}_{i,j} \rceil - \varepsilon \\ &= \frac{1}{2^{h_1-h_2}} \sum_j^{n-1} \left(\frac{v_j}{2^{h_2}} - \varepsilon_i \right) \left(2^{h_1}\mathbf{b'}_{i,j} - \varepsilon'_j \right) - \varepsilon \\ &= \sum_j^{n-1} v_j\mathbf{b'}_{i,j} - \sum_j^{n-1} \varepsilon'_j \frac{v_j}{2^{h_1}} - \sum_j^{n-1} \varepsilon_j 2^{h_2}\mathbf{b'}_{i,j} \\ &\quad + \frac{1}{2^{h_1-h_2}} \sum_{j=0}^{n-1} \varepsilon_j\varepsilon'_j - \varepsilon \end{aligned}$$

with $0 \leq \varepsilon, \varepsilon_j < 1$ and $-1/2 \leq \varepsilon'_j < 1/2$. We deduce that

$$\begin{aligned} |S_i| &= \left| \sum_j^{n-1} v_j\mathbf{b'}_{i,j} - \left\lfloor \frac{< \lfloor \frac{v}{2^{h_2}} \rfloor, \lfloor 2^{h_1}\mathbf{b}'_\mathbf{i} \rceil >}{2^{h_1-h_2}} \right\rfloor \right| \\ &\leq \sum_j^{n-1} \frac{|v_j|}{2^{h_1+1}} + 2^{h_2} \left| \sum_j^{n-1} \mathbf{b'}_{i,j} \right| + \frac{n}{2^{h_1-h_2+1}} + 1 \\ &\leq \frac{\|v\|_1}{2^{h_1+1}} + K_i \end{aligned}$$

■

Proposition 2: Let $\mu_1 = \sum_i \|\mathbf{b}_i\|_1$. There is a constant K such that

$$\|S\|_1 \leq (\frac{\|v\|_1}{2^{h_1+1}} + K)\mu_1.$$

Proof:

From the previous lemma we have that $S = \sum_i S_i\mathbf{b}_i$ with $|S_i| \leq \frac{\|v\|_1}{2^{h_1+1}} + K_i$. Set $K = \max_i(K_i)$ we obtain that desired result. ■

Corollary 2: Let $(p, n, \gamma, \rho, E, 1)$ be a PMNS and K be given by Proposition 2. If v satisfies

$$\|v\|_1 \leq 2^{h_1+1}\left(\frac{\rho}{\mu_1} - K\right)$$

then Algorithm 5 solves the PMNS reduction problem.

C. Error/Speed trade-off

It is possible to speed-up both reduction algorithms at the cost of an increase of the error. In some situations, the error remains small enough so that the algorithms still solve the PMNS problem. It is based on the observation that, generally speaking,

$$\sum_i a_i \simeq \left\lfloor \frac{\sum_i \lfloor a_i 2^h \rfloor}{2^h} \right\rfloor \simeq \sum_i \left\lfloor \frac{\lfloor a_i 2^h \rfloor}{2^h} \right\rfloor .$$

The first approximation is more accurate but the second one perform additions on smaller integers at the cost of additional shifts. However if h can be chosen to be equal to w, the register size of the target architecture, those shifts can be considered as virtually free. Algorithms 6 and 7 describe two optimized versions of algorithms 4 and 5. In both cases $n^2 - n$ $\mathbf{add}_{2w}$ are replaced by $n^2 - n$ $\mathbf{add}_w$ at the cost of $n^2 - n$ (virtually free) **shft**. The $\mathbf{shft}_w$ notation is added to represent such w-bit right shifts. The respective complexities of Algorithm 6 and 7 are

$$2n^2\,\mathbf{mult}_w + (3n^2 - 2n)\,\mathbf{add}_w + n^2\,\mathbf{shft} + n^2\,\mathbf{shft}_w$$

and

$$2n^2\,\mathbf{mult}_w + (2n^2 - n)\,\mathbf{add}_w + n\,\mathbf{shft} + n^2\,\mathbf{shft}_w\,.$$

On the other hand for both algorithms the value of K_i from lemma 1 and 2 is modified to $K_i + n - 1$.

Algorithm 6: Optimized Nearest P.

```
s ← v;
for i = n − 1 ... 0 do
    c ← 0;
    for j = 0 ... n − 1 do
        r ← s_j ≫ h2;
        c ← c + ((r × G_ij) ≫ (h_1 − h_2))
    s ← s − r × b_i;
return s;
```

Table I summarizes the complexities of the different internal reduction algorithms. The optimized version of the Rounding method appears to be more efficient that all other methods and performs 33% less additions than Montgomery's algorithm. Even the standard version should perform slightly faster. The efficiency of the Nearest Plane method is more dependent on the target architecture and the relative cost between additions and shifts.

Algorithm 7: Optimized Rounding

```
s ← v;
v' ← (v_0 ≫ h_2, ..., v_{n−1} ≫ h_2);
for i = 0 ... n − 1 do
    r ← 0;
    for j = 0 ... n − 1 do
        r ← r + ((v'_j × B'_ij) ≫ (h_1 − h_2))
    s ← s − r × b_i;
return s;
```

Alg.	$\mathbf{mult}_w$	$\mathbf{add}_w$	**shft**	$\mathbf{shft}_w$
Montgomery [10]	$2n^2$	$3n^2 - n$	0	n
Nearest P.	$2n^2$	$4n^2 - 3n$	n^2	n
Nearest P. (opt)	$2n^2$	$3n^2 - 2n$	n^2	n^2
Rounding	$2n^2$	$3n^2 - 2n$	n	n
Rounding (opt)	$2n^2$	$2n^2 - n$	n	n^2

TABLE I
THEORETICAL COMPLEXITIES OF THE INTERNAL REDUCTION ALGORITHMS

IV. IMPLEMENTATION

This section is dedicated to implementing the previous algorithms on a 64-bit architecture in C. A code generation tool has been developed along with a parameter search tool in order to simplify any implementation of our algorithms on a given field. The software was build using `SageMath` [16] and available in the following **Git** repository

`https://plmlab.math.cnrs.fr/melon359/pmns`

A. Choice of the the polynomial E

The effectiveness of the different methods studied here depends on the value of the 1-norm of the input vector V as shown in Corollaries 1 and 2. Given two polynomials A and B such that $V = AB \mod E$, $\|V\|_1$ highly depends on the choice of the polynomial E. Indeed, the column vector V can be expressed as a matrix-vector product

$$\begin{pmatrix} v_0 \\ \vdots \\ v_{n-1} \end{pmatrix} = \begin{pmatrix} I_n & E_{n-1} \end{pmatrix} (A^C) \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix} = \mathcal{R}_{A,E} \begin{pmatrix} b_0 \\ \vdots \\ b_{n-1} \end{pmatrix}$$

where I_n is the $n \times n$ identity matrix , E_{n-1} is the $n \times (n-1)$ matrix whose column i corresponds to the vector $X^{n+i} \mod E(X)$ in the base $(1, X, \ldots, X^{n-1})$ and A^C is the $(2n-1) \times n$ matrix whose column i correspond to the vector $a_i X^i$ in the base $(1, X, \ldots, X^{2n-2})$.

For a matrix M, let

$$\|M\|_l = \sup_{\|x\|_l = 1} \|Mx\|_l$$

be the matrix norm induced by the vector norm $\|.\|_l$. Such norm satisfies $\forall x \in \mathbb{R}^n, \|Mx\|_l \leq \|M\|_l \|x\|_l$. For instance, the 1-matrix norm is the maximum absolute column sum of the matrix, so that $\|(I_n E_{n-1})\|_1 = \max(\|I_n\|_1, \|E_{n-1}\|_1) = \|E_{n-1}\|_1$. In our case we obtain the following inequalities

$$\begin{aligned}\|V\|_1 &\leq \|\mathcal{R}_{A,E}\|_1\|B\|_1 &(1)\\ &\leq \|(I_nE_{n-1})\|_1\|(A^C)\|_1\|B\|_1 &(2)\\ &\leq \|E_{n-1}\|_1\|A\|_1\|B\|_1 &(3)\\ &\leq n\times\|E_{n-1}\|_1\|A\|_2\|B\|_2. &(4)\end{aligned}$$

Example 2: Consider $A = a_0 + a_1X + a_2X^2$, $B = b_0 + b_1X + b_2X^2$ and $E = X^3 - \lambda$. Then $X^4 \mod E = \lambda$ and $X^5 \mod E = \lambda X$. Considered as column vector in base $(1, X, X^2)$ we get that $E_2 = \begin{pmatrix}\lambda & 0\\ 0 & \lambda\\ 0 & 0\end{pmatrix}$

Let $V = AB \mod E$ we have

$$\begin{aligned}\begin{pmatrix}v_0\\v_1\\v_2\end{pmatrix} &= \begin{pmatrix}1&0&0&\lambda&0\\0&1&0&0&\lambda\\0&0&1&0&0\end{pmatrix}\begin{pmatrix}a_0&0&0\\a_1&a_0&0\\a_2&a_1&a_0\\0&a_2&a_1\\0&0&a_2\end{pmatrix}\begin{pmatrix}b_0\\b_1\\b_2\end{pmatrix}\\ &= \begin{pmatrix}a_0&\lambda a_2&\lambda a_1\\a_1&a_0&\lambda a_2\\a_2&a_1&a_0\end{pmatrix}\begin{pmatrix}b_0\\b_1\\b_2\end{pmatrix}\end{aligned}$$

and $\|V\|_1 \leq |\lambda|\|A\|_1\|B\|_1$.

Inequality (1) usually provides tighter bounds than (3) but is also harder to evaluate. In both cases E should be chosen so that the associated matrix E_{n-1} has the smallest norm. The rest of this work focuses on polynomials of the form $X^n - \lambda$ with $|\lambda|$ as small as possible in order to compare the algorithms presented here to that used in [9]. In that case the following theorem holds.

