Arithmexotiques

Laurent Imbert

Habilitation à diriger les recherches

Université Montpellier 2

Spécialité Informatique

Arithmexotiques

par

Laurent Imbert

Soutenue le 11 avril 2008 devant le jury composé de

Jean-Claude Bajard	
Claude Carlet	
Christiane Frougny	rapporteuse
Graham A. Jullien	
Gilles Villard	rapporteur
Paul Zimmermann	rapporteur

Chapitre 1

Introduction

Lorsqu'en novembre 1859, Bernhard Riemann publie *Über die Anzahl der Primzahlen unter einer gegebenen Größe* (On the number of primes less than a given quantity), il ne mesure certainement pas l'impact qu'aura cet article de 8 pages, le seul qu'il publiera en théorie des nombres, sur les mathématiques de la fin du 19^{e} siècle à nos jours.

En août 1900, lors du congrès international des mathématiciens, dans un amphithéâtre bondé de la Sorbonne, David Hilbert présentait sa fameuse liste de 23 problèmes qui, estimait-il, servirait de cap aux explorateurs mathématiques du 20^{e} siècle. L'hypothèse de Riemann, qui constituaient le 8^{e} problème avec la conjecture de Goldbach, reste aujourd'hui un des rares problèmes non résolus de la liste initiale de Hilbert (certains ayant été partiellement prouvés, d'autres estimés trop vagues pour être considérés comme résolus). Un siècle plus tard, l'hypothèse de Riemann est le seul des 23 problèmes de Hilbert présent dans la liste des 7 problèmes du millénaire (*Millenium prize problems*, cf. table 1.1), établie en 2000 par l'institut de mathématiques Clay (CMI) qui s'est engagé à verser à la première personne qui résoudra un de ces problèmes la somme d'un million de dollars US.

Si la plupart des gens sont prêts à admettre que les mathématiques sont au cœur d'avancées importantes dans de nombreux domaines, comme la conquête de l'espace ou le développement des technologies modernes, rares sont ceux qui imaginent que le monde mystérieux des nombres premiers pourrait avoir une influence quelconque sur notre quotidien. De nombreux mathématiciens du 20^{e} siècle partageaient cet avis, comme G. H. Hardy qui, en 1940 déclarait à propos de la théorie des nombres : « Tant Gauss que les mathématiciens de moindre importance peuvent se réjouir de ce que nous avons là une science, au moins, qui est si éloignée des activités humaines ordinaires qu'elle restera vierge et immaculée. » Il ne faudra pas très longtemps pour que cette « science immaculée » entre dans nos vies de tous les jours et se retrouve au cœur du monde des affaires. À partir de la fin des années 70, deux personnes n'ayant échangé aucune information préalable, allaient pouvoir communiquer en secret. La cryptographie moderne était née en apportant une solution à la fameuse règle 22 de la cryptographie : *Pour qu'Alice and Bob puissent communiquer en secret, ils doivent tout d'abord communiquer en secret.*

En 1976, en effet, un article fondateur de Whitfield Diffie et Martin Hellman [31], définissait les bases de la cryptographie moderne asymétrique en introduisant une méthode permettant à deux personnes d'établir un secret commun en ne s'échangeant que des informations publiques. Ils avaient résolu un des problèmes majeurs de la cryptographie : la distribution des clés secrètes. Lorsque deux ans plus tard, en 1978, Ron Rivest, Adi Shamir et Leonard Adleman proposaient le célèbre algorithme RSA [70], qui permet de signer et de chiffrer des messages, la quête des nombres premiers sortait brutalement des centres universitaires pour devenir une ambitieuse opération commerciale planétaire.

L'explosion d'Internet et du commerce électronique allait voir passer la cryptographie (l'art d'écrire en caractères secrets) du statut d'art-science réservée aux échanges confidentiels de

P versus NP

The question is whether, for all problems for which a computer can verify a given solution quickly (that is, in polynomial time), it can also find that solution quickly. This is generally considered the most important open question in theoretical computer science.

The Hodge conjecture

The Hodge conjecture is that for projective algebraic varieties, Hodge cycles are rational linear combinations of algebraic cycles.

The Poincaré conjecture

In topology, a sphere with a two-dimensional surface is essentially characterized by the fact that it is simply connected. It is also true that every 2-dimensional surface which is both compact and simply connected is topologically a sphere. The Poincaré conjecture is that this is also true for spheres with three-dimensional surfaces. The question had long been solved for all dimensions above three. Solving it for three is central to the problem of classifying 3-manifolds. A proof of this conjecture was given by Grigori Perelman in 2003 ; its review was completed in August 2006, and Perelman was awarded the Fields Medal for his solution. Perelman declined the award.

The Riemann hypothesis

The Riemann hypothesis is that all nontrivial zeros of the Riemann zeta function have a real part of 1/2. A proof or disproof of this would have far-reaching implications in number theory, especially for the distribution of prime numbers. This was Hilbert's eighth problem, and is still considered an important open problem a century later.

Yang-Mills existence and mass gap

In physics, classical Yang-Mills theory is a generalization of the Maxwell theory of electromagnetism where the chromo-electromagnetic field itself carries charges. As a classical field theory it has solutions which travel at the speed of light so that its quantum version should describe massless particles (gluons). However, the deictic phenomenon of color confinement permits only bound states of gluons, forming massive particles. This is the mass gap. Another aspect of confinement is asymptotic freedom which makes it conceivable that quantum Yang-Mills theory exists without restriction to low energy scales. The problem is to establish rigorously the existence of the quantum Yang-Mills theory and a mass gap.

Navier-Stokes existence and smoothness

The Navier-Stokes equations describe the movement of liquids and gases. Although they were found in the 19th century, they still are not well understood. The problem is to make progress toward a mathematical theory that will give us insight into these equations.

The Birch and Swinnerton-Dyer conjecture

The Birch and Swinnerton-Dyer conjecture deals with a certain type of equation, those defining elliptic curves over the rational numbers. The conjecture is that there is a simple way to tell whether such equations have a finite or infinite number of rational solutions. Hilbert's tenth problem dealt with a more general type of equation, and in that case it was proven that there is no way to decide whether a given equation even has any solutions.

Tab. 1.1 – Millenium prize problems

certains organismes gouvernementaux, à un outil de base indispensable aux échanges quotidiens du monde des affaires et des particuliers. Sans le savoir, nous utilisons quotidiennement la cryptographie à clé publique ; par exemple en retirant de l'argent liquide dans un distributeur de billets, ou lors de la saisie et de l'envoi de notre numéro de carte de crédit sur Internet. Des protocoles d'authentification ou de signature permettent par exemple de vérifier l'identité des acteurs d'une transaction électronique. Sans ces protocoles sécurisés, ce type de transactions que beaucoup considèrent aujourd'hui comme naturelles, ne seraient pas possibles.

Pour expliquer comment les nombres premiers interviennent dans ces algorithmes de signature et de chiffrement, je dirais, de manière très schématique, que les propriétés que nous leur connaissons permettent de concevoir ces protocoles, et que tout ce que nous ignorons permet de définir des paramètres assurant leur robustesse. Prenons par exemple l'algorithme RSA, dont la sécurité est basée sur la difficulté de factoriser un entier construit en multipliant entre eux deux grands nombres premiers (plusieurs centaines de chiffres décimaux). Pour de telles tailles, il n'existe pas d'algorithme efficace permettant de retrouver les facteurs premiers à partir de leur produit. Le mystère qui entoure les nombres premiers est donc au cœur de la cryptographie moderne. Les nombres premiers apparaissent aussi dans d'autres protocoles cryptographiques basés sur un autre problème difficile ; le problème du logarithme discret (DLP pour *Discrete Logarithm Problem*) sur un groupe fini : Soit $(G, \times)$ un groupe cyclique fini d'ordre n et soit g un générateur de G. Tout élément $h \in G$ peut s'écrire $h = g^e$ avec $0 \leq e \leq n-1$. Le problème du logarithme discret consiste à retrouver e étant donnés $g, h \in G$. Les protocoles d'échange de clé de Diffie-Hellman, de chiffrement ElGamal ou de signature DSA (*Digital Signature Algorithm*) utilisent le groupe multiplicatif $\mathbb{Z}_p^*$ du corps fini à p éléments, où p est un nombre premier. Bien qu'il existe sur ces groupes des algorithmes de complexité sous-exponentielle pour résoudre le DLP, ils restent encore très utilisés en pratique car les opérations arithmétiques modulo p sont très simples à calculer. Dans la plupart des cas, il est même possible de choisir des nombres premiers particulièrement adaptés à l'arithmétique modulo p, rendant les protocoles encore plus rapides (cf. chapitre 2).

Les courbes elliptiques et les jacobiennes de courbes hyperelliptiques de petit genre, définies sur un corps fini, constituent une autre classe d'excellents outils mathématiques pour ces protocoles cryptographiques. De manière très schématique, une courbe elliptique peut être vue comme l'ensemble des solutions d'une équation à deux variables. L'ensemble de ces solutions permet de définir un groupe abélien fini procurant un très haut niveau de sécurité. En fait, « casser » un système cryptographique basé sur les courbes elliptiques (ou hyperelliptiques de petit genre) est considéré comme un problème extrêmement difficile (de complexité exponentielle) ; largement plus difficile, à taille égale, que la factorisation d'entiers (RSA) ou le problème du logarithme discret sur le groupe multiplicatif d'un corps fini (Diffie-Hellman, DSA). Les paramètres permettant d'obtenir avec des courbes un niveau de sécurité équivalent à ces autres protocoles sont par conséquent beaucoup plus petits, d'où des protocoles plus rapides nécessitant beaucoup moins de mémoire. Malgré de nombreux avantages, l'utilisation des courbes elliptiques en cryptographie reste encore relativement limitée ; RSA restant l'algorithme de référence pour la plupart des acteurs du monde économique. Grâce notamment à un très grand nombre de travaux de la communauté scientifique sur les courbes elliptiques, ce constat pourrait cependant évoluer rapidement en faveur de ces dernières dans un avenir proche. De nombreux standards internationaux fixent des règles d'utilisation pour l'utilisation des courbes elliptiques ; et la société Certicom (basée à Mississauga, Ontario, Canada) qui commercialise des produits basés sur les courbes elliptiques a des clients dans le monde entier, en particulier la *National Security Agency* (NSA) Américaine. Les courbes hyperelliptiques, qui n'ont attiré l'attention de la communauté que plus récemment, commencent à s'affirmer comme une alternative sérieuse aux courbes elliptiques. Des résultats récents ont en effet montré qu'elles peuvent s'avérer aussi, voire plus efficaces que leurs équivalents elliptiques, sans sacrifier le niveau de sécurité. Outre leurs propriétés intéressantes, il

est important de noter que, contrairement aux protocoles basés sur les courbes elliptiques pour lesquels Certicom détient quasiment tous les brevets, la cryptographie des courbes hyperelliptiques reste, pour l'instant, relativement libre d'usage et de droits. (Le chapitre 3 est consacré à l'arithmétique des courbes elliptiques.)