Theorem 1: Let $(p, n, \gamma, \rho, X^n - \lambda, l)$ be a PMNS. Let A, B and V be three degree $n-1$ polynomial and $V = AB \mod E$ with $\|A\|_l \leq \rho$ and $\|B\|_l \leq \rho$.

1) If $l = 1$, let K be given by Proposition 2. If
$$|\lambda|K\mu_1^2 \leq 2^{h_1-1}$$
then Algorithm 5 solves the PMNS reduction problem.
2) If $l = 2$, let α and K be given by Proposition 1. If
$$n\alpha|\lambda|K\mu_2^2 \leq 2^{h_1-1}$$
then Algorithm 4 solves the PMNS reduction problem.

Proof: As $E = X^n - \lambda$, we have

$$E_{n-1} = \lambda\times\begin{pmatrix}&I_{n-1}&\\0&\cdots&0\end{pmatrix} = \begin{pmatrix}\lambda&0&\cdots&0\\0&\lambda&\ddots&\vdots\\\vdots&\ddots&\ddots&0\\0&\cdots&0&\lambda\\0&\cdots&\cdots&0\end{pmatrix}$$

so that $\|E_{n-1}\|_1 = |\lambda|$.

- If $l = 1$, combining Corollary 2 with the inequality 3 gives that Algorithm 5 solves the PMNS problem if
$$\left(\frac{|\lambda|\mu_1}{2^{h_1+1}}\right)\rho^2 - \rho + K\mu_1 \leq 0$$
is satisfied. The real function $x \mapsto ax^2 - x + c$ reaches its minimum when $x = 1/2a$. We deduce that the previous inequality is satisfied as soon as
$$\left(\frac{|\lambda|\mu_1}{2^{h_1+1}}\right)\left(\frac{2^{h_1}}{|\lambda|\mu_1}\right)^2 - \left(\frac{2^{h_1}}{|\lambda|\mu_1}\right) + K\mu_1 \leq 0$$
that is
$$|\lambda|K\mu_1^2 \leq 2^{h_1-1}.$$
- If $l = 2$, combining Corollary 1 with the inequality 4 gives that Algorithm 4 solves the PMNS problem if
$$\left(\frac{n\alpha|\lambda|\mu_2}{2^{h_1+1}}\right)\rho^2 - \rho + K\mu_1 \leq 0$$
which is satisfied as soon as
$$n\alpha|\lambda|K\mu_2^2 \leq 2^{h_1-1}$$

■

B. Parameter generation

Let p be a prime number. Generating the parameters for p is done in the following sequence:

1) chose n the degree of the reduction polynomial E. Targeting a 64-bit architecture, n must satisfy $64 \times n \geq \lceil \log_2(p) \rceil$;
2) find λ such that $E = X^n \pm \lambda$ has a root γ in $\mathbb{F}_p$;
3) compute the lattice associated to the polynomial E and a LLL-reduced basis $\mathbf{B}$;
4) compute the GSO $(\tilde{\mathbf{b}}_\mathbf{0}, \ldots, \tilde{\mathbf{b}}_{\mathbf{n-1}})$ and inverse matrix $\mathbf{B}^{-1} = (\mathbf{b}'_0, \ldots, \mathbf{b}'_{n-1})$ of $\mathbf{B}$;
5) for each algorithm (Nearest plane and Rounding) find a pair of integer (h_1, h_2) such that
   - for all i, j $\left\lfloor \frac{2^{h_1}\tilde{\mathbf{b}}_{ij}}{\|\tilde{\mathbf{b}}_i\|^2} \right\rceil$ or $\lfloor 2^{h_1}\mathbf{b}'_{ij} \rceil$ fit in a 64-bit register;
   - every coordinates of $\lfloor v/2^{h_2} \rfloor$ fit in a 64-bit register for any valid input vector v;
   - there exists a ρ satisfying Theorem 1;
6) If no parameters can be found, go to step 2 and change λ or to step 1 and increase n.

Remark 3: The search for h_1 and h_2 is performed through exhaustive search with some conditions to speed up the process. Set $M_1 = \max_{i,j}\left|\frac{\tilde{\mathbf{b}}_{ij}}{\|\tilde{\mathbf{b}}_i\|^2}\right|$ and $M_2 = \max_{i,j}|\mathbf{b}'_{ij}|$, then h_1 must satisfy $2^{h_1}M_k < 2^{63} - 1/2$ which gives a maximum bound for h_1.

Similarly, $\lfloor \|v\|_\infty/2^{h_2} \rfloor$ must fit in a 64-bit register and we know from the inequality that $\|v\|_\infty \leq \|v\|_1 \leq |\lambda|\|A\|_1\|B\|_1$ which gives a minimum bound on h_2.

Remark 4: When verifying that Theorem 1 holds it is possible to compute the exact values of the K_i from Lemmas 1 and 2 thus obtaining tighter bounds on the norm of the output vector.

C. Comparisons

In [10] the authors have provided numerous C implementations of Montgomery PMNS reduction algorithms for many key size relevant for elliptic curve cryptography (i.e., 192, 224, 256, 384 and 521 bits). The same prime numbers have been used to generate parameters for our reduction algorithms. For each prime p, we have generated 10000 pairs of elements of $\mathbb{Z}/p\mathbb{Z}$, converted then into the relevant PMNS representation and performed a polynomial multiplication followed by an external reduction. Then, for each algorithm, we have measured the number of clock cycles required to perform the internal reduction as well as the total number of instructions performed by the CPU. The experiment was performed on a Linux system powered by an *Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz* and the code was compiled with *gcc 9.3.0* using the *-O3* option. The benchmark test results are shown in Table II and III.

p size	192	224		256	384		512	
degree n	4	4	5	5	7	8	10	11
Montgomery [10]	60	63	86	83	155	206	300	357
Nearest P.	83	83	143	133	265	331	494	606
Nearest P. (opt)	70	70	108	117	196	248	405	494
Rounding	50	52	76	74	155	205	321	362
Rounding (opt)	45	45	64	70	148	199	291	344

TABLE II
NUMBER OF CPU CLOCK CYCLES TO PERFORM AN INTERNAL REDUCTION USING A POLYNOMIAL OF THE FORM $X^n - \lambda$

p size	192	224		256	384		512	
degree n	4	4	5	5	7	8	10	11
Montgomery [10]	191	188	255	254	496	649	966	1122
Nearest P.	228	214	388	357	793	1021	1624	2014
Nearest P. (opt)	194	199	298	304	538	743	1218	1486
Rounding	181	178	261	253	468	644	987	1021
Rounding (opt)	159	156	221	220	381	493	735	879

TABLE III
NUMBER OF CPU INSTRUCTIONS TO PERFORM AN INTERNAL REDUCTION USING A POLYNOMIAL OF THE FORM $X^n - \lambda$

First let us say that CPU clock cycle measurements should be taken with caution especially when numbers are that small. However, the measurements shown in II are very consistent from one execution to another with a standard deviation of less than one cycle.

Generally speaking, the results are consistent with the theoretical complexities from Table I. The Rounding and Optimized Rounding algorithms are faster than Montgomery reduction in most cases. On top of that the optimized version of the Rounding method executes 10 to 20% less instructions than any other approaches. The Nearest Plane algorithms however do not appear to be that competitive at the moment. These algorithms are significantly slower as shown in both tables and it clearly highlights that the numerous shifts, whose cost is often overlooked, have a great impact on the performances of core operations such as modular multiplications.

Another phenomenon is surprising: as the size of p increases, so does the degree of the polynomial E and the difference between the Rounding algorithm and the Montgomery-like algorithm tends to decrease. For large primes, Montgomery reduction becomes even faster than the standard Rounding method. The analysis of the instruction count is interesting for that matter. For instance, with $n = 11$, Table III confirms that the Rounding method executes less operations but Montgomery reduction ends up being faster. One explanation is that the super-scalar nature of modern processors combine with instruction pipelining and compiler optimizations can introduce a significant level of variability between theoretical analysis and practical results on a given platform. This is confirmed by the fact that compiling the same code without any optimization (*-O0* option of *gcc*) reverses the results with the Rounding algorithm being faster than Montgomery's.

Even more surprising is the fact that, for $n = 10$, the Rounding algorithm executes more instructions than the Montgomery-like counter-part. An analysis of the assembly code shows that the number of additions or multiplications are consistent with the theory but a large number of *move* instructions slows down the Rounding method. It appears that, in some cases, the compiler has a hard time optimizing memory allocation thus impacting the instruction count and execution speed.

V. CONCLUSIONS

The Polynomial Modular Number System have received a lot of interest recently. Various approaches and sets of parameters have been studied but the internal reduction step of all these works remains the same and is always based on a Montgomery-like algorithm originally proposed in [14]. In this work we have proposed two completely different internal reduction algorithms based Babai's Closest Vector algorithms for integer lattices. Our algorithms are both provided in two versions: one standard that aim at minimizing the size of the output vector and one optimized version where an error/speed trade-off is made.

Both algorithms are competitive with the previous works. In particular, algorithms based on Babai's rounding method are theoretically faster, the optimized version performing 33% less additions than Montgomery's. Experimental results comparing our implementations to those in [10] confirms the efficiency of our approach.

All our work has been carried in the most general case possible, with a prime p with no particular shape. Adapting our algorithms to special primes is one potential field for future studies. On top of that, our C code generation tool could be optimized depending on the targeted platform.

REFERENCES

[1] L. Babai. On lovász' lattice reduction and the nearest lattice point problem (shortened version). *Combinatorica*, 1986.

[2] Jean-Claude Bajard, Laurent Imbert, and Thomas Plantard. Arithmetic Operations in the Polynomial Modular Number System. In *ARITH'05: 17th IEEE Symposium on Computer Arithmetic*, pages 206–213, USA, 2005. IEEE computer society.

[3] Jean-Claude Bajard, Laurent Imbert, and Thomas Plantard. Modular Number Systems: Beyond the Mersenne Family. In *SAC'04: 11th International Workshop on Selected Areas in Cryptography*, number 3357 in Lecture Notes in Computer Science, pages 159–169, University of Waterloo, Ontario (Canada), August 2005. Springer Verlag.