Chapitre 2

Arithmétique modulaire

L'une des premières et plus importantes contributions de Gauss à été l'invention d'une calculatrice virtuelle fonctionnant sur le principe d'un cadran horaire. Sur une horloge classique, 12 et 0 représentent la même valeur. Gauss avança l'idée d'une arithmétique qui fonctionnerait sur le même principe quel que soit le nombre d'heures indiquées sur le cadran. Cette idée offrait la possibilité de faire de l'arithmétique avec des nombres jusqu'alors considérés comme peu maniables. Cette arithmétique d'un nouveau genre, communément appelée arithmétique modulaire, allait révolutionner les mathématiques du 19^{e} siècle jusqu'à nos jours où elle se retrouve au cœur de la cryptographie moderne.

En effet, l'efficacité de la plupart des protocoles cryptographiques asymétriques dépend principalement de la rapidité de l'arithmétique modulaire sous-jacente. La taille des opérandes joue évidemment un rôle prépondérant dans l'efficacité de cette arithmétique. Pour les algorithmes construits sur un anneau fini de type $\mathbb{Z}_n$ (RSA, Diffie-Hellman, DSA, etc), les entiers considérés pour obtenir un niveau de sécurité suffisant font plusieurs milliers de bits. Dans le cas des courbes elliptiques définies sur un corps fini de type $\mathbb{F}_p$ avec p premier, la taille des entiers manipulés est beaucoup plus petite. La table 2.1, dont les valeurs numériques sont issues de [76], présente une synthèse des propriétés et des besoins arithmétiques pour les protocoles cryptographiques les plus communs. Un niveau de sécurité de n bits correspond à la sécurité procurée par un chiffrement symétrique utilisant une clé de n bits.

Niveau de sécurité (en bits)	RSA	DH, DSA	ECC
	$\mathbb{Z}_n$, $n = pq$ p, q premiers (taille de n en bits)	$\mathbb{Z}_p^*$ p premier (taille de p, en bits)	$\mathbb{F}_p$ p premier (taille de p, en bits)
80	1024	1024	160
112	2048	2048	224
128	3072	3072	256
192	4096	4096	384
256	15360	15360	512

Tab. 2.1 – Arithmétique modulaire de quelques protocoles cryptographiques.

L'opération principale en arithmétique modulaire, appelée réduction modulaire, consiste à évaluer le reste de la division entière x/n. Soit $0 \leq x < n^2$ (on suppose généralement que l'entier à réduire est le résultat du produit de deux entiers inférieurs à n), alors il existe $q \geq 0$ et $0 \leq r < n$ tels que $x = qn + r$. La valeur r, appelée le reste, est notée $r = x \bmod n$. Ici « mod » doit être vu comme un opérateur qui calcule l'unique entier $0 \leq r < n$; la relation de congruence étant notée $x \equiv r \pmod{n}$.

Bien évidemment, les algorithmes de division [45, 46], comme la célèbre méthode SRT[1], longtemps implantée dans les unités flottantes des microprocesseurs (on se souvient tous du bug du premier Pentium d'Intel [66]) avant d'être remplacée par des méthodes à base d'évaluation polynomiale, où les méthodes à convergence quadratique de Newton et Goldsmith [48, 44], peuvent être utilisées pour calculer non seulement le quotient, mais aussi le reste d'une division. Néanmoins, pour les applications cryptographiques, il est plus avantageux d'utiliser des algorithmes qui permettent de calculer un reste modulo n sans avoir à évaluer le quotient.

L'algorithme de Barrett [13] permet de calculer $r = x \bmod n$ sans effectuer de division (sauf dans la phase de précalcul). On suppose que n s'écrit sur k chiffres en base b, avec $b_{k-1} \neq 0$ et que x s'écrit sur $2k$ chiffres (cette condition est bien vérifiée pour $x < n^2$). On suppose aussi que la valeur $\mu = \lfloor b^{2k}/n \rfloor$ est précalculée. En raison de ce précalcul faisant intervenir une division par n, l'algorithme de Barrett n'est intéressant que lorsque plusieurs réductions modulo n doivent être calculées, comme pour une exponentiation modulaire. L'idée de Barrett consiste à calculer le reste $r = x - qn$ à partir d'une approximation $\tilde{q}$ du quotient. Plus précisément, plutôt que d'évaluer

$$q = \lfloor x/n \rfloor = \left\lfloor \frac{(x/b^{k-1})(b^{2k}/n)}{b^{k+1}} \right\rfloor, \tag{2.1}$$

l'algorithme de Barrett calcule

$$\tilde{q} = \left\lfloor \frac{\lfloor x/b^{k-1} \rfloor \, \mu}{b^{k+1}} \right\rfloor, \tag{2.2}$$

où les divisions par b^{k-1} et b^{k+1} se ramènent à de simples décalages (à droite) dans la base b. On remarque que $-b^{k+1} < r_1 - r_2 < b^{k+1}$ et par conséquent, après l'étape 4 de l'algorithme 1 ci-dessous, on a $0 < \tilde{r} < b^{k+1}$. On vérifie aussi aisément la double inégalité suivante

$$\frac{x}{n} - \frac{b^{k-1}}{n} - \frac{x}{b^{2k}} + \frac{1}{b^{k+1}} \leq \frac{\lfloor x/b^{k-1} \rfloor \, \mu}{b^{k+1}} \leq \frac{(x/b^{k-1})(b^{2k}/n)}{b^{k+1}}$$

qui nous assure que la valeur $\tilde{q}$ calculée vérifie $q-2 \leq \tilde{q} \leq q$. Si $r = x-qn$, on montre que la valeur $\tilde{r}$ calculée après l'étape 4 vérifie $\tilde{r} \equiv (q - \tilde{q})n + r \pmod{b^{k+1}}$. De plus, $(q - \tilde{q})n + r < 3n < b^{k+1}$ pour $b > 3$, ce qui nous assure que la boucle « tant que » de l'étape 5 ne sera jamais exécutée plus de 2 fois. (Voir [59, chapitre 14] pour une analyse fine du coût de l'algorithme de Barrett.)

Algorithme 1 Réduction modulaire de Barrett

Entrées : $x = (x_{2k-1}, \ldots, x_1, x_0)_b, \quad n = (n_{k-1}, \ldots, n_1, n_0)_b$ avec $n_{k-1} \neq 0$

Sorties : $0 \leq x \bmod n < n$

1: $q_0 = \lfloor x/b^{k-1} \rfloor \mu, \quad \tilde{q} = \lfloor q_0/b^{k+1} \rfloor$

2: $r_1 = x \bmod b^{k+1}, \quad r_2 = \tilde{q}n \bmod b^{k+1}, \quad \tilde{r} = r_1 - r_2$

3: **Si** $\tilde{r} < 0$ **Alors**

4: $\quad \tilde{r} = \tilde{r} + b^{k+1}$

5: **Tant que** $\tilde{r} \geq n$ **faire**

6: $\quad \tilde{r} = \tilde{r} - n$

7: **Renvoyer** $\tilde{r}$

En simulant une division entière, on peut voir l'algorithme de Barrett comme une réduction modulaire poids forts en tête. À l'opposé, l'algorithme de Montgomery [62] effectue la réduction par les poids faibles. Pour calculer $r = x \bmod n$, l'idée très élégante de Montgomery repose sur le

[1] Nommée ainsi par Freiman [47] en l'honneur de Sweeney, Robertson [71] et Tocher qui l'auraient découvert indépendamment vers la fin des années 50.

fait qu'on ne modifie pas la valeur de r en ajoutant à x n'importe quel multiple de n. Le but étant de faire en sorte que le résultat de $x+kn$ soit facilement divisible par une valeur m prédéfinie, en général une puissance de la base de numération. Par exemple, si n est impair, et en choisissant $m = 2$ (de telle sorte que $\text{pgcd}(m, n) = 1$) alors, que l'on calcule $x + n$ pour x impair, ou $x + 2n$ pour x pair, le résultat obtenu sera toujours pair ; on peut donc facilement le diviser par $m = 2$ par un simple décalage et obtenir une valeur congrue à $x/2 \pmod n$. L'algorithme de réduction modulaire de Montgomery présenté ci-dessous (cf. algorithme 2) généralise ce principe. La valeur précalculée $n' = -n^{-1} \bmod m$ nous permet de calculer la valeur q telle que $x+qn$ soit un multiple de m, et par conséquent divisible par m. Cette valeur n'existe que si n et m sont premiers entre eux. En pratique, n est un nombre premier, ou le produit impair de deux nombres premiers, et m est une puissance de 2. Comme pour Barrett, le précalcul de Montgomery, (n' peut être

Algorithme 2 Réduction modulaire de Montgomery (MR)

Entrées : $x < n^2$

Sorties : $r = xm^{-1} \bmod n$

1: $q = xn' \bmod m$
2: $r = (x + qn)/m$
3: **Si** $r \geq n$ **Alors**
4: $\quad r = r - n$
5: **Renvoyer** r

obtenu grâce à l'algorithme d'Euclide étendu), devient négligeable lorsque plusieurs réduction modulo n sont effectuées. Dans le cadre d'une exponentiation, il faut cependant gérer le facteur m^{-1} qui apparaît à chaque réduction. En fait, plutôt que de calculer $x^e \bmod n$ directement, on commence par effectuer un changement de représentation $x \to \tilde{x} = xm \bmod n$: en supposant que $m' = m^2 \bmod n$ est précalculé, le passage dans la représentation de Montgomery se fait en calculant $\text{MR}(xm')$ (on a bien $xm' < n^2$). L'avantage de cette représentation est d'être stable par MR. Par exemple, $\text{MR}\left(\tilde{x}^2\right) \equiv x^2m^2m^{-1} \bmod n \equiv x^2m \bmod n$ que je note $\tilde{x}_2$ (la valeur retournée par MR étant inférieure à n, on a bien le résultat souhaité). Le résultat final de l'exponentiation $x^e \bmod n$ est obtenu par un dernier appel à $\text{MR}(\tilde{x}_e)$ (si l'algorithme d'exponentiation se termine par le produit $\tilde{x}_{e-1} \times \tilde{x}$, on peut économiser un appel à MR en réduisant $\tilde{x}_{e-1}x < n^2$). Le surcoût engendré par les changements de représentations est donc négligeable par rapport au coût de l'exponentiation. L'algorithme de Montgomery a été très étudié ; voir [59, chapitre 14] pour une analyse détaillée de sa complexité et [25] pour une bonne étude comparative de différentes options d'implémentation.

Il existe des situations où l'arithmétique modulaire peut être grandement accélérée ; en particulier lorsqu'il est possible de choisir le corps fini sur lequel sont effectués les calculs. Dans de tels cas, on privilégie généralement des entiers (premiers) dont la forme est particulièrement bien adaptée à l'arithmétique modulaire. Les nombres de Mersenne de la forme $p = 2^n - 1$ constituent une famille, a priori très intéressante, puisque la réduction modulo p se réduit à l'addition (modulo p) de la partie haute et de la partie basse du nombre que l'on désire réduire. En effet, soit $x < p^2$. Pour calculer x modulo p, il suffit d'écrire $x = x_1 2^n + x_0$ avec $x_0, x_1 < 2^n$ et d'utiliser la relation de congruence $2^n \equiv 1 \pmod p$ pour obtenir que $x \equiv x_0 + x_1 \pmod p$. Si le résultat dans $\mathbb{Z}$ de l'addition $x_0 + x_1$ est supérieur à p, le résultat final est obtenu en calculant $(x_0 + x_1) \bmod 2^n + 1 < p$. Le coût total de la réduction modulo un nombre de Mersenne est donc très proche du coût d'une addition (exactement une addition dans $\mathbb{Z}$ plus éventuellement une addition de 1). Les nombres de Mersenne ressemblent donc aux candidats parfaits. Malheureusement, il n'existe que très peu de nombres de Mersenne premiers pour les tailles cryptographiques usuelles (cf. table 2.2). La recherche de Mersenne premiers de plus en plus grands est un problème

M_2	3
M_3	7
M_5	31
M_7	127
M_{13}	8191
M_{17}	131071
M_{19}	524287
M_{31}	2147483647
M_{61}	2305843009213693951
M_{89}	618970019642690137449562111
M_{107}	162259276829213363391578010288127
M_{127}	170141183460469231731687303715884105727
M_{521}	6864797660130609714981259074082891115057151

TAB. 2.2 – Quelques nombres de Mersenne premiers.

difficile. Le plus grand nombre de Mersenne premier connu à ce jour, découvert en septembre 2006 dans le cadre du GIMPS[2] (*Great Internet Mersenne Prime Search*), est l'entier $2^{32582657}-1$ de plus de $9,8$ millions de chiffres décimaux !

Les nombres de Mersenne premiers étant malheureusement très rares, plusieurs auteurs, dont R. Crandall [30] ont proposé d'utiliser des entiers dont la forme et les propriétés arithmétiques sont proches de celles des Mersenne. Ces entiers, baptisés pseudo-Mersenne, sont de la forme $p = 2^n - c$, où c est un petit entier. En utilisant la même décomposition que précédemment et la relation de congruence $x \equiv x_1 c + x_0 \pmod{p}$, la réduction modulo un pseudo-Mersenne se ramène à une multiplication par un petit entier et un petit nombre d'additions/soustractions (dans $\mathbb{Z}$). Il est beaucoup plus facile de trouver des pseudo-Mersenne que des Mersenne premiers pour les tailles cryptographiques ; par exemple, le pseudo-Mersenne $p = 2^{255} - 19$ est un nombre premier de 255 bits qui a été utilisé pour accélérer les calculs sur les courbes elliptiques [14] (cf. chapitre 3). Il est cependant regrettable que Crandall ait préféré déposer un brevet [30], plus ou moins controversé[3], plutôt qu'en publiant un article.

Les nombres premiers proposés dans [77] par le SECG (*Standards for Efficient Cryptography Group*) pour l'implémentation des protocoles basés sur les courbes elliptiques (cf. table 2.3) sont tous des pseudo-Mersenne premiers ; ils sont tous de la forme $2^n - c$ avec c petit, mais on remarque que $c > 2^{32}$ (sauf `secp160r1` et `secp521r1` aussi recommandé par le NIST [67]). Cette contrainte, a priori inutile et surprenante puisqu'il existe des courbes sûres pour des valeurs de c plus petites, pourrait bien venir, encore une fois, de ce brevet déposé en 1991 par Crandall qui spécifie (*Claim 2*) : « where c is a binary number having a length no greater than 32 bits ». (Si c'est le cas, je n'explique pas les valeurs `secp160r1` et `secp521r1` !).

Peut être encore à cause (ou grâce) à ce brevet, la communauté s'est intéressée à rechercher d'autres formes de nombres premiers adaptés à l'arithmétique modulaire. En 1999, J. Solinas [74] introduit la famille des Mersenne généralisés de la forme $p = f(t)$, où f est un polynôme unitaire à coefficients entiers et t une puissance de 2. Par exemple, l'entier $p = 2^{192} - 2^{64} - 1$ peut être défini par le polynôme $f(t) = t^3 - t - 1$ évalué en $t = 2^{64}$. Ainsi, tout entier $x < p^2$ peut s'écrire comme un polynôme de degré ≤ 5, sous la forme $x = \sum_{i=0}^{5} x_i t^i$. Pour calculer $r = x \bmod p$, il

[2] `http://www.mersenne.org/prime.htm`

[3] Allez donc faire un tour à l'adresse `http://cr.yp.to/patents/us/5159632.html`.

`secp160k1`	$2^{160} - 2^{32} - 2^{14} - 2^{12} - 2^9 - 2^8 - 2^7 - 2^3 - 2^2 - 1$
`secp160r1`	$2^{160} - 2^{31} - 1$
`secp160r2`	$2^{160} - 2^{32} - 2^{14} - 2^{12} - 2^9 - 2^8 - 2^7 - 2^3 - 2^2 - 1$
`secp192k1`	$2^{192} - 2^{32} - 2^{12} - 2^8 - 2^7 - 2^6 - 2^3 - 1$
`secp192r1`	$2^{192} - 2^{64} - 1$
`secp224k1`	$2^{224} - 2^{32} - 2^{12} - 2^{11} - 2^9 - 2^7 - 2^4 - 2 - 1$
`secp224r1`	$2^{224} - 2^{96} + 1$
`secp256k1`	$2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$
`secp256r1`	$2^{224}(2^{32} - 1) + 2^{192} + 2^{96} - 1$
`secp384r1`	$2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$
`secp521r1`	$2^{521} - 1$

TAB. 2.3 – Les premiers recommandés par le groupe SECG pour les courbes elliptiques sur $\mathbb{F}_p$.

suffit d'utiliser les relations de congruences :

$$\begin{aligned} t^3 &\equiv t + 1 \pmod{p}, \\ t^4 &\equiv t^2 + t \pmod{p}, \\ t^5 &\equiv t^2 + t + 1 \pmod{p}, \end{aligned}$$

et d'en déduire les coefficients du polynôme (de degré 2) représentant le reste :

$$\begin{aligned} r_0 &= x_0 + x_3 + x_5, \\ r_1 &= x_1 + x_3 + x_4 + x_5, \\ r_2 &= x_2 + x_4 + x_5. \end{aligned}$$

Le résultat entier est obtenu en évaluant le polynôme $r = r_2t^2 + r_1t + r_0$ pour $t = 2^{64}$. L'entier considéré dans l'exemple précédent fait partie des 5 nombres premiers recommandés par le NIST dans le standard FIPS 186-2 [67] pour l'utilisation des courbes elliptiques sur $\mathbb{F}_p$ (cf. table 2.4). On remarque que, comme dans les propositions du SECG, l'entier p_{521} est un nombre de Mersenne.

Valeur	polynôme	t
$p_{192} = 2^{192} - 2^{64} - 1$	$t^3 - t - 1$	2^{64}
$p_{224} = 2^{224} - 2^{96} + 1$	$t^7 - t^3 + 1$	2^{32}
$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$	$t^8 - t^7 + t^6 + t^3 - 1$	2^{32}
$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$	$t^{12} - t^4 - t^3 + t - 1$	2^{32}
$p_{521} = 2^{521} - 1$	$t - 1$	2^{521}

TAB. 2.4 – Les Mersenne généralisés proposés par le NIST.

En 2003, J. Chung et A. Hasan ont publié un article intitulé *more generalized Mersenne numbers* [26], où ils étendent l'idée de Solinas en autorisant n'importe quelle valeur pour t. Leur approche permet d'engendrer, grâce à des valeurs de t différentes, plusieurs nombres premiers de même taille avec le même polynôme de départ. La même arithmétique polynomiale peut donc être réutilisée par plusieurs implémentations.

Tous les entiers de la grande famille des Mersenne, pseudo-Mersenne, Mersenne généralisés, Mersenne généralisés de Chung et Hasan possèdent un point commun : les propriétés des nombres

dépendent essentiellement de leur écriture en base 2 (ou en base t dans le cas des moduli de Chung et Hasan). Dans le cadre de son travail de thèse, Thomas Plantard s'est intéressé à la recherche d'autres formes de nombres adaptés à l'arithmétique modulaire. Dans un article publié à SAC 2004 [12] nous avons proposé un nouveau système, appelé système de numération modulaire, ou MNS de l'anglais *Modular Number System*, pour représenter les éléments et calculer sur $\mathbb{Z}_n$.

Dans un système classique de position [65], on représente les entiers (positifs) en base β sous la forme usuelle, où la position de chaque chiffre $x_i \in \{0, \ldots, \beta - 1\}$ correspond à une puissance de la base de numération :

$$x = \sum_{i=0}^{k-1} x_i \beta^i, \tag{2.3}$$

Si $x_{k-1} \neq 0$, on dit que l'entier x s'écrit sur k chiffres en base β.

Définition 1 (MNS)**.** Un Système de Numération Modulaire $\mathcal{B}$ est défini par un quadruplet (γ, ρ, n, p). Dans ce système, tout entier $0 \leq x < p$ peut s'écrire sous la forme

$$x = \sum_{i=0}^{n-1} x_i \gamma^i \bmod p, \tag{2.4}$$

avec $1 < \gamma < p$ et $x_i \in \{0, \ldots, \rho - 1\}$. Le vecteur $(x_0, x_1, \ldots, x_{n-1})_{\mathcal{B}}$ correspond à l'écriture de l'entier x dans la base $\mathcal{B}$.