[4] Jean-Claude Bajard, Jérémy Marrez, Thomas Plantard, and Pascal Véron. On Polynomial Modular Number Systems over Z/pZ. working paper or preprint, June 2020.

[5] Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology - CRYPTO '86, Santa Barbara, California, USA, 1986, Proceedings*, volume 263 of *Lecture Notes in Computer Science*, pages 311–323. Springer, 1986.

[6] Cyril Bouvier and Laurent Imbert. An Alternative Approach for SIDH Arithmetic. working paper or preprint, April 2021.

[7] Jaewook Chung and Anwar Hasan. More generalized mersenne numbers. In Mitsuru Matsui and Robert J. Zuccherato, editors, *Selected Areas in Cryptography*, pages 335–347, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[8] Titouan Coladon, Philippe Elbaz-Vincent, and Cyril Hugounenq. MPHELL: A fast and robust library with unified and versatile arithmetics for elliptic curves cryptography (extended version). In *ARITH 2021*, Transactions on Emerging Topics in Computing, Torino, Italy, June 2021.

[9] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, Nadia El Mrabet, Jérémy Marrez, and Pascal Véron. Randomization of Arithmetic over Polynomial Modular Number System. In *26th IEEE International Symposium on Computer Arithmetic*, volume 1 of *Proceedings of the 2019 IEEE 26th Symposium on Computer Arithmetic*, pages 199–206, Kyoto, Japan, June 2019. IEEE Computer Society.

[10] Laurent-Stéphane Didier, Fangan-Yssouf Dosso, and Pascal Véron. Efficient modular operations using the adapted modular number system. *Journal of Cryptographic Engineering*, January 2020.

[11] Nadia El Mrabet and Nicolas Gama. Efficient Multiplication over Extension Fields. In *WAIFI 2012*, Ghent, Belgium, July 2012.

[12] H.W. jr. Lenstra, A.K. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261:515–534, 1982.

[13] Peter L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[14] Christophe Negre and Thomas Plantard. Efficient modular arithmetic in adapted modular number system using lagrange representation. In Yi Mu, Willy Susilo, and Jennifer Seberry, editors, *Information Security and Privacy*, pages 463–477, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[15] J.A. Solinas. Generalized Mersenne Number. Technical report, Center for Applied Cryptographic Research, University of Waterloo, ON, 1999.

[16] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.3)*, 2021. `https://www.sagemath.org`.

[17] Peter van Emde Boas. Another np-complete problem and the complexity of computing short vectors in a lattice. Technical report, University of Amsterdam, Department of Mathematics,Netherlands, 1981.

Google
Google



Nicolas Méloni Nicolas Méloni received the MS degree in mathematics in 2004 and the PhD degree in computer science in 2007 from the Université Montpellier 2, France. He did postdoctoral work with the Centre for Applied Cryptographic Research at the University of Waterloo, Canada from 2008 to 2010. Since September 2010, he is an assistant professor at Université de Toulon, France. His research interests are in computer arithmetic and cryptography.

Manuscript submitted to AIMS Journal

doi:

PMNS FOR CRYPTOGRAPHY : A GUIDED TOUR

Nicolas Méloni✉, François Palma✉, Pascal Véron✉

Laboratoire IMath, Université de Toulon
La Garde, France

Abstract. The Polynomial Modular Number System (PMNS) offers an alternative to the conventional binary multi-precision representation system for large integers. Its effectiveness has been demonstrated for various cryptosystems using prime field arithmetic [2, 4, 6], with prime sizes ranging from 256 to 736 bits. However, as the size of p increases, the relative performance of PMNS compared to standard arithmetic diminishes. Furthermore, the generation process of a PMNS has a worst-case complexity of $\mathcal{O}(2^n)$, where n denotes the number of symbols used to represent an integer modulo p in this representation system. In this paper, we present several alternatives and improvements to the construction and implementation processes of PMNS, which are tailored to the size of p.

1. **Introduction.** Modern cryptography relies heavily on finite field arithmetic and especially on modular arithmetic. The standard way to represent a field element is through a 2^k-ary positional number system. In this system, an element a of $\mathbb{Z}/p\mathbb{Z}$, where p is a prime, is represented as an $(n = \lfloor \log_2(p)/k \rfloor + 1)$-element array $(a_0, \ldots, a_{n-1})$ corresponding to the integer

$$a = \sum_{i=0}^{n-1} a_i 2^{ki}$$

with $0 \leqslant a_i < 2^k$ or $-2^{k-1} \leqslant a_i < 2^{k-1}$. All arithmetic operations are performed modulo p. Modular multiplication is the most investigated operation, as many cryptosystems rely heavily on it. It is typically performed by a standard integer multiplication followed by a modular reduction. Various approaches have been proposed to accelerate modular reduction modulo a prime number. Two main cases are usually considered, depending on whether the prime can be freely chosen or not. Mersenne primes, which are primes of the form $2^t - 1$, are a typical example of the former case. In that case, a modular reduction can be computed easily as a shift-and-add operation. Generalizations of Mersenne primes have been proposed to extend the range of possible candidates, but they are limited to a small number of primes [3]. In the latter case, when the cryptosystem does not allow for enough freedom to choose a generalized Mersenne number, more generic algorithms must be used.

The Polynomial Modular Number System (PMNS) was introduced by Bajard, Imbert, and Plantard as an alternative to the standard radix 2^k multi-precision representation, with the goal of speeding up modular arithmetic when special primes

Key words and phrases. Polynomial Modular Number System, Finite field, Modular arithmetic.

are not available [1]. In this system, a field element is represented as a polynomial of bounded degree, and field arithmetic is performed using polynomial arithmetic. Specifically, an element a of $\mathbb{Z}/p\mathbb{Z}$ is represented as a polynomial $A(X) = \sum_{i=0}^{n-1} a_i X^i \mod E(X)$, where E is a degree n polynomial and $|a_i| < \rho$ for some ρ.

A key advantage of PMNS is that, even if p has no special form, the polynomial E can usually be chosen such that polynomial reduction modulo $E(X)$ is fast (for example, by setting $E(X) = X^n - 1$). In this case, the modular reduction that occurs in the positional number system is replaced by a fast polynomial reduction (known as external reduction) followed by a coefficient reduction (known as internal reduction). However, the internal reduction operation becomes the critical point.

Previous studies have shown that the PMNS is highly competitive with state-of-the-art algorithms for integers ranging from 256 to 736 bits [2, 4, 6]. However, as the size of integers increases beyond 1024 bits, it becomes increasingly difficult to generate good parameters to construct a PMNS for larger integers. On top of that relative performance of PMNS compared to standard arithmetic decreases.

In this work we propose to address these issues with a two-fold approach. First, we generalize the Montgomery-like internal reduction method to allow for the generation of good parameters for larger primes. Second, we address the decrease in PMNS relative efficiency due to two phenomena.

First, as the size of p increases, the relative size of the PMNS data structure (i.e., the degree of the polynomials) increases faster than that of the radix 2^k standard representation leading to additional computations. To address this, we propose a method to improve the relative size of the PMNS data structure.

Second, most libraries switch to subquadratic multiplication algorithms (such as Karatsuba or Toom-Cook) beyond 2000 bits when PMNS uses quadratic matrix vector products. To address this, we show that PMNS can be performed faster by parallelizing the computation and taking advantage, under some conditions, of the Toeplitz matrix form.

The rest of the paper is organized as follows. Section 2 provides the fundamental knowledge required to understand PMNS. In Section 3, we present a generalized Montgomery algorithm to generate PMNS for large primes. In Section 4 we examine the reasons for the decreasing efficiency of PMNS as p grows and propose a practical solution for software implementation on 64- bit architecture, based on 128-bit arithmetic. In Section 5 two methods, based on parallelism and Toeplitz matrices, are proposed to improve the PMNS efficiency, which can be combined. In Section 6, we detail our software implementation and results.

2. Mathematical Background.

In this section, we will provide a brief overview of the necessary mathematical background related to PMNS.

2.1. PMNS.

Definition 2.1. Let $p \geqslant 3$, $n \geqslant 2$, $\gamma \in [1, p-1]$ and $\rho \in [1, p-1]$. Let $E \in \mathbb{Z}[X]$ a monic polynomial of degree n, such that $E(\gamma) \equiv 0 \pmod{p}$. A PMNS is a set $\mathcal{B} \subset \mathbb{Z}[X]$ such that :

1. $\forall A \in \mathcal{B}$, $\deg(A) < n$,
2. $\forall A(X) = \sum_{i=0}^{n-1} a_i X^i \in \mathcal{B}$, $-\rho < a_i < \rho$ for all i,

3. $\forall a \in \mathbb{Z}/p\mathbb{Z}$, $\exists A \in \mathcal{B}$ such that $A(\gamma) \equiv a \pmod{p}$.

With the PMNS, elements of a finite field are represented by polynomials with degree at most $n-1$ and bounded coefficients. Finite field arithmetic operations are then performed on these polynomials subject to additional constraints:

1. Given two field elements a and b, compute a polynomial $A(X)$ representing a and a polynomial $B(X)$ representing b in $\mathcal{B}$.
2. Compute $C(X) = A(X) \odot B(X) \bmod E(X)$, where $\odot$ denotes polynomial multiplication or polynomial addition, $E(X)$ is the external reduction polynomial, and mod denotes the polynomial remainder operation. This step is called *external reduction*.
3. Find a polynomial $\tilde{C}(X)$ such that $\deg C(X) = \deg \tilde{C}(X)$, $\tilde{C}(\gamma) \equiv C(\gamma) \bmod p$ and $|\tilde{c}_i| < \rho$ for all coefficients $\tilde{c}_i$ of $\tilde{C}(X)$. This step is called *internal reduction*.

These additional constraints ensure that the arithmetic operations produce results in the appropriate residue class modulo the prime p and with coefficients bounded by a given threshold ρ.