Comme pour les Mersenne généralisés, l'arithmétique des MNS manipule essentiellement des polynômes, et comme pour les nombres de Chung et Hasan, la valeur de γ est libre. Par contre, l'évaluation de (2.4) se faisant dans $\mathbb{Z}_p$, on peut choisir une valeur de γ arbitrairement grande (de l'ordre de p), en essayant de faire en sorte que les coefficients x_i soient aussi petits que possible ($< \rho$).

Exemple 1. Soit le MNS défini par $\gamma = 7, \rho = 3, n = 3, p = 17$. Dans ce système, on représente les éléments de $\mathbb{Z}_{17}$ comme des polynômes de degré ≤ 2, à coefficients dans $\{0, 1, 2\}$. Il est facile

0	1	2	3	4	5
0	1	2	$\gamma + 2\gamma^2$	$1 + \gamma + 2\gamma^2$	$\gamma + \gamma^2$

6	7	8	9	10	11
$1 + \gamma + \gamma^2$	γ	$1 + \gamma$	$2 + \gamma$	$2\gamma + 2\gamma^2$	$1 + 2\gamma + 2\gamma^2$

12	13	14	15	16	
$2\gamma + \gamma^2$	$1 + 2\gamma + \gamma^2$	2γ	$1 + 2\gamma$	$2 + 2\gamma$	

TAB. 2.5 – Les éléments de $\mathbb{Z}_{17}$ dans la base $\mathcal{B} = MNS(7, 3, 3, 17)$.

de vérifier que l'évaluation modulo $p = 17$ de chaque polynôme de la table 2.5 en $\gamma = 7$ donne bien les entiers de $\mathbb{Z}_{17}$.

Définition 2 (AMNS)**.** Un système de numération modulaire $\mathcal{B} = MNS(\gamma, \rho, n, p)$ est dit adapté (AMNS pour *Adapted Modular Number System* en anglais) si $\gamma^n \bmod p = c \in \mathbb{Z}$, avec c petit. On le note alors $\mathcal{B} = AMNS(\gamma, \rho, n, p, c)$.

Dans ce système, l'addition correspond à une addition polynomiale ; les coefficients de la somme peuvent être supérieurs à ρ mais il n'est pas toujours nécessaire d'effectuer une réduction des coefficients immédiatement après l'addition. Voir [12] pour plus de détails.

Comme dans [26], la multiplication modulaire dans un AMNS est effectuée en trois étapes (cf. algorithme 3). Les deux premières étapes font appel à des calculs simples sur des polynômes ;

Algorithme 3 Multiplication modulaire dans un AMNS

Entrées : Un AMNS $\mathcal{B} = (\gamma, \rho, n, p, c)$, et deux entiers (polynômes) $A = (a_0, ..., a_{n-1})_\mathcal{B}$, $B = (b_0, ..., b_{n-1})_\mathcal{B}$ dans $\mathcal{B}$

Sorties : Le polynôme $S = (s_0, ..., s_{n-1})_\mathcal{B}$ tel que $S(\gamma) = A(\gamma)B(\gamma) \bmod p$

1: Multiplication polynomiale (dans $\mathbb{Z}[X]$) : $U(X) = A(X)B(X)$

2: Réduction polynomiale : $V(X) = U(X) \bmod (X^n - c)$

3: Normalisation : $S = CR(V)$, tel que $S(\gamma) \equiv V(\gamma) \pmod p$ et $s_i < \rho$ pour $i = 0, \ldots, n-1$.

la troisième consiste à normaliser les coefficients du polynôme obtenu à l'étape 2. Cette dernière étape, la plus coûteuse, est équivalente à une réduction modulo un réseau euclidien.

Le coût des deux premières étapes de l'algorithme 3 est facile à évaluer (voir [12] pour les détails). Pour l'étape 1, la multiplication de deux polynômes de degré $< n$ dont les coefficients sont inférieurs à ρ demande n^2 produits entre opérandes de tailles $\log_2(\rho)$ bits, et $(n-1)^2$ additions de nombres de taille $\log_2(n\rho^2)$ bits. L'étape 2 demande $n-1$ produits par la constante c (le coefficient à multiplier par c s'écrit sur $\log_2(n\rho^2)$ bits), et $n-1$ additions de nombres de taille $\log_2(cn\rho^2)$ bits. Pour c petit ou creux, le coût des multiplications par c est négligeable.

L'algorithme de multiplication dans un AMNS montre clairement deux niveaux d'arithmétique modulaire : la réduction polynomiale et la normalisation des coefficients. La réduction modulo $(X^n - c)$, que l'on peut qualifier de réduction « externe », est une opération facile car c est un petit entier. En revanche, la réduction « interne » des coefficients est plus difficile. Même si ces niveaux sont moins visibles, l'arithmétique multiprécision, où les entiers de $\mathbb{Z}_p$ sont représentés sur plusieurs mots de w bits, possède aussi deux niveaux d'arithmétique modulaire : la réduction « externe » modulo p (difficile) et la réduction « interne » implicite modulo 2^w (facile).

La réduction des coefficients consiste à calculer, à partir du polynôme V obtenu à l'étape 2, un polynôme S qui représente le même entier (i.e., tel que $S(\gamma) \equiv V(\gamma) \pmod p$) et dont les coefficients sont tous $< \rho$. Pour simplifier l'analyse, et sans perte de généralité, on fixe $\rho = 2^{k+1}$, et on représente les polynômes comme des vecteurs de coefficients ; ainsi le vecteur $V = (v_0, \ldots, v_{n-1})$ représente le polynôme $V(x) = v_0 + v_1 x + \cdots + v_{n-1}x^{n-1}$. La phase de normalisation consiste à trouver une représentation de V dont les éléments sont tous inférieurs à ρ, c'est-à-dire de taille au plus $k+1$ bits.

Tout coefficient v_i de V peut s'écrire sous la forme $v_i = \underline{v_i} + \overline{v_i}\, 2^k$, avec $\underline{v_i} < 2^k$. De même,

$$V = \underline{V} + \overline{V} \cdot 2^k I, \tag{2.5}$$

où I est la matrice identité de taille $n \times n$. La décomposition (2.5) montre que si nous sommes capables de trouver une représentation du vecteur $\overline{V} \cdot 2^k I$ dont les éléments sont tous inférieurs à 2^k, alors nous obtenons une représentation de V dont les coefficients sont $< \rho$. L'idée consiste à définir une matrice M avec de petits coefficients qui vérifie

$$M \cdot (1, \gamma, \ldots, \gamma^{n-1})^T \equiv 2^k I \cdot (1, \gamma, \ldots, \gamma^{n-1})^T \pmod p, \tag{2.6}$$

De manière schématique, la matrice M peut être vue comme une représentation de $2^k I$ dans notre AMNS.

Si $2^k = (z_0, \ldots, z_{n-1})_\mathcal{B}$ est une représentation de 2^k dans notre AMNS, la définition 1 nous dit que $2^k \equiv z_0 + z_1\gamma + \cdots + z_{n-1}\gamma^{n-1} \pmod p$. En multipliant par γ et en utilisant la relation

$\gamma^n \equiv c \pmod p$, nous obtenons $\gamma 2^k \equiv cz_{n-1} + z_0\gamma + \cdots + z_{n-2}\gamma^{n-1} \pmod p$. On procède ainsi jusqu'à obtenir la matrice M suivante :

$$M = \begin{pmatrix} z_0 & z_1 & \cdots & & z_{n-1} \\ cz_{n-1} & z_0 & \cdots & & z_{n-2} \\ \vdots & & & & \\ cz_1 & cz_2 & \cdots & cz_{n-1} & z_0 \end{pmatrix} \tag{2.7}$$

Cette matrice vérifie (2.6), et donc $\overline{V} \cdot 2^k I \equiv \overline{V} \cdot M \pmod p$. L'équation (2.5) peut donc s'écrire

$$V = \underline{V} + \overline{V} \cdot M. \tag{2.8}$$

Cette équation est à la base de l'algorithme de normalisation, ou plus exactement d'un algorithme de réduction partielle (cf. algorithme 4). En effet, en procédant comme décrit ci-dessus, nous ne pouvons pas réduire les coefficients de V directement. Par contre, si les coefficients de M sont suffisamment petits (la condition exacte étant $c\sum_{i=0}^{n-1} z_i < 2^{\lfloor k/2 \rfloor}$) nous pouvons transformer des coefficients de taille $\lceil 3k/2 \rceil$ bits en des coefficients de taille $k+1$ bits. Pour réduire des coefficients

Algorithme 4 Réduction partielle : $\lceil 3k/2 \rceil$ bits $\longrightarrow k+1$ bits

Entrées : un AMNS $\mathcal{B} = (\gamma, \rho, n, P, c)$ avec $\rho = 2^{k+1}$; une représentation de $2^k = (z_0, ..., z_{n-1})_{\mathcal{B}}$ avec $c\sum_{i=0}^{n-1} z_i < 2^{\lfloor k/2 \rfloor}$; la matrice M définie par (2.7) ; un vecteur $V = (v_0, \ldots, v_{n-1})$ tel que $v_i < 2^{\lceil 3k/2 \rceil}$ pour $i = 0 \ldots n-1$.

Sorties : Un vecteur $S = (s_0, ..., s_{n-1})$ tel que $s_i < \rho$ pour $i = 0 \ldots n-1$.

1: Décomposer V en $V = \underline{V} + \overline{V} \cdot 2^k I$

2: Calculer $S = \underline{V} + \overline{V} \cdot M$

de taille $> 3k/2$ bits, il suffit d'appliquer itérativement l'algorithme 4 de réduction partielle à partir des poids forts. (Voir [12] pour les détails, les preuves et l'étude de la complexité.) L'exemple suivant illustre la multiplication dans un AMNS.