Step 3 is the most critical point from an efficiency standpoint. This is because step 1 is carried out only once during the entire cryptographic computation and is often precomputed offline. Step 2 involves polynomial arithmetic, which is a fast operation. Therefore, only step 3 requires special attention and is the main bottleneck in terms of performance.

2.2. **Montgomery Algorithm.** Let $\mathbb{Z}_{n-1}[X]$ be the set of polynomials of degree at most $n-1$ in $\mathbb{Z}[X]$. Let us consider the set $\mathfrak{L}$ given by:

$$\mathfrak{L} = \left\{ (x_0, \ldots, x_{n-1}) \in \mathbb{Z}^n : \sum_{i=0}^{n-1} x_i \gamma^i \equiv 0 \mod p \right\}.$$

It is an n-dimensional lattice that can be viewed as a subset of $\mathbb{Z}_{n-1}[X]$ which consists of polynomials that vanish at γ modulo p. In order to solve step 3, we can look for a polynomial $T(X) \in \mathfrak{L}$ such that $|C(X) - T(X)|_\infty < \rho$. We can then set $\tilde{C}(X) = C(X) - T(X)$. Finding such a polynomial is closely related to solving the closest vector problem in a lattice, which is known to be an NP-hard problem [12]. Therefore, the efficiency of constructing a PMNS for a given integer p mainly depends on the efficiency of the algorithm used to solve step 3.

To summarize, in order to generate a PMNS with efficient internal reduction for a given integer p, we need an efficient process to generate all the parameters of the PMNS that guarantee the existence of an efficient internal reduction process. The choice of parameters and the way they are defined depend mainly on the internal reduction algorithm used.

The classical algorithm used to perform step 3 is called the Montgomery-like reduction algorithm and is described in [11] (see Alg. 1). This algorithm depends on a parameter ϕ which for practical reasons is set to 2^k where k is the size of the hardware register being used, typically 2^{64}. The algorithm also mainly depends on the existence of a polynomial $M(X) \in \mathfrak{L}$ invertible modulo (E, ϕ). The existence and the effective construction of this polynomial has been widely studied in [5, 6]. It is proven in [6] that the output of Algorithm 1 is in $\mathcal{B}$ as soon as ρ and ϕ satisfy

$$\rho \geqslant 2\|\mathcal{M}\|_1 \quad \text{and } \phi \geqslant 2w\rho\,,$$

Algorithm 1 Coefficients reduction [11]

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$ a PMNS, $V \in \mathbb{Z}_{n-1}[X]$, $M \in \mathbb{Z}_{n-1}[X]$ such that $M(\gamma) \equiv 0 \pmod{p}$, $\phi \in \mathbb{N} \setminus \{0\}$ and $M' = -M^{-1} \bmod (E, \phi)$.
Ensure: $S(\gamma) = V(\gamma)\phi^{-1} \pmod{p}$, with $S \in \mathbb{Z}_{n-1}[X]$
1: $Q \leftarrow V \times M' \bmod (E,\ \phi)$
2: $T \leftarrow Q \times M \bmod E$
3: $S \leftarrow (V + T)/\phi$
4: return S

where

$$\mathcal{M} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & & \vdots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{1}$$

and w is a constant depending on $E(X)$.

With that in mind, for a given prime p, in the context of a software implementation, the PMNS parameter generation process requires the following steps :

1. Choose $\phi = 2^k$ and $n > \frac{\log_2 p}{k}$.
2. Find an irreducible degree n polynomial $E(X)$ and γ a root of $E(X) \bmod p$.
3. Build the lattice $\mathfrak{L}$.
4. Apply LLL on $\mathfrak{L}$ to obtain a reduced basis $\mathcal{L}$.
5. Find a polynomial $M(X) \in \mathfrak{L}$ with small coefficients and invertible modulo (E, ϕ).
6. Set $\rho = 2\|\mathcal{M}\|_1$.
7. Check that $\phi \geqslant 2w\rho$.

Note that, once the parameters have been generated, they induced a minimum value for the parameter ρ. If it is larger than some expected value, or if step 7 fails, one must go through the whole process all over again from either step 1 or 2. As p grows in size, step 4 can become time-consuming but the LLL algorithm remains a polynomial time algorithm. However, step 5, which involves finding a polynomial $M(X)$ with small coefficients, is the most critical step in terms of efficiency whose worst case scenario complexity is exponential in the degree of $E(X)$ [6, Algorithm 8]. The following section is devoted to addressing that critical issue.

3. Generating PMNS for large primes.

The main challenge in the PMNS parameter generation process is that, in the worst-case scenario, step 5 requires an exhaustive search among a set of 2^n polynomials in the lattice $\mathfrak{L}$. On a 64-bit architecture, n is at least as large as $n_{\text{opt}} = \lfloor (\log_2 p)/64 \rfloor + 1$. For instance, for 256-bit primes p, this exhaustive search involves only 2^5 iterations and is thus feasible. However, for a 4096-bit integer, if we want to construct a PMNS with $n \geqslant 65$, an exhaustive search among a set of at least 2^{65} polynomials is required, which is extremely challenging and computationally expensive.

To overcome this issue, we propose a generalized version of the Montgomery-like algorithm that eliminates the need for any search.

3.1. Montgomery algorithm over lattices.

The Montgomery algorithm for integers can be described as follows [10]. Let $c \in \mathbb{Z}$, $m \in \mathbb{N}$ (m odd), such that $2^{k-1} \leqslant m < 2^k (= \phi)$. To compute a representative of c in $\mathbb{Z}/m\mathbb{Z}$, the algorithm

adds a multiple qm of m to c so that c and $c+qm$ are in the same equivalence class in $\mathbb{Z}/m\mathbb{Z}$. Now, q is chosen such that $c+qm$ is a multiple of ϕ. In other words, one computes q such that $c+qm \equiv 0 \bmod \phi$. From this, one can deduce that $q \equiv -m^{-1}c \bmod \phi$ and the algorithm outputs the value $(c+qm)/\phi$ which is a representative of $c\phi^{-1}$ in $\mathbb{Z}/m\mathbb{Z}$.

C. Negre and T. Plantard generalized this idea in [11] to design an internal reduction process on $\mathbb{Z}[X]$. The operation $c+qm$ becomes $C(X)+Q(X)M(X)$ and this implies that $M(X)$ must satisfy $M(\gamma) \equiv 0 \bmod p$ so that the output result be equal to $C(\gamma)/\phi$ in $\mathbb{Z}/p\mathbb{Z}$. Conditions on ϕ and ρ are given in [11] so that $\|(C(X)+Q(X)M(X) \bmod E(X))/\phi\|_\infty < \rho$.

In the Montgomery multiplication algorithm, adding qm to c means adding a representative of 0 in $\mathbb{Z}/m\mathbb{Z}$. In the Montgomery internal reduction algorithm, adding $Q(X)M(X) \bmod E(X)$ is equivalent to adding a polynomial which vanishes in $\gamma \bmod p$. We have seen in Subsection 2.2 that the set $\mathfrak{L}$ is an n-dimensional lattice. Any polynomial in $\mathfrak{L}$ can be computed using a basis of this lattice. The set of polynomials of the form $Q(X)M(X) \bmod E(X)$ is only a sublattice of this lattice. Thus, we propose to modify the computation of $C(X)+Q(X)M(X)$ by $C(X)+Z(X)$ where $Z(X) \in \mathfrak{L}$.

Let $L_0(X), \ldots, L_{n-1}(X)$ be a reduced basis of $\mathfrak{L}$, and let $\mathcal{L}$ be the matrix where the i^{th} row contains the coefficients of $L_{i-1}(X)$. Let $C=(C_0,\ldots,C_{n-1})$ be the coefficients of $C(X)$. We aim to find a vector $q \in \mathbb{Z}^n$ such that $C+q\mathcal{L} \equiv 0 \bmod \phi$. This equality gives $q = C(-\mathcal{L}^{-1}) \bmod \phi$. From [11, Definition 3], we have that $\det(\mathcal{L})=p$ which is coprime with ϕ for $\phi = 2^k$, hence $-\mathcal{L}^{-1} \bmod \phi$ always exists. Algorithm 2 sums up this new internal reduction. This algorithm does not need to precompute a specific polynomial $M(X)$ that might require an exhaustive search.

Algorithm 2 Coefficients reduction, new version

Require: $\mathcal{B} = (p, n, \gamma, \rho, E)$ a PMNS, $C \in \mathbb{Z}_{n-1}[X]$, $\mathcal{L}$ a reduced basis of $\mathfrak{L}$, $\phi \in \mathbb{N} \setminus \{0\}$ and $-\mathcal{L}^{-1} \bmod \phi$.
Ensure: $S(\gamma) \equiv C(\gamma)\phi^{-1} \bmod p$, with $S \in \mathbb{Z}_{n-1}[X]$
1: $q \leftarrow C(-\mathcal{L}^{-1}) \bmod \phi$
2: $S \leftarrow (C+q\mathcal{L})/\phi$
3: return S

Proposition 3.1. *Let $C(X)$ be a polynomial of degree at most $n-1$, if*

$$\|C\|_\infty \leqslant \phi(\rho - \|\mathcal{L}\|_1)$$

then the output S of Algorithm 2 (with C as input) is such that $\|S\|_\infty < \rho$ (i.e., $S \in \mathcal{B}$).

Proof. Step 1 of algorithm 2 ensures that $\|q\|_\infty < \phi$. Now,

$$\begin{aligned} \|q\mathfrak{L}\|_\infty &= \max_i |\textstyle\sum_j q_j \mathcal{L}_{ji}| \\ &< \phi \max_i \textstyle\sum_j |\mathcal{L}_{ji}| \end{aligned}$$

Hence

$$\|q\mathfrak{L}\|_\infty < \phi \|\mathcal{L}\|_1 .$$

Now,

$$\|C + q\mathfrak{L}\|_\infty \leqslant \|C\|_\infty + \|q\mathfrak{L}\|_\infty < \|C\|_\infty + \phi\|\mathfrak{L}\|_1 \,.$$

Thus, as soon as

$$\|C\|_\infty + \phi\|\mathfrak{L}\|_1 \ < \rho\phi \,,$$

then

$$\|S\|_\infty < \rho \,.$$

□

Now, we give sufficient conditions on ϕ and ρ so that the polynomial $C(X)$ meets the requirements of proposition 3.1.

Proposition 3.2. *Let $A(X)$ and $B(X)$ be two elements of $\mathcal{B}$ and let $C(X) = A(X)B(X) \bmod E(X)$. Let $w > 0$ such that $\|C(X)\|_\infty \leqslant w\|A(X)\|_\infty\|B(X)\|_\infty$, if $\phi \geqslant 2w\rho$ and $\rho \geqslant 2\|\mathcal{L}\|_1$ then $\|C\|_\infty \leqslant \phi(\rho - \|\mathcal{L}\|_1)$.*

Proof. Since $A(X)$ and $B(X)$ belong to $\mathcal{B}$, then

$$\|C\|_\infty \leqslant w\rho^2 \,,$$

hence if

$$w\rho^2 < \phi\rho - \phi\|\mathcal{L}\|_1 \,,$$

then $C(X)$ will satisfy the constraint on its infinity norm. Now,

$$\begin{aligned} w\rho^2 < \phi\rho - \phi\|\mathcal{L}\|_1 \quad &\Leftrightarrow \quad \phi\|\mathcal{L}\|_1 + w\rho^2 < \phi\rho \\ &\Leftrightarrow \quad \tfrac{w\rho^2}{\phi} + \|\mathcal{L}\|_1 < \rho \,. \end{aligned}$$

Hence, if

$$\frac{w\rho^2}{\phi} \leqslant \frac{\rho}{2} \quad \text{and} \quad \|\mathcal{L}\|_1 \leqslant \frac{\rho}{2} \,,$$

we obtain the required result. □

Remark 1. A tight bound on the value w is given in [6, subsection 4.3]. It only depends on the shape of the polynomial $E(X)$.

Remark 2. For a software implementation, ϕ is often chosen equal to 2^{64}. Hence to satisfy the bound $\phi \geqslant 2w\rho$, it is better to choose w and ρ "small". The value w depends on the coefficients of $E(X)$, which explains why sparse polynomials with small coefficients are used to build a PMNS.

With our generalized Montgomery algorithm, the PMNS parameter generation process requires the following steps:

1. Let $n \geqslant n_{\text{opt}} (= \lfloor (\log_2 p)/64 \rfloor + 1)$.
2. Find an irreducible degree n polynomial $E(X)$ and γ a root of $E(X) \bmod p$.
3. Build the lattice $\mathfrak{L}$.
4. Apply LLL on $\mathfrak{L}$ to obtain a reduced basis $\mathcal{L}$.
5. Set $\rho = 2\|\mathcal{L}\|_1$ and $\phi = 2w\rho$ (proposition 3.2).

Let us notice once again that we do not need to perform the exhaustive search of a polynomial $M(X)$, which ensures that a PMNS can always be generated, whatever the prime p and the selected polynomial $E(X)$. Practical implementations usually set additional constraints on ρ and ϕ, typically $\rho < 2^{64}$ and $\phi = 2^{64}$. If those are not met, repeat the generation process either from step 1 or 2 with a different n or polynomial E.

Alg./size of p	256	512	1024
Montgomery	121	432	1709
This work	122	443	1728

TABLE 1. Number of cycles to compute a PMNS modular multiplication, gcc 12.1.0, i9-11900KF.

Remark 3. As p grows in size, the PMNS becomes less efficient compared to more standard approaches. Indeed, most software libraries switch to subquadratic multiplication algorithms when p is too large and, to generate good PMNS parameters, it is often necessary to chose n further away from the optimal value $n_{\text{opt}} = \lfloor (\log_2 p)/k \rfloor + 1$ (where k is the target architecture register size) inducing additional computational costs compared to standard representations. Throughout the remainder of this paper, we explore these challenges and present two approaches to address them.

Table 1 shows that, in practice, the performance of our internal reduction process is similar to that of the Montgomery-like reduction. The test protocol is detailed in Section 6.

4. PMNS 128-bit.

In the previous section, we have demonstrated that it is feasible to generate PMNS parameters without requiring an exhaustive search of exponential complexity. However, in this section, we show that even though the PMNS parameter generation process can be performed efficiently, the data structure of PMNS still grows at a faster rate than the standard binary representation. This results in an automatic loss in competitiveness of the PMNS arithmetic. We propose a new implementation of the PMNS algorithm that addresses the issue of the relative size of the PMNS data structure. This new implementation is based on 128-bit arithmetic and enables us to circumvent this issue.

4.1. Data structure size problem.

For a software implementation on a 64-bit architecture, the parameter ϕ is set to 2^{64}. At the end of the parameter generation process, if $2w\rho \geqslant 2^{64}$ then we increment n by 1 and repeat. This explains why, in practice, n moves away from n_{opt}, degrading performances. Hence, for any p, it is of importance that n be not "too far" from n_{opt}. Unfortunately, this condition is hard to meet. Indeed, from proposition 3.2, the parameters ρ and ϕ must satisfy the following bounds to guarantee the existence of the PMNS and the consistency of the internal reduction process :

$$\phi \geqslant 2w\rho \quad \text{and} \quad \rho \geqslant 2\|\mathcal{L}\|_1 \,. \tag{2}$$

A classical result in lattice theory states that the infinity-norm of a "short" vector in $\mathfrak{L}$ is about $p^{1/n}$. Hence, among the lines of the matrix $\mathcal{L}$, which is a reduced basis of $\mathfrak{L}$, there is a line L_i whose infinity norm is greater than $p^{1/n}$. Hence,

$$p^{1/n} \leqslant \|L_i\|_\infty \leqslant \sum_j |L_{ji}| \leqslant \|\mathcal{L}\|_1 \,.$$

From equation 2, we deduce that n must be chosen such that

$$p^{1/n} \leqslant \frac{\phi}{4w} \,.$$

In practice, as ϕ is equal to 2^{64}, the more p grows , the more it will be difficult to find an integer n near from n_{opt} which satisfies this inequality.

Proposition 4.1. *For any $\varepsilon > 0$, there exists an integer p such that*

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > \frac{\phi}{4w}\,.$$

Proof. Notice that since $w > 1$, $\frac{\phi}{4w} < 2^{62}$. Now,

$$n_{\text{opt}} + \varepsilon < \frac{\log_2 p}{64} + \varepsilon + 1 \Leftrightarrow \frac{\log_2 p}{n_{\text{opt}} + \varepsilon} > \frac{64 \log_2 p}{\log_2 p + 64\varepsilon + 64}\,.$$

Hence

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > (2^{64})^{\frac{\log_2 p}{\log_2 p + 64\varepsilon + 64}}\,.$$

As

$$\lim_{p\to\infty} \frac{\log_2 p}{\log_2 p + 64\varepsilon + 64} = 1\,,$$

there still exists an integer p such that

$$p^{\frac{1}{n_{\text{opt}}+\varepsilon}} > 2^{62}\,.$$

□

This proves that as p grows, we must choose a n far away from n_{opt}.
As an example, let us suppose that we aim to build a PMNS for a 2048-bit integer, then $n_{\text{opt}} = 33$. On one hand, we have

$$p^{1/n_{\text{opt}}} \simeq 2^{62.06}\,,$$

and on the other hand, since $w \geqslant n$ (from the definition given in [6, subection 4.3]),

$$\frac{2^{64}}{4w} \simeq 2^{56.96}\,.$$

To satisfy the required constraint, we have to choose $n \geqslant 37$. The global performances of the PMNS suffer in consequence.

4.2. **128-bit PMNS vs 64-bit.** We have shown that the parameters must satisfy $p^{1/n} \leqslant \frac{\phi}{4w}$. Practically speaking, for a given p, there are thus two way to meet this condition: either increase n or ϕ. The usual approach is to set $\phi = 2^{64}$ so that one must increase n. The reason why ϕ is set that way is that the reduction algorithm involves divisions by ϕ that can be costly for a random value and virtually free when $\phi = 2^k$ where k matches the register size of the target device. Hence, rather than increasing n, an alternative is thus to choose $\phi = 2^{128}$ instead of 2^{64}. Doing so still guarantees fast divisions and allows to set much lower value for n. As a consequence, the software implementation of the PMNS has to be upgraded in order to deal with 128 and 256-bit integer operations on polynomial coefficients.

Let n_{64} (resp. n_{128}), the actual value of n for $\phi = 2^{64}$ (resp. for $\phi = 2^{128}$), Table 2 shows how n moves away from n_{opt} depending on the size of the prime p. When $\phi = 2^{64}$, for integers whose size exceeds 1024 bits, the value of n_{64} will certainly give rise to a PMNS multiplication process whose complexity will be higher than the classical multiplication algorithms when applied to the same operands split into n_{opt} 64-bit blocks (as it is done in various multi-precision libraries). For $\phi = 2^{128}$, the gap between n_{128} and n_{opt} remains negligible and justifies the use of this value for ϕ. From a practical point of view, we can observe that $n_{128} <$

size of p	256	512	1024	2048	4096	8192
n_{opt64}	5	9	17	33	65	129
n_{64}	5	9	19	40	83	187
n_{opt128}	3	5	9	17	33	65
n_{128}	3	5	9	18	36	72

TABLE 2. Optimal polynomial degrees vs practical ones used in implementation per size of prime considered.

size of p	256	512	1024	2048	4096	8192
Red-64	122	443	1709	7690	33471	169070
Red-128	245	720	2367	10920	39446	**156310**

TABLE 3. Comparative table of performances for Red-64 and Red-128 in number of processor cycles for one modular multiplication on Intel processor i9-11900KF with gcc 12.1.0.