Exemple 2. Soient $\gamma = 127006, k = 6, \rho = 128, n = 3, p = 250043$. Puisque $\gamma^3 \equiv 2 \pmod p$, on définit $\mathcal{B} = AMNS(127006, 128, 3, 250043, 2)$. Il est facile de vérifier que $1 + \gamma^2 \bmod p = 2^6$; on a donc $2^k = (1, 0, 1)_{\mathcal{B}}$ et d'après (2.7),

$$M = \begin{pmatrix} 1 & 0 & 1 \\ 2 & 1 & 0 \\ 0 & 2 & 1 \end{pmatrix}$$

Soient $a = 65842$ et $b = 8816$ deux entiers représentés respectivement par les polynômes $A(X) = 7 + 30X + 100X^2$ et $B(X) = 59 + 2X + 76X^2$ (on vérifie que $A(\gamma) \bmod p = a$ et $B(\gamma) \bmod p = b$). On commence par calculer le produit

$$U(X) = A(X) \times B(X) = 413 + 1784X + 6492X^2 + 2480X^3 + 7600X^4,$$

que l'on réduit modulo $X^3 - 2$ pour obtenir

$$V(X) = 5373 + 16984X + 6492X^2.$$

L'algorithme 4 de réduction partielle ne permettant de réduire que des coefficients de taille ≤ 9 bits, on procède par étape en commençant par les bits de poids forts. Le plus grand coefficient de V faisant 15 bits, on « supprime » les $15 - 9 = 6$ bits de poids faibles de chaque coefficient de V pour ne garder que les bits de poids $> 2^6$. Tous les coefficients du vecteur $V_0 = (83, 265, 101)_{\mathcal{B}}$ ainsi construit sont $< 2^9$ et sont donc réduits en calculant $S_0 = \underline{V_0} + \overline{V_0} \cdot M = (28, 15, 39)_{\mathcal{B}}$.

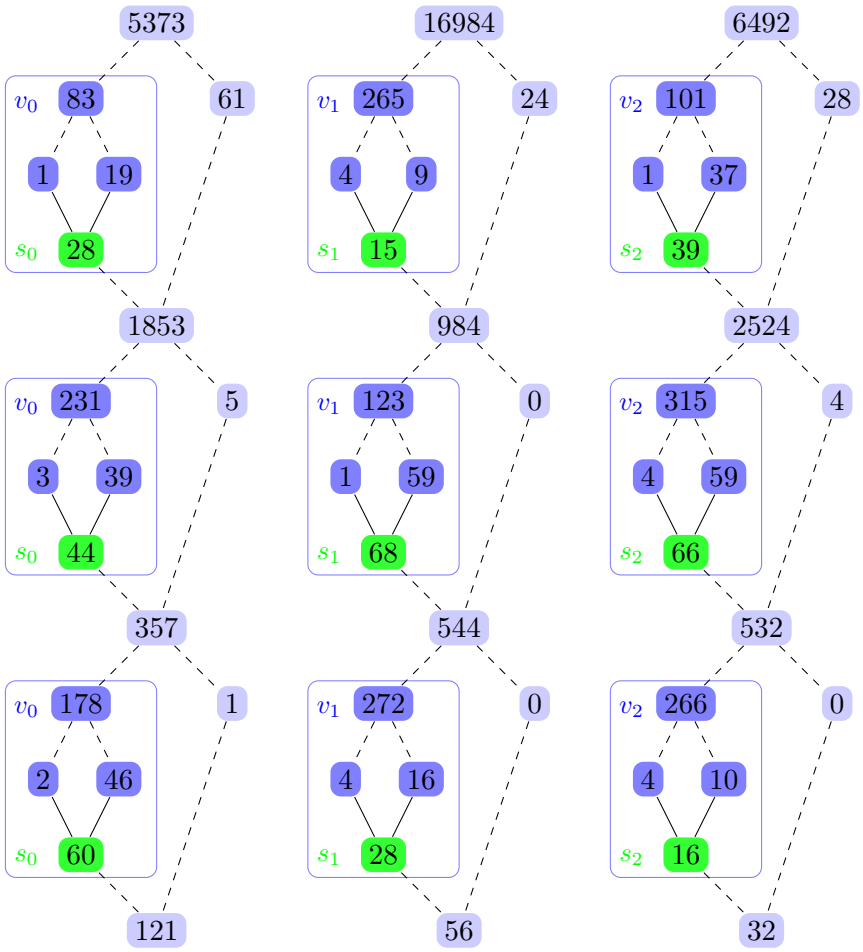

FIG. 2.1 – Décomposition du calcul de $65842 \times 8816 \bmod 250043$. Les parties encadrées correspondent à l'algorithme 4.

On termine la première étape de réduction en « recollant » les coefficients de S_0 aux bits de V de poids $< 2^6$ que l'on avait supprimés. On obtient le vecteur $V_1 = (1853, 984, 2524)_{\mathcal{B}}$, dont les coefficients sont tous de taille $\leq k + 1 + 6 = 13$ bits. On recommence ainsi jusqu'à ce que le vecteur S obtenu soit complètement réduit, comme illustré sur la figure 2.1. On peut vérifier que le vecteur (polynôme) $S_3 = (121, 56, 32)_{\mathcal{B}}$ correspond bien à l'entier $113269 = a \times b \bmod p$ et que ses coefficients sont tous inférieurs à ρ.

Les paramètres p et γ choisis pour l'exemple précédent définissent effectivement un AMNS très intéressant puisque l'étape de réduction des coefficients ne demande qu'un petit nombre d'additions et de décalages (18 additions suffisent pour l'exemple 2). Tous les nombres premiers p ne permettent pas de définir un AMNS, et parmi ceux qui le permettent, il existe des choix $(n, \gamma$, etc) plus intéressants que d'autres. Pour trouver de bons candidats, la stratégie est la suivante : On choisit $\rho = 2^{k+1}$ en fonction de l'architecture visée (en pratique, $k = 15, 31$ ou 63), puis on fixe n de telle sorte que $(k+1) \times n$ corresponde à peu près à la taille du corps souhaité. Pour que l'étape de réduction des coefficients soit très peu coûteuse, on impose des conditions sur les coefficients z_i (par exemple, $z_i \in \{0, 1, 2\}$) et on n'autorise que de petites valeurs pour c. En fonction de ces choix, on calcule, si il en existe, des valeurs pour p et γ qui définissent un AMNS. On déduit p du calcul de déterminant $d = \left|2^k \cdot I - M\right|$ qui, d'après (2.6), doit être un multiple de p, et on déduit γ du calcul des racines de $\gcd(X^n - c, 2^k - \sum_i z_i X^i) \bmod p$. Nous donnons un exemple de recherche d'AMNS pour la cryptographie des courbes elliptiques.

Exemple 3. Nous cherchons un corps $\mathbb{F}_p$ de taille au moins égale à 2^{160}. Pour ce faire, nous fixons $\rho = 2^{16}$ et $n = 11$. Nous choisissons par exemple $c = 3$ et supposons que 2^k s'écrit $(1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1)_{\mathcal{B}}$, c'est-à-dire $2^k = 1 + x^5 + x^8 + x^9 + x^{10}$. Nous calculons le déterminant

$$d = \left|2^k \cdot I - M\right| = 467523550650744744856027134573563377101619107673 27$$

qui possède comme facteur premier un nombre de 160 bits :

$$p = 792412797713126686196656160294175215426473063853.$$

Nous calculons une racine modulo p de $\gcd(X^{11} - 3, 2^{15} - 1 - x^5 - x^8 - x^9 - x^{10})$ pour obtenir

$$\gamma = 474796736496801627149092588633773724051936841406.$$

Dans sa thèse de doctorat [68], Thomas Plantard que j'ai eu le plaisir d'encadrer, propose des paramètres d'AMNS pour des corps premiers de taille cryptographiques allant de 128 à 512 bits. Il reste maintenant à coder ces résultats pour valider la méthode.

Chapitre 3

Arithmétique des courbes elliptiques

De manière très générale, une courbe elliptique est définie par l'ensemble des solutions d'une équation à deux variables. En fait, ces objets mathématiques sont apparus dans la résolution de nombreux problèmes bien avant la naissance de la cryptographie à clé publique.

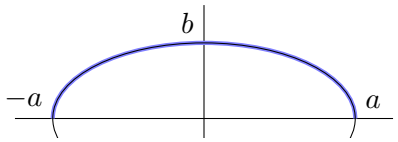

FIG. 3.1 – Arc d'une demi-ellipse d'équation $x^2/a^2 + y^2/b^2 = 1$.

Par exemple, pour mesurer la longueur ℓ de l'arc d'une demi-ellipse (figure. 3.1), on doit calculer l'intégrale suivante

$$\ell = \int_{-a}^{a} \sqrt{\frac{a^2 - (1 - b^2/a^2)x^2}{a^2 - x^2}} dx.$$

En posant $k^2 = 1 - b^2/a^2$ et après le changement de variable $x \to ax$ on obtient une intégrale particulière appelée intégrale elliptique

$$\ell = a \int_{-1}^{1} \frac{(1 - k^2x^2)}{\sqrt{(1 - x^2)(1 - k^2x^2)}} dx.$$

La fonction apparaissant au dénominateur définie par $y^2 = (1 - x^2)(1 - k^2x^2)$ est une courbe elliptique, dite réelle, dont le graphe ressemble à la courbe de la figure 3.2.

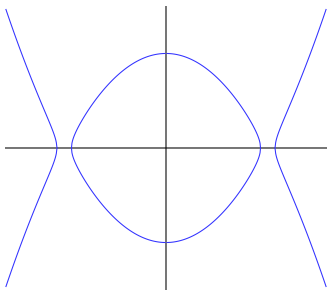

FIG. 3.2 – La courbe elliptique d'équation $y^2 = (1 - x^2)(1 - 3x^2/4)$.

Le même type d'intégrale elliptique intervient dans le calcul exact de la position d'un pendule pesant en fonction du temps pour de grandes oscillations. De nombreux problèmes de gravitation

ou d'électromagnétisme ont été résolus par des calculs d'intégrales elliptiques ; on en retrouve en particulier dans des travaux d'Euler et de Gauss datant des 18e et 19e siècles. Mais les courbes elliptiques interviennent aussi dans la résolution de nombreux problèmes en théorie des nombres. Par exemple, déterminer pour quelles valeurs de n la série $1 + 2^2 + 3^2 + \cdots + n^2$ est un carré parfait revient à calculer les solutions de l'équation $y^2 = x(x+1)(2x+1)/6$. Cette équation de degré 3, représentée sur la figure 3.3, est une courbe elliptique dite imaginaire. C'est ce genre de courbes (définies sur un corps fini) que nous utilisons en cryptographie.

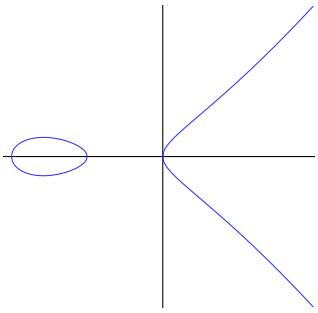

FIG. 3.3 – La courbe elliptique d'équation $y^2 = x(x+1)(2x+1)/6$.