$2n_{64}$. Hence performances should be better using $\phi = 2^{128}$, but this has to be tempered considering the cost of a software implementation of 128-bit arithmetic. The reference implementation with $\phi = 2^{64}$ makes use of internal compiler `__int128` registers which are 128-bit integer words handled directly by the compiler. This allows the algorithm to completely forgo any multi-precision considerations such as dealing with carries and leave it all to the compiler to be done internally in assembly. No such equivalent 256-bit register exists at the current time so, for $\phi = 2^{128}$, multi-precision algorithms have to be used which lead to slower calculations mainly due to carry propagation. Each 128-bit coefficient will be stored on two 64-bit registers and arithmetic on this coefficient has to consider this representation.

Both Karatsuba and the Schoolbook version of the multiplication of two elements were evaluated and the Schoolbook version has been found to be slightly faster so far. This means that, for $\phi = 2^{128}$, any coefficient multiplication involves 4 distinct multiplication operations on 64-bit integers. Our internal reduction process involves two vector-matrix products. For $\phi = 2^{64}$, each operation involves n_{64}^2 multiplications between 64-bit variables, whereas for $\phi = 2^{128}$ it involves $4n_{128}^2$ between 64-bit variables. For 1024-bit primes and beyond, since $n_{64} > 2n_{128}$, it means that for a given prime p, the 64-bit version of PMNS should be slower than its 128-bit counterpart. Unfortunately, the 128-bit version has additional carry operations dragging down the complexity function and giving us worse results until higher values of p widen the gap substantially between n_{64} and n_{128}. Table 3 shows that for primes larger than 8000 bits, reducing the size of the data structure leads to faster implementation as long as quadratic algorithms are used. For the sequel of the paper, we will note :

- **Red-64** : the 64-bit version of our internal reduction algorithm,
- **Red-128** : the 128-bit version of our internal reduction algorithm.

5. **Faster PMNS for large primes.** As the size of the prime p exceeds 2000 bits, modular multiplication implementations often switch from classical schoolbook algorithms to subquadratic alternatives like Karatsuba or Toom-Cook. A first approach

to lower our execution time is to optimize the computation of $A(X)B(X)$ mod $E(X)$ using Toeplitz matrices.

5.1. **Toeplitz matrix version.** When $E(X) = X^n - \lambda$, we can optimize the computation of $A(X)B(X)$ mod $E(X)$ using Toeplitz matrices. Let $A(X)$ and $B(X)$ be two elements of a PMNS $\mathfrak{B}$, to compute a representative of $A(X)B(X)$ in $\mathfrak{B}$, we first have to compute $A(X)B(X)$ mod $E(X)$. We could use Karatsuba algorithm to compute the product of two polynomials and then reduce the result. It appears that this operation can be more efficient if seen as a vector matrix product involving Toeplitz matrices.

5.2. **Toeplitz vs Karatsuba.**

Definition 5.1. An $n \times n$ matrix is a matrix defined by $2n-1$ values $a_0, a_1, \ldots, a_{2n-2}$ such that

$$\begin{pmatrix} a_0 & a_1 & a_2 & \cdots & \cdots & a_{n-1} \\ a_n & a_0 & a_1 & \ddots & & \vdots \\ a_{n+1} & a_n & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_1 & a_2 \\ \vdots & & \ddots & a_n & a_0 & a_1 \\ a_{2n-2} & \cdots & \cdots & a_{n+1} & a_n & a_0 \end{pmatrix}$$

There is a natural link between Toeplitz matrices and modular polynomial multiplication when the reduction polynomial $E(X)$ is equal to $X^n - \lambda$. As an example let us consider $A(X) = a_0 + a_1X + a_2X^2 + a_3X^3$ and $B(X) = b_0 + b_1X + b_2X^2 + b_3X^3$, and let $C(X) = A(X)B(X)$ mod $X^4 - \lambda$. A straight forward computation gives

$$\begin{aligned} C(X) \quad = \quad & a_3b_0X^3 + a_2b_1X^3 + a_1b_2X^3 + a_0b_3X^3 + \\ & \lambda a_3b_3X^2 + a_2b_0X^2 + a_1b_1X^2 + a_0b_2X^2 + \\ & \lambda a_3b_2X + \lambda a_2b_3X + a_1b_0X + a_0b_1X + \\ & \lambda a_3b_1 + \lambda a_2b_2 + \lambda a_1b_3 + a_0b_0\,. \end{aligned}$$

This last result can be computed as

$$\begin{pmatrix} b_0 & b_1 & b_2 & b_3 \end{pmatrix} \times \begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ \lambda a_3 & a_0 & a_1 & a_2 \\ \lambda a_2 & \lambda a_3 & a_0 & a_1 \\ \lambda a_1 & \lambda a_2 & \lambda a_3 & a_0 \end{pmatrix}$$

The matrix used in this formula is a Toeplitz matrix defined by the values a_0, a_1, a_2, a_3, λa_3, λa_2 and λa_1. Now, a classical result states that the vector matrix product in the Toeplitz context can be efficiently computed using the following proposition [7].

Proposition 5.1. *Let*

$$A = \begin{pmatrix} A_0 & A_1 \\ A_2 & A_0 \end{pmatrix}$$

an $n \times n$ Toeplitz matrix where each submatrix is a $n/2$ square matrix. Let $B = \begin{pmatrix} B_0 & B_1 \end{pmatrix}$ a vector of size n where each B_i is of size $n/2$. The product $B \times A$ can be computed as $(P_0 - P_2, P_0 - P_1)$, where

$$\begin{aligned} P_0 \quad &= \quad (B_0 + B_1)A_0 \\ P_1 \quad &= \quad B_1(A_0 - A_2) \\ P_2 \quad &= \quad B_0(A_0 - A_1)\,. \end{aligned}$$

In [7] the authors show that this approach can be used recursively to obtain a complexity of $\mathcal{O}(n^{\log_2 3})$ similar to that of Karatsuba. Although the two complexities are asymptotically the same, in the next subsection we show that the Toeplitz approach is less costly in practice.

Proposition 5.2. *Let $A(X)$ and $B(X)$ be two polynomials of degree at most $n-1$ and let $E(X) = X^n - \lambda$. Let $T(n)$ be the execution time needed to compute $A(X)B(X) \bmod E(X)$, then, on a 64-bit architecture :*

- $T(n) = 3T(n/2) + nA_{64} + 4nA_{128} - 4A_{128}$, *using Karatsuba method,*
- $T(n) = 3T(n/2) + (\frac{5n}{2} - 2)A_{64} + nA_{128}$, *using Toeplitz method.*

where A_{128} (resp. A_{64}) is the execution time of a 128-bit (resp. 64-bit) addition.

Proof. Let $A(X) = A_H(X)X^{n/2} + A_L(X)$ and $B(X) = B_H(X)X^{n/2} + B_L(X)$ where the degree of $A_H(X), A_L(X), B_H(X)$ and $B_L(X)$ is at most $\frac{n}{2} - 1$, then

$$A(X)B(X) = A_H(X)B_H(X)X^n + N(X)X^{n/2} + A_L(X)B_L(X)$$

with

$$N(X) = (A_H(X) + A_L(X))(B_H(X) + B_L(X)) - A_H(X)B_H(X) - A_L(X)B_L(X)\,.$$

To reduce modulo $E(X)$ the product $A(X)B(X)$, let

- $A_HB_H(X) = A_HB_H^{(H)}(X)X^{n/2} + A_HB_H^{(L)}(X)$, with $\deg A_HB_H^{(H)}(X) \leqslant \frac{n}{2} - 2$ and $\deg A_HB_H^{(L)}(X) \leqslant \frac{n}{2} - 1$,
- $A_LB_L(X) = A_LB_L^{(H)}(X)X^{n/2} + A_LB_L^{(L)}(X)$, with $\deg A_LB_L^{(H)}(X) \leqslant \frac{n}{2} - 2$ and $\deg A_LB_L^{(L)}(X) \leqslant \frac{n}{2} - 1$,
- $N(X) = N_H(X)X^{n/2} + N_L(X)$, with $\deg N_H(X) \leqslant \frac{n}{2} - 2$ and $\deg N_L(X) \leqslant \frac{n}{2} - 1$.

We thus obtain

$$\begin{aligned} A(X)B(X) \bmod E(X) \quad = \quad & A_LB_L^{(L)}(X) + \lambda N_H(X) + \lambda A_HB_H^{(L)}(X) + \\ & (N_L(X) + A_HB_H^{(L)}(X) + \lambda A_HB_H^{(H)}(X))X^{n/2}\,. \end{aligned}$$

For the sequel of this analysis, let us consider the case where $A(X)$ and $B(X)$ are two elements of a PMNS, meaning that their coefficients are strictly less than ρ with $\rho < 2^{63}$ on a 64-bit architecture (see equation 2). Then one step of the Karatsuba algorithm needs to compute :

- $A_H(X) + A_L(X)$ and $B_H(X) + B_L(X)$, which involves at most $\frac{n}{2}$ additions between 64-bit words,
- a recursive call to compute the three products in $N(X)$,
- $2(n-1)$ additions on 128-bit words to compute $N(X)$,
- $\frac{n}{2} - 1 + \frac{n}{2}$ additions on 128-bit words to compute $A_LB_L^{(L)}(X) + \lambda N_H(X) + \lambda A_HB_H^{(L)}(X)$,
- $\frac{n}{2} + \frac{n}{2} - 1$ additions on 128-bit words to compute $N_L(X) + A_HB_H^{(L)}(X) + \lambda A_HB_H^{(H)}(X)$.

which gives a total cost of n 64-bit additions, $4n-4$ 128-bit additions and 3 recursive calls.

Now, concerning the Toeplitz approach, one step needs to compute :

- $\frac{n}{2}$ additions on 64-bit words for $B_0 + B_1$,
- $2\frac{n}{2} - 1$ additions on 64-bit words for $A_0 - A_2$ (resp. $A_0 - A_1$) since each one is a $\frac{n}{2} \times \frac{n}{2}$ Toeplitz matrix depending only on $2\frac{n}{2} - 1$ coefficients,

Size	1024	2048	4096	8192
GnuMP				
n	16	32	64	128
Low level	1715	5501	16897	53042
Classical Mont.	1534	4653	14522	44587
Mont. CIOS	1187	4661	16793	61478
This work				
n	19	40	84	187
Red-64	1709	7690	33471	169070
n	-	-	-	72
Red-128	-	-	-	156310

TABLE 4. Number of cycles to compute a modular multiplication using GnuMP and PMNS on Intel processor i9-11900KF with gcc 12.1.0.