Une courbe elliptique est donc un objet géométrique ; c'est une courbe non singulière définie par une équation de la forme $y^2 = f(x)$, où f est un polynôme de degré 3 ou 4 en x sans racine double. Les solutions (x, y) de cette équation, auxquelles on ajoute un point particulier $\mathcal{O}$, appelé point à l'infini, forment l'ensemble des points d'une courbe elliptique :

$$E = \{(x, y); y^2 = f(x)\} \cup \{\mathcal{O}\}.$$

Étant donnés deux points $(x_1, y_1), (x_2, y_2)$ de E, on peut définir, à l'aide de règles simples, un troisième point (x_3, y_3) appartenant à E. Une courbe elliptique est donc aussi un objet algébrique, au sens où il est possible d'« additionner » des points comme on le ferait avec des nombres. De manière plutôt surprenante, cette addition de points se décrit géométriquement de manière simple et élégante (voir figure 3.4). Si $P \neq Q$, le point $R = P + Q$ est le symétrique par rapport à l'axe des abscisses du point d'intersection entre la courbe et la droite (P, Q) ; on calcule $2R$ de la même manière, en considérant la tangente à la courbe au point R. Les droites de pente infinie passent toutes par le point $\mathcal{O}$; l'opposé du point de coordonnées (x, y) est donc le point $(x, -y)$.

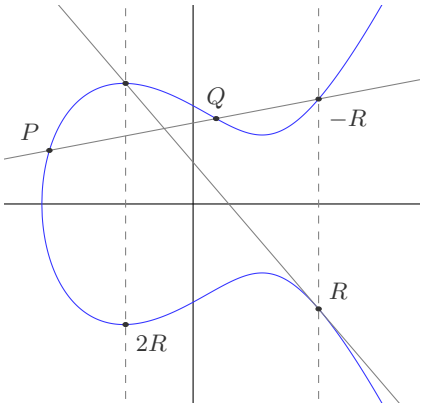

FIG. 3.4 – Interprétation géométrique de la loi d'addition sur une courbe elliptique.

Pour tout points P, Q, R de E, la loi $+$ décrite ci-dessus vérifie : $P + \mathcal{O} = \mathcal{O} + P = P$, $P + (-P) = \mathcal{O}$, $(P + Q) + R = P + (Q + R)$ et $P + Q = Q + P$. Muni de cette loi d'addition,

l'ensemble $(E, +)$ forme donc un groupe abélien.

En traduisant les règles géométriques, cette loi d'addition se décrit aisément de manière analytique. considérons les points $P = (x_1, y_1)$ et $Q = (x_2, y_2)$ sur une courbe d'équation $y^2 = x^3 + ax + b$. On a

$$P + Q = (\lambda^2 - x_1 - x_2, -\lambda^3 + 2\lambda x_1 + \lambda x_2 - y_1), \tag{3.1}$$

où $\lambda = (y_2 - y_1)/(x_2 - x_1)$ si $P \neq \pm Q$ et $\lambda = (3x_1^2 + a)/2y_1$ si $P = Q$ (si $P = -Q$ on a $P + (-P) = \mathcal{O}$). En observant les coordonnées de $P + Q$ dans (3.1), on remarque que si les paramètres a et b, et les coordonnées de P et Q sont des nombres rationnels, alors les coordonnées des points $P + Q$ et $2P$ sont elles aussi rationnelles.

La remarque précédente est à l'origine d'un résultat crucial, démontré par H. Poincaré à l'aube du 20$^{\text{e}}$ siècle : l'ensemble $E(\mathbb{Q})$ des points rationnels d'une courbe E définie sur $\mathbb{Q}$ est un sous-groupe de E. Depuis l'antiquité, la recherche des solutions entières ou rationnelles d'une équation polynomiale est un problème fondamental en théorie des nombres. L'étude des courbes elliptiques sur $\mathbb{Q}$ va fournir aux mathématiciens du début du 20$^{\text{e}}$ siècle des outils précieux pour l'étude de ces équations diophantiennes. Certainement le plus célèbre de ces résultats est la preuve par A. Wiles du dernier théorème de Fermat qui dit que pour $n > 2$, l'équation $a^n + b^n = c^n$ n'a pas de solutions entières (non nulles). En fait, Wiles à réussi à démontrer la non existence de courbes elliptiques très particulières, construites « sur mesure » par G. Frey à partir des solutions hypothétiques de l'équation $a^n + b^n = c^n$.

Les courbes elliptiques que nous avons tracées jusqu'ici étaient définies sur $\mathbb{R}$. Les propriétés de ces courbes, en particulier la structure de groupe commutatif est préservée quel que soit le corps sur lequel est définie la courbe. Comme nous l'avons vu pour $E(\mathbb{Q})$, nous pouvons définir des courbes elliptiques sur le corps des nombres complexes, sur un corps fini $\mathbb{F}_q$, sur des corps de nombres, etc. L'étude des fonctions elliptiques (fonctions complexes doublement périodiques) et des réseaux euclidiens permet de montrer qu'une courbe elliptique définie sur $\mathbb{C}$ est équivalente à un tore (voir par exemple [80, chapitre 9]).

Les courbes elliptiques définies sur un corps fini sont à l'origine des applications cryptographiques. La formule (3.1) d'addition/doublement s'applique de la même manière aux points à coordonnées dans $\mathbb{F}_q$, comme nous le montrons dans l'exemple 4 pour une courbe définie sur $\mathbb{F}_{37}$. Sur la figure 3.5 on peut remarquer la symétrie et le fait que la droite passant par P et Q coupe la courbe en un unique troisième point $(-R)$ dont le symétrique est égal à $P + Q$.

Exemple 4. Soit $E : y^2 = x^3 - 5x + 8$ la courbe définie sur le corps premier $\mathbb{F}_{37}$, et soient $P = (6, 3)$ et $Q = (9, 10)$ deux points de $E(\mathbb{F}_{37})$. Le petit programme Magma [24] ci-dessous permet de calculer facilement l'ensemble des points de E (cf. table 3.1).

```
fp_37 := FiniteField(37);
E := EllipticCurve( [fp_37 | -5, 8] );
O := E ! 0;
Pts := { E ! [x, Sqrt(P)] : x in fp_37 | IsSquare(P) where P is x^3 - 5*x + 8 };
Pts join:= { -P : P in Pts };
Pts join:= { O };
```

Les formules d'addition et de doublement vues plus haut, où tous les calculs sont faits modulo 37, nous permettent de calculer $2P = (35, 11)$, $3P = (34, 25)$, $4P = (8, 6)$, $P + Q = (11, 10)$, $3P + 4Q = (31, 28)$, etc.

Les mathématiciens de l'époque se sont naturellement intéressés à compter le nombre de points d'une courbe elliptique définie sur un corps fini. C'est Hasse qui en 1922 démontra le résultat suivant sur l'ordre du groupe des points d'une courbe elliptique sur un corps fini :

$$|\#E(\mathbb{F}_q) - q - 1| \leq 2\sqrt{q}. \tag{3.2}$$

Pour la courbe $y^2 = x^3 - 5x + 8$ sur $\mathbb{F}_{37}$ de l'exemple précédent, on vérifie facilement que $|\#E(\mathbb{F}_{37}) - q - 1| = 7 \leq 2\sqrt{37} \approx 12.17$. Compter le nombre de points d'une courbe elliptique

(10, 12)	(21, 32)	(9,10)	(11, 27)	(1, 35)	(22, 1)	(12, 14)	(33, 1)	(34, 25)
(19, 1)	(17, 10)	(6, 34)	(31, 9)	(21, 5)	(5, 21)	(34, 12)	(9, 27)	(11, 10)
(26, 8)	(22, 36)	(16, 19)	(17, 27)	(35, 26)	(16, 18)	(20, 8)	(5, 16)	(20, 29)
(1, 2)	(33, 36)	(30, 25)	(19, 36)	(28, 8)	(12, 23)	(28, 29)	(35, 11)	(6, 3)
(30, 12)	(10, 25)	(8, 31)	(36, 30)	(36, 7)	(31, 28)	(26, 29)	(8, 6)	$\mathcal{O}$

TAB. 3.1 – L'ensemble des points de la courbe elliptique $y^2 = x^3 - 5x + 8$ définie sur $\mathbb{F}_{37}$.

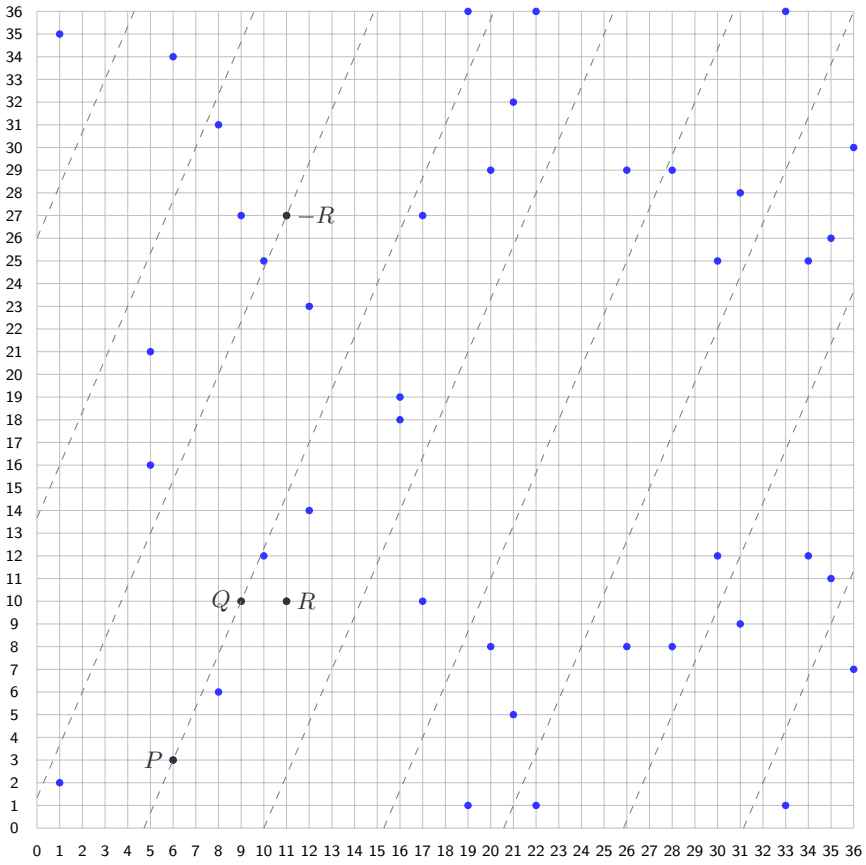

FIG. 3.5 – La courbe elliptique $y^2 = x^3 - 5x + 8$ et la loi de groupe définies sur le corps fini $\mathbb{F}_{37}$.

fait partie des problématiques importantes en cryptographie ; c'est une étape indispensable dans la recherche de courbes cryptographiquement sûres.

L'utilisation cryptographique du groupe des points d'une courbe elliptique définie sur un corps fini a été proposée indépendemment par N. Koblitz [55] et V. Miller [60] dans les années 80. Même si, pour des raisons purement économiques, elles tardent à s'imposer dans le monde de l'industrie et du commerce, les avantages procurés par les courbes elliptiques ont rapidement convaincu la communauté universitaire. Plusieurs ouvrages leur sont consacrées [19, 49, 28, 20]. La sécurité de ces protocoles repose sur la difficulté, étant donnés P et kP deux points d'une courbe, de calculer l'entier k ; c'est la version additive du problème du logarithme discret, noté ECDLP suivant l'acronyme anglais. Les meilleurs algorithmes connus pour résoudre le ECDLP, comme les méthodes baby-step giant-step ou Pollard's rho, nécessitent de l'ordre de $\sqrt{n}$ opérations, où $n = \#E/h$ (le cofacteur h doit absolument être petit ; idéalement $h = 1$). Pour un niveau de sécurité s donné, une des méthodes permettant de définir une courbe sûre consiste à choisir aléatoirement une courbe sur $\mathbb{F}_q$ avec $q \approx 2^{2s}$ et d'utiliser un algorithme de comptage de points générique, comme les algorithmes de Schoof [72] ou Schoof-Elkies-Atkin [73], jusqu'à obtenir une courbe d'ordre quasi-premier. Par exemple, pour un niveau de sécurité de 128 bits, on cherchera une courbe d'ordre premier ou quasi-premier ($h \leq 4$) sur un corps fini $\mathbb{F}_q$ de cardinal $q \approx 2^{256}$. Ce qui revient à manipuler des nombres 12 fois plus petits que pour RSA ou DSA pour un niveau de sécurité équivalent (cf. table 2.1, chapitre 2). Notons que d'autres précautions doivent être prises pour le choix du corps de base afin d'éviter certaines attaques spécifiques, comme la descente de Weil sur les extensions non premières de $\mathbb{F}_2$, ou l'attaque MOV lorsque n divise $p^k - 1$ avec k petit.

Néanmoins, effectuer un grand nombre de calculs sur des « nombres » de 200 à 500 bits, le plus rapidement possible, nécessite des algorithmes et des implantations optimisés. Nous avons présenté quelques idées pour accélérer les calculs au niveau du corps fini dans le chapitre 2. Les opérations arithmétiques à optimiser au niveau de la courbe elliptique se situent à deux niveaux : les formules d'addition et de doublement (nous verrons plus loin d'autres primitives intéressantes comme le triplement) ; et les algorithmes de multiplication scalaire (équivalent additif d'une exponentiation) permettant de calculer kP où k est un grand entier. Naturellement, les meilleures solutions prennent en compte ces deux niveaux conjointement.

Une première méthode élémentaire pour le calcul de $kP = P + \cdots + P$ (k fois) consiste à parcourir les bits de k en partant des poids forts ; on effectue un doublement pour chaque bit de k, suivi d'une addition pour chaque bit non nul[1]. Fort logiquement, cet algorithme a été baptisé double-and-add en anglais. Dans les deux cas, le coût total est de $n-1$ doublements et $n/2$ additions en moyenne, où n représente le nombre de bits de k. En remarquant que le calcul de l'opposé d'un point sur une courbe elliptique est quasiment gratuit, une première optimisation naturelle consiste à considérer une représentation signée [22] du scalaire k et à remplacer les additions par des soustractions pour les chiffres de k valant -1. En pratique, on utilise une décomposition particulière de k, appelée CSD (Canonic Sign Digit) dans la communauté arithmétique des ordinateurs et NAF (Non-Adjacent Form) dans la communauté cryptographique.

$$\mathrm{NAF}(k) = \sum_{i=0}^{n} k_i 2^i, \quad \text{avec } k_i \in \{-1, 0, 1\} \text{ et } k_i k_{i+1} = 0. \tag{3.3}$$

Parmi toutes les représentations signées de k (avec $k_i \in \{-1, 0, 1\}$), il est possible de montrer que le nombre de chiffres non nuls de $\mathrm{NAF}(k)$ est minimal. Cette décomposition, facile à calculer (voir [69] ou [52] pour une version poids forts en tête) permet de réduire à $n/3$ le nombre moyen d'additions (voir [23] pour une analyse précise) pour une multiplication scalaire ; le nombre de doublements passant éventuellement de $n - 1$ à n. Une deuxième source d'accélération est

[1] Il existe une version quasiment équivalente à partir poids faibles.

l'utilisation d'un petit nombre de précalculs que l'on peut exploiter en écrivant l'entier k en base 2^w, ce qui revient à découper k en fenêtres de taille w, ou à l'aide de fenêtres glissantes, en écrivant k sous la forme

$$\mathrm{NAF}_w(k) = \sum_{i=0}^{n} k_i 2^i, \text{ avec } |k_i| < 2^{w-1}, \tag{3.4}$$

Pour w donné, la décomposition $\mathrm{NAF}_w(k)$ est unique. Elle possède au plus 1 chiffre non nul parmi w chiffres consécutifs et chaque chiffre $k_i \neq 0$ est impair. Cette famille de représentations a été très étudiée. Le lecteur intéressé pourra par exemple consulter [29, 27, 2, 64, 78] pour plus de détails.

Exemple 5. Soit $k = 314159$. ($\overline{c}$ représente le chiffre négatif $-c$.)

$$\begin{aligned}
(n)_2 &= 1\,0\,0\,1\,1\,0\,0\,1\,0\,1\,1\,0\,0\,1\,0\,1\,1\,1\,1\\
\mathrm{NAF}_2(n) &= 1\,0\,1\,0\,\overline{1}\,0\,1\,0\,\overline{1}\,0\,\overline{1}\,0\,1\,0\,\overline{1}\,0\,0\,0\,\overline{1}\\
\mathrm{NAF}_3(n) &= 1\,0\,0\,0\,3\,0\,0\,1\,0\,0\,3\,0\,0\,0\,3\,0\,0\,0\,\overline{1}\\
\mathrm{NAF}_4(n) &= 0\,0\,5\,0\,0\,0\,\overline{3}\,0\,0\,0\,\overline{5}\,0\,0\,0\,3\,0\,0\,0\,\overline{1}
\end{aligned}$$

Sous cette forme, le nombre moyen d'additions nécessaires pour une multiplication scalaire est égal à $n/(w+1)$, sans compter les quelques précalculs (les points jP pour $j = 1, 3, \ldots, 2^{w-1}-1$).

Certaines familles de courbes elliptiques possèdent des propriétés supplémentaires permettant d'accélérer encore les opérations. Citons par exemple :

- Les courbes de Montgomery définies sur $\mathbb{F}_p$ ($by^2 = x^3 + ax^2 + x$) et les courbes non-supersingulières sur $\mathbb{F}_{2^m}$ ($y^2+xy = x^3+ax^2+b$) proposées par Lopez et Dahab qui utilisent un algorithme de multiplication scalaire adapté, appelé échelle de Montgomery [63], qui permet de ne calculer que la coordonnée x de $P+Q$ lorsque les points P, Q et $P-Q$ sont connus [61] ;
- les courbes de Koblitz [56] définies sur $\mathbb{F}_2$ ($y^2 + xy = x^3 + ax^2 + 1$) qui exploitent l'endomorphisme de Frobenius $\tau : (x, y) \mapsto (x^2, y^2)$ au travers d'un algorithme de multiplication scalaire de type τ-and-add [75] ;
- Les familles DIK2 (resp. DIK3) [41] qui possèdent un doublement (resp. triplement) rapide obtenu par composition d'isogénies de degrés 2 (resp. 3) ;
- Les courbes d'Edwards [43] et leurs dérivées [15] ($x^2 + y^2 = c^2(1 + dx^2y^2)$) définies sur un corps de caractéristique $\neq 2$ pour lesquelles la loi de groupe est complète et homogène (on calcule $P + Q$ et $2P$ grâce à la même formule, et le point à l'infini est un point affine).

L'origine de mes contributions dans ce domaine vient d'un algorithme proposé par Dimitrov [32] pour le calcul matriciel $I + A + \cdots + A^{t-1}$, qui faisait intervenir dans le codage de l'exposant t des bits en base 2 et des « bits » en base 3. Par la suite, cet algorithme a donné lieu à un système de numération, appelé double-base number system (DBNS) [39], et à plusieurs applications en traitement numérique du signal [40, 33, 38]. Dans ce système, tout nombre entier strictement positif était représenté comme une somme de puissances combinées de 2 et 3 :

$$k = \sum_{i,j} d_{i,j} 2^i 3^j, \text{ avec } d_{i,j} \in \{0, 1\}. \tag{3.5}$$

Ce codage se visualise aisément comme un tableau de $0, 1$ à deux dimensions, où chaque ligne correspond au codage en base 2 du facteur d'une puissance de 3.

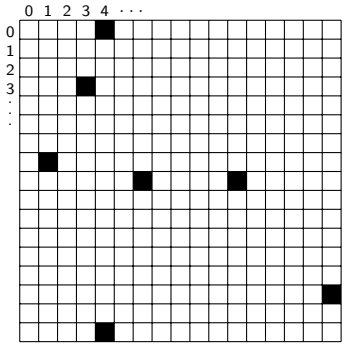

FIG. 3.6 – Un codage possible de l'entier 314159265358 en DBNS.

Exemple 6. Dans ce système, l'entier $k = 314159265358$ peut s'écrire comme $k = 2^{16}3^{14}+2^{4}3^{16}+2^{11}3^{8}+2^{6}3^{8}+2^{1}3^{7}+2^{3}3^{3}+2^{4}3^{0}$ et peut se visualiser comme sur la figure 3.6 (les carrés noirs correspondent aux bits non nuls du codage $d_{i,j} = 1$).