- a recursive call to compute the three vectors P_0, P_1 and P_2,
- $2\frac{n}{2}$ additions on 128-bit words to compute $P_0 - P_2$ and $P_0 - P_1$,

which gives a total cost of $\frac{n}{2} + 2n - 2$ 64-bit additions, n 128-bit additions and 3 recursive calls.
Now we have to evaluate the complexity of the last step of each method. For Karatsuba , the last step is a product of two 64-bit integers which has to be done using the `__int128` `gcc` type.
The same observation can be done for Toeplitz method. Hence, the difference between the two methods relies on the splitting and reconstruction process. The Toeplitz method has a lower complexity for these two steps. □

Remark 4. There exists other ways to split the initial matrix [8]. For example one can use a 3-way split or 5-way split depending on the initial size of the matrix, and one can mix them during the recursive process. These multi-way splittings all lead to subquadratic algorithms.

Even if, using Toeplitz matrices, computing $A(X)B(X) \bmod E(X)$ becomes subquadratic, the two other steps of our internal reduction algorithm (Algorithm 2) remain quadratic. We thus propose to take into account the parallel nature of PMNS to optimize these last two steps.

5.3. **Parallel PMNS (PPMNS).** Proposition 3.2 guarantees that for any prime p, we can build a PMNS with a suitable internal reduction process. Unfortunately, for prime p whose size exceeds 1024, PMNS performances are not competitive with GnuMP library if no parallelism is used. For 1024-bit primes our performances are similar to low level GnuMP functions, but for higher sizes, our performances are worse (see Table 4, where n stands for the number of 64-bits block – or 128-bit blocks for Red-128 – used to represent an integer). It is all the more obvious when taking into account the two undocumented implementations of the classical Montgomery multiplication on integers (respectively named Classic Montgomery and Montgomery CIOS [9] in Table 4). The PMNS multiplication process we propose, involves three vector-matrix products whose complexity is quadratic in n, while GnuMP (or OpenSSL) switches to algorithms whose complexity is subquadratic in

n_{opt}, when n_{opt} is larger than some threshold computed at compile time. More precisely, to compute $ab \mod p$, first GnuMP low level functions computes ab using subquadratic algorithm depending on the size of a and b, then the result is reduced modulo p using a division algorithm depending on the size of the operands. The more n moves away from n_{opt}, the worse the PMNS performances get, compared to a classical multi-precision library. Now, as the operations in PMNS can be parallelized, one can hope that on a multi-core processor, the PMNS will give better performances despite the high difference between n and n_{opt}. Let us denote by $M(n)$ the computational cost of ab, $R(n)$ that of the modular reduction and $VM(n)$ that of the vector-matrix product. On a δ-core processor, if $VM(n)/\delta < M(n_{\text{opt}}) + R(n_{\text{opt}})$, then the PPMNS will perform better than any classical multi-precision library. In section 6, we provide a proof of concept targeting 8192-bit integers.

6. **Implementation and results.** All the results given in this paper have been computed using the following procedure :

- the *Turbo-Boost*® is deactivated during the tests;
- 1000 runs are executed in order to "heat" the cache memory, i.e. we ensure that the cache memory (data and instruction) is in an enough stabilized state in order to avoid untimely cache faults;
- one generates 500 random data sets, and for each data set the execution clock cycle numbers over a batch of 1000 runs is recorded;
- the performance is the median value.

All the source code is available on Github[1].

6.1. **A proof of concept for 8192-bit integers.** From the preceding results, it appears that a suitable implementation of a PMNS depends on the size of the parameter p. For integers used in elliptic curve cryptography (from 256 bits to 1024 bits), the internal reduction we proposed in section 3.1 guarantees that for any prime p, one can always build a 64-bit version of a PMNS with an efficient reduction process which outperforms low level GnuMP functions and is competitive with the undocumented Montgomery multiplication functions (until 512 bits). Moreover any polynomial $E(X)$ can be used to build the PMNS which enlarges the set of PMNS linked to a prime p.
For larger sizes, our internal reduction process is not competitive with subquadratic algorithms used by GnuMP, hence we have to consider the parallel version of PMNS (PPMNS). A major advantage is that, despite the increasing value of n, we can keep $\phi = 2^{64}$ and the corresponding source code only relies on processor 64-bit arithmetic instead of a costly software 128-bit arithmetic. For any polynomial $E(X)$, the multiplication process in the PMNS will involve three vector-matrix product which are well suited for parallel implementation. As a proof of concept, we consider in this section 8192-bit integers. Depending on the size of the operands a and b , the GnuMP library switches to different algorithms in order to compute ab. The thresholds to select the multiplication algorithm is chosen when compiling the source code of the library. On a Skylake processor, for GnuMP 6.2.1, the thresholds for sizes 1664 up to 13248 are (from `mpn/x86_64/skylake/gmp-mparam.h`):

```
#define MUL_TOOM22_THRESHOLD                26
#define MUL_TOOM33_THRESHOLD                73
#define MUL_TOOM44_THRESHOLD               208
```

[1] https://github.com/francoispalma/PMNS/

```
...
#define DC_DIV_QR_THRESHOLD                    55
```

This means that up to $25 \times 64 = 1600$-bit integers, GnuMP uses the classical schoolbook algorithm to compute the product ab. For size in the range 1664 to 4608, GnuMP uses Toom22 which is similar to Karatsuba algorithm. Finally it switches to the classical 3-way split Toom-Cook algorithm up to 13248-bit integers. For 8192-bit integers, we can obtain a rough estimation of the number of multiplication done by GnuMP. Such integers are split into 128 64-bit blocks. Then the product of two such sequences is done using 3-way split Toom-Cook and involves 5 multiplications between 43 64-bit blocks. Each multiplication is done using a Karatsuba-like method which in turn requests 3 multiplications on 22 64-bit blocks, each one being computed using the classical schoolbook algorithm. Hence, the total process involves $22^2 \times 3 \times 5$ multiplications on 64-bit integers, leading to 7260 multiplications. Notice that this is a very optimistic estimation as we do not take into account all the pre and post processing of the Toom-Cook and Karatsuba algorithms. Next to reduce modulo p, a divide and conquer algorithm is used for integers whose size is greater than $55 \times 64 = 3520$ bits[2]. As mentioned in GnuMP documentation, for 8192-bit integers, the complexity is about $2.63M(8192)$ where $M(8192)$ is the cost of the product of two 8192-bit integers. Such a product will be computed using a 3-way split Toom-Cook step followed by a Karatsuba step which will boil down to a total of 15 schoolbook multiplications on 1350-bit integers, that is to say about 19000 64-bit multiplications. To sum up to compute $ab \bmod p$ at least 26260 64-bit multiplications are performed.

Now, for 8192-bit integers, one can build a PMNS with $n = 187$. For such a value, if $E(X)$ has no particular shape, the multiplication process will exactly involve $3 \times 187 \times 187 = 104907$ 64-bit multiplications. On a δ-core processor, the parallelized version of the PMNS will perform better if $104907/\delta \leqslant 26260$, which gives $\delta \geqslant 4$. In practice, we observe that on a 5-core processor the PPMNS gives better performances than low-level GnuMP functions and , on a 8-core processor, we beat the undocumented Montgomery modular multiplication functions (see Table 5).

5-core	6-core	8-core	Low level	Classical Mont.	Mont. CIOS
n=187			n=128		
51979	**46224**	**40778**	53042	44567	61478

TABLE 5. Number of cycles to compute a modular multiplication between two 8192-bit integers using GnuMP and PPMNS (for any $E(X)$) on Intel processor i9-11900KF with gcc 12.1.0.

When $E(X) = X^n - \lambda$, the first vector-matrix product can be done using a Toeplitz decomposition if n is a multiple of 2 or 3. Even if $n = 187$ is the smallest value that can be used to build a PMNS for a 8192-bit integer, we consider here $n = 189 = 3 \times 63$. This way, we can use the 3-way split decomposition which involves 6 vector-matrix products between vectors of size 63 and 63×63 matrices and perform them using a classical vector-matrix product since experimental results show that there is no point in splitting further. The next two steps are done using classical vector-matrix product. Hence, on a 6-core processor, the total number of 64-bit multiplications done by one core will be $63^2 + \frac{2 \times 189^2}{6} = 15876$ which is less

[2] https://gmplib.org/manual/Divide-and-Conquer-Division

than the number of multiplications done by GnuMP. Table 6 gives the results we obtained for the polynomial $E(X) = X^{189} - 2$.

6-core	Low level	Classical Mont.	Mont. CIOS
n=189	n=128		
42510	53042	44567	61478

TABLE 6. Number of cycles to compute a modular multiplication between two 8192-bit integers using GnuMP and PPMNS (for $E(X) = X^{189} - 2$) on an Intel processor i9-11900KF with gcc 12.1.0.

The results we obtained show that the parallelized version of PMNS is the best way to manage large integers. When the polynomial $E(X)$ is arbitrary, the vector-matrix product is well suited for parallelization since the use of δ-core leads to approximately a gain factor of δ. When $E(X) = X^n - \lambda$ and $n = 0 \bmod 3$, one can take advantage of the 3-way split Toeplitz decomposition on a 6-core processor.
For integer ranging from 2048-bit up to 8192-bit, our experimental results show that the PPMNS does not perform faster than GnuMP functions (see Table 7). For 2048-bit and 4096-bit integers we ran several benchmarks using from 1 core to 8 cores. The best performances are summarized in Table 7. For 2048-bit integers, the size of the matrices are too small to benefit from multi-threading. For 4096-bit integers, the performances of our 64-bit version using 6 cores are close to those of low level GnuMP functions. In any case, it appears that within this size range, PMNS are not well suited.

6.2. **Toeplitz and internal Montgomery reduction.** Even if there is no guarantee that a suitable polynomial $M(X)$ can be computed in reasonable time when $E(X) = X^n - \lambda$ with λ odd, there is a huge advantage to consider again the Montgomery-like internal reduction. Indeed, steps 1 and 2 of Algorithm 1 involve polynomial modular reductions. As mentioned in the previous section, such an operation leads to a vector-matrix product with a Toeplitz matrix. Hence, all the internal reduction process can be performed using Toeplitz matrices. There are two alternatives to perform the vector-matrix product :

- either using the recursive splitting detailed in Subsection 5.2,

Method/Size	2048	4096
n	32	64
Low level	5501	16897
Classical Mont.	4653	14524
Mont. CIOS	4607	16793
n	40	84
Red-64	7491 (1-core) 10165 (2-core)	36195 (1-core) 17756 (6-core)
Red-128	10920 (1-core) 15314 (2-core)	39446 (1-core) 25797 (4-core)

TABLE 7. Number of cycles to compute a modular multiplication for 2048-bit and 4096-bit integers using GnuMP, PMNS and PPMNS on an Intel processor i9-11900KF with gcc 12.1.0.

Size	2048	4096	8192
8 cores			
Toeplitz Montgomery-like	14421	18193	37026
This work	14529	22782	40778

TABLE 8. Number of cycles to compute a modular multiplication for 2048,4096 and 8192-bit integers using Toeplitz form vs random matrix form on Intel processor i9-11900KF with gcc 12.1.0.

Size	1024	1664	2048
Low level	1715	4039	5501
Classical Mont.	1539	3466	4653
Montgomery CIOS	1187	3153	4607
Toeplitz (this work)	1449	3454	5489

TABLE 9. Number of cycles to compute a modular multiplication for 1024, 1664 and 2048-bit integers using Toeplitz form vs GnuMP on Intel processor i9-11900KF with gcc 12.1.0.

- or using a classical vector-matrix product algorithm taking into account that a Toeplitz matrix is defined by $2n-1$ coefficients instead of n^2 and is thus stored in a 1-dimensional array.

Our experimental results show that the parallel version of the Montgomery-like reduction algorithm using the Toeplitz recursive splitting performs worse than the parallel version of the Montgomery-like reduction algorithm using the classical vector-matrix product. For 8192-bit integers, if the polynomial $M(X)$ exists and can be computed in reasonable time, we obtain better performances than our internal reduction process (see Table 8).

For integers ranging from 1024 to 2048 bits, we have shown that there is no point in using the PPMNS. Within this range, the Toeplitz recursive splitting on a single core outperforms some GnuMP functions as summarized in Table 9.

Concerning the polynomial $M(X)$, if $E(X) = X^n - \lambda$ with λ odd, we exhibit in the next proposition, particular cases where it can be computed in linear time.

Proposition 6.1. *Let n an odd integer such that $1+X+\cdots+X^{n-1}$ is irreducible in $\mathbb{F}_2[X]$, then one can compute in linear time a polynomial $M(X)$ invertible modulo (E, ϕ).*

Proof. From [6, corollary 2 and proposition 5], the polynomial $M(X)$ exists iff the determinant of the matrix $\mathcal{M}$ is odd, where

$$\mathcal{M} = \begin{pmatrix} m_0 & m_1 & \dots & m_{n-1} \\ \dots & \dots & \dots & \dots \\ \vdots & \vdots & & \vdots \\ \dots & \dots & \dots & \dots \end{pmatrix} \begin{matrix} \leftarrow M \\ \leftarrow X.M \bmod E \\ \\ \leftarrow X^{n-1}.M \bmod E \end{matrix} \tag{3}$$

Equivalently, this means that this matrix is invertible over $\mathbb{F}_2$. Now, as $E(X) = X^n+1$ over $\mathbb{F}_2$, the matrix $\mathcal{M}$ is a circulant one and there is an isomorphism between

the set of circulant matrices and the algebra of polynomials modulo $X^n + 1$. Hence $\mathcal{M}$ is invertible iff $M(X)$ is coprime with $X^n + 1$ in $\mathbb{F}_2[X]$.

Let L the matrix whose rows are a reduced basis of the set of polynomials that vanishes in $\gamma \bmod p$. We know that $\det(L) = p$, so L is invertible in $\mathbb{F}_2$. Hence, there exists a line in L which has an odd number of odd coefficients. Otherwise, in $\mathbb{F}_2$, all the lines of the matrix will contain an even number of ones, which implies that the all one vector will be in the kernel of the linear application defined by L, meaning that L is not invertible.

Let n such that $O(X) = 1 + X + \cdots + X^{n-1}$ is irreducible over $\mathbb{F}_2$ (which implies n odd), then $x^n + 1$ is the product of two irreducible polynomials, $X + 1$ and $O(X)$. Let $M(X)$ the polynomial which contains an odd number of odd coefficients, if $M(X)$ is coprime with $O(X)$ in $\mathbb{F}_2[X]$, then $M(X)$ is coprime with $E(X)$ since it does not vanish in 1. Otherwise, it means that $M(X) = O(X)$. In this case, there are only two alternatives:

- either there exists in the matrix L another line with an odd number of odd coefficients, then the corresponding polynomial $\bar{M}(X)$ cannot be equal to $O(X)$, and thus $\bar{M}(X)$ is the polynomial we are searching for,
- or all the other lines of L have an even number δ of odd coefficients. Notice that since $\det(L) = 1 \bmod 2$, then $\delta > 0$, otherwise L will contain a line with only even coefficients which would give a null vector in $\mathbb{F}_2$. Let L_i any line with δ odd coefficients, add the corresponding polynomial to $O(X)$ to obtain a polynomial $M(X)$ with an odd number of odd coefficients coprime with $O(X)$.

□

If we consider that until $n = 40$ an exhaustive search is reasonable to find $M(X)$, here are some $n > 40$ which meets our requirements : 53, 59, 61, 67, 83, 101, 107, 131, 139, 149, 163, 173, 179, 181, 197, 211, 227, 269, 293.

7. **A roadmap for using PMNS.** In this work, we have developed several new ways to use PMNS:

- the original Montgomery-like internal reduction adapted to the Toeplitz recursive splitting (the drawback being that it relies on finding a suitable polynomial $M(X)$),
- a novel internal reduction process that generalizes the Montgomery-like version (the advantage being that there is no need to search for a suitable polynomial $M(X)$),
- a 128-bit version of the two previous approaches,
- a parallel version of all of the above.

Performance wise, the 128-bit version outperforms its 64-bit counterpart only for integers larger than 8192 bits. Unfortunately within this range GnuMP uses faster algorithms so that there is no real point in considering the 128-bit version of PMNS. As a consequence, it is important to carefully consider the specific requirements of each application to determine the most appropriate method for using PMNS. Considering this fact, the roadmap to use PMNS is as follows :

- up to 1024 bits, the 64-bit generalized version is competitive with GnuMP,
- from 1024 to 2048 bits, the Toeplitz splitting method is well suited,
- in the 2048 to 4096 bits range, it appears that PMNS are not well suited to perform modular multiplications,

- above 4096 bits, the parallel version PPMNS is a way to be competitive with GnuMP again.

REFERENCES

1. J.-C. Bajard, Laurent Imbert, and Thomas Plantard, *Modular number systems: Beyond the mersenne family*, Selected Areas in Cryptography, 11th International Workshop, SAC 2004, Waterloo, Canada, 2004, pp. 159–169.
2. Cyril Bouvier and Laurent Imbert, *An alternative approach for sidh arithmetic*, Public-Key Cryptography – PKC 2021 (Cham) (Juan A. Garay, ed.), Springer International Publishing, 2021, pp. 27–44.
3. Richard P. Brent and Paul Zimmermann, *Modern Computer Arithmetic*, Cambridge Monographs on Applied and Computational Mathematics, vol. 18, Cambridge University Press, 2010.
4. Titouan Coladon, Philippe Elbaz-Vincent, and Cyril Hugounenq, *MPHELL: A fast and robust library with unified and versatile arithmetics for elliptic curves cryptography*, ARITH 2021 (Torino, Italy), Transactions on Emerging Topics in Computing, June 2021.
5. Laurent-Stéphane Didier, Fangan Yssouf Dosso, and Pascal Véron, *Efficient modular operations using the Adapted Modular Number System*, Journal of Cryptographic Engineering (2020), 1–23.
6. Fangan Yssouf Dosso, Jean-Marc Robert, and Pascal Véron, *PMNS for Efficient Arithmetic and Small Memory Cost*, IEEE Transactions on Emerging Topics in Computing **10** (2022), no. 3, 1263 – 1277.
7. H. Fan and M.A. Hasan, *Alternative to the karatsuba algorithm for software implementation of GF(2^n) multiplication*, IET Information Security **3** (2009), 60–65(5).
8. M. Anwar Hasan and Christophe Nègre, *Multiway splitting method for toeplitz matrix vector product*, IEEE Transactions on Computers **62** (2013), 1467–1471.
9. Çetin Kaya Koç, Tolga Acar, and Burton S. Kaliski, *Analyzing and comparing montgomery multiplication algorithms*, IEEE Micro **16** (1996), no. 3, 26–33.
10. Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), no. 170, 519–521.
11. Christophe Negre and Thomas Plantard, *Efficient modular arithmetic in adapted modular number system using lagrange representation*, Information Security and Privacy, 13th Australasian Conference, ACISP 2008, Wollongong, Australia, 2008, pp. 463–477.
12. Peter van Emde Boas, *Another np-complete problem and the complexity of computing short vectors in a lattice*, Tech. report, University of Amsterdam, Department of Mathematics,Netherlands, 1981.

Université de Toulon, Institut de Mathématiques, Toulon, France
E-mail address: francois.palma@univ-tln.fr
E-mail address: nicolas.meloni@univ-tln.fr
E-mail address: pascal.veron@univ-tln.fr