Plutôt que de considérer le codage précédent, il est souvent plus commode de manipuler des expansions et d'autoriser l'utilisation de chiffres signés sous la forme

$$k = \sum_{i=1}^{n} s_i 2^{a_i} 3^{b_i}, \text{ avec } s_i \in \{-1, 1\} \text{ et } a_i, b_i \geq 0 \text{ et } (a_i, b_i) \neq (a_j, b_j) \iff i \neq j. \tag{3.6}$$

La taille (ou longueur) d'une expansion DBNS est le nombre n de termes de (3.6).

Ce système possède de nombreuses propriétés intéressantes, en particulier pour certaines applications cryptographiques, et cache, sous son apparente simplicité, de nombreux problèmes d'approximation diophantienne et de théorie des nombres.

Une première question naturelle concerne le nombre de représentations d'un entier donné dans ce système très redondant. Pour tout entier $k > 0$, la fonction récursive ci-dessous retourne le nombre de représentations de k sous la forme initiale (3.5), c.-à-d. avec des chiffres pris dans $\{0, 1\}$ uniquement. (Voir [35] pour une preuve via les fonctions génératrices.)

$$f(k) = \begin{cases} 1 & \text{si } k = 1, \\ f(k-1) + f(k/3) & \text{si } k \equiv 0 \pmod 3, \\ f(k-1) & \text{sinon.} \end{cases} \tag{3.7}$$

On remarque facilement que les entiers de cette suite vont par triplets ($f(3k) = f(3k+1) = f(3k+2)$). On montre aussi qu'ils correspondent au nombre de partitions de $3k$ sous la forme $3k = k_0 + 3k_1 + 9k_2 + \cdots + 3^t k_t$, dont les premiers termes sont 1, 2, 3, 5, 7, 9, 12, 15, 18, 23, 28, 33, 40, 47, 54, 63, 72, 81, 93, 105, 117. (cf. la suite A005704 de l'encyclopédie en-ligne des suites d'entiers : the on-line encyclopedia of integer sequences).

Pour un entier k donné, trouver une expansion de longueur minimale est un problème difficile qui devient rapidement hors de portée de nos ordinateurs, en particulier pour des entiers de tailles cryptographiques. Il est néanmoins possible de calculer une décomposition DBNS de longueur raisonnable en utilisant une approche gloutonne qui consiste à rechercher le $\{2, 3\}$-entier (un entier dont les seuls facteurs premiers sont 2 et 3) $z = 2^a 3^b$ le plus proche de k (ou le plus proche par défaut pour une décomposition non signée) et de continuer tant que $k := k - z$ est différent de zéro. Cet algorithme ne garantit pas une décomposition de longueur minimale (le premier exemple est 41 ; l'algorithme glouton retourne $41 = 36 + 4 + 1$ alors qu'une représentation optimale est $41 = 32 + 9$) mais il est possible de démontrer [39] qu'il renvoie une somme de $O(\log k / \log\log k)$ termes. La preuve de ce résultat important utilise un théorème de

Tijdeman [79] qui dit qu'il existe une constante C telle qu'on trouve toujours un $\{2,3\}$-entier dans l'intervalle $[k - k/(\log k)^C, k]$.

Cette approche gloutonne nécessite de pouvoir calculer le $\{2,3\}$-entier le plus proche d'un entier donné. Je me suis intéressé à cette question en proposant un algorithme de complexité asymptotique optimale [18]. Intéressons-nous, pour commencer, à la recherche du plus grand $\{2,3\}$-entier inférieur ou égal à k ; on cherche deux entiers $a, b \geq 0$ tel que

$$2^a 3^b = \max\{2^c 3^d \leq k \ ; \ c, d \geq 0\}. \tag{3.8}$$

ou, de manière équivalente

$$a \log 2 + b \log 3 \leq \log k, \tag{3.9}$$

avec la contrainte supplémentaire qu'il n'existe pas de meilleure approximation par défaut de $\log k$.

Soit $\alpha = \log_3 2$. Les solutions de l'équation $c\alpha + d < \log_3 x$ sont les points $(c, d) \in \mathbb{Z}^2$ situés sous la droite Δ d'équation $v = -\alpha u + \log_3 x$ (en gris sur la figure 3.7). Parmi ces solutions, la meilleure approximation par défaut est le point (a, b) dont la distance verticale à la droite est minimale : on note $\delta(u) = \{-\alpha u + \log_3 x\}$, où $\{.\}$ représente la partie fractionnaire.

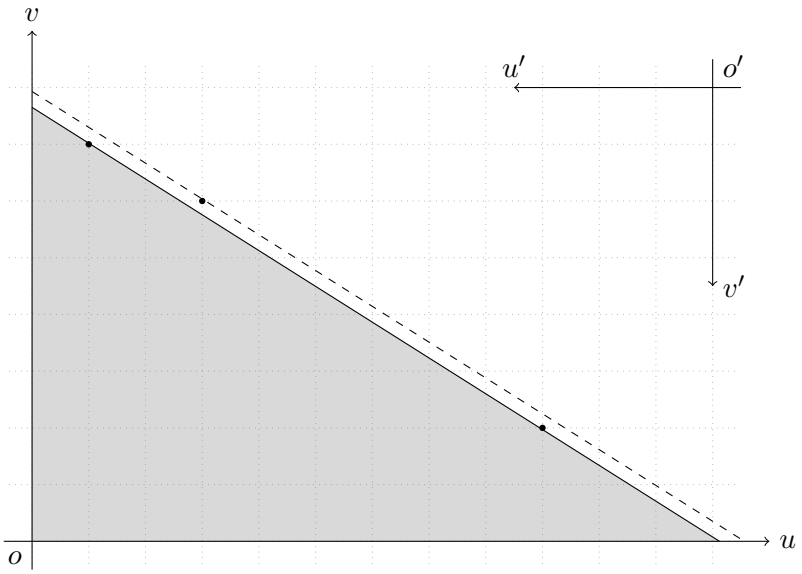

FIG. 3.7 – Interprétation graphique du problème de la recherche du $\{2,3\}$-entier le plus proche d'un entier donné.

Dans certains cas, comme pour trouver une expansion DBNS signée ($s_i \in \{-1, 1\}$) d'un entier, il peut s'avérer utile de calculer la meilleure approximation à droite, i.e. le point (a_r, b_r) le plus proche au dessus de la droite (toujoura au sens de la distance δ). Ce point est obtenu simplement par symétrie : il s'agit du point (a', b') le plus proche en dessous de la droite Δ' d'équation $v' = \alpha u' + (1 - \{\log_3 x\})$ (la même droite que Δ dans le repère (o', u', v'), représentée dans le repère (o, u, v) en pointillé sur la figure 3.7) auquel on applique les transformations $a_r = \lfloor \log_2 x \rfloor - a'$, $b_r = \lceil \log_3(x) \rceil - b'$.

Exemple 7. La figure 3.7 illustre un exemple pour $k = 4444$. On a $\log_2 k \simeq 12.1176$, $\log_3 k \simeq 7.6454$. La meilleure approximation à gauche est le point de coordonnées $(1, 7)$. La meilleure approximation à droite est le point $(12 - 3, 8 - 6) = (9, 2)$. On vérifie aisément que $1\alpha + 7 \simeq 7.6309 < \log_3 x < 9\alpha + 2 \simeq 7.6784$.

Une approche évidente à cette question consiste à calculer l'entier qui minimise $\delta(u)$ pour $0 \leq u \leq \lfloor \log_2 k \rfloor$; le point le plus proche sous la droite étant le point de coordonnées $(u, \lfloor v \rfloor)$. La complexité de cet algorithme est $O(\log k)$. Est-il possible de faire mieux ?

Dans [17, 18], nous avons proposé un algorithme basé sur le système de numération d'Ostrowski [1] pour les nombres réels [16]. Ce système est associé à la suite (p_n/q_n) des convergents du développement en fraction continues d'un irrationnel $\alpha \in]0,1[$. Tout irrationnel α admet un développement simple en fraction continues infini sous la forme

$$\alpha = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}}$$

avec $a_0 = \lfloor \alpha \rfloor$ et $a_i \geq 1$ pour $i > 0$. On note de manière condensée $\alpha = [a_0, a_1, a_2, a_3, \dots]$. On appelle réduite de rang n la fraction continue $[a_0, a_1, a_2, \dots, a_n] = p_n/q_n$. On démontre que les réduites sont les meilleures approximations fractionnaires de α (voir [53]).

Dans le système d'Ostrowski, tout réel $-\alpha \leq \beta < 1 - \alpha$ s'écrit de manière unique sous la forme

$$\beta = \sum_{i=1}^{+\infty} b_i (q_i \alpha - p_i) \pmod 1, \tag{3.10}$$

avec $0 \leq b_1 \leq a_1 - 1$; $0 \leq b_k \leq a_k$, pour $k > 1$; $b_k = 0$ si $b_{k+1} = a_{k+1}$ et $b_k \neq a_k$ pour une infinité d'entiers pairs et une infinité d'entiers impairs.

L'algorithme que nous démontrons dans [18] utilise le fait que β peut être approché modulo 1 par des nombres de la forme $N\alpha$ (avec N entier) ; les meilleures approximations successives étant données par la suite

$$N_j = \sum_{i=1}^{j} b_i q_{i-1}. \tag{3.11}$$

Le cas qui nous intéresse pour la recherche de la meilleure approximation par défaut est un peu plus délicat en raison des changements de signe de la suite $(q_n\alpha - p_n)$. Nous considérons une décomposition de β sous la forme

$$\beta = \sum_{i=1}^{+\infty} c_i \, |q_i \alpha - p_i| \pmod 1 \tag{3.12}$$

et un algorithme qui construit une suite croissante en s'autorisant de soustraire des termes d'ordre i lorsque l'ajout de $|q_{i+1}\alpha - p_{i+1}| \pmod 1$ rend la suite supérieure à β. (Voir [18] pour plus de détails.) Notre algorithme a une complexité asymptotique de $O(\log\log k)$, ce qui rend l'algorithme glouton de décomposition en DBNS optimal en $O(\log k)$.

Revenons maintenant aux motivations cryptographiques de mon travail, et voyons comment j'ai exploité cette décomposition en DBNS de l'entier k pour accélérer la multiplication scalaire sur les courbes elliptiques.

D'après (3.6), on peut écrire

$$kP = \sum_{i=1}^{n} s_i 2^{a_i} 3^{b_i} P, \text{ avec } s_i \in \{-1, 1\}. \tag{3.13}$$

Il est facile de voir qu'une implémentation directe de cette formule n'est pas satisfaisante car elle peut demander jusqu'à $\sum_i a_i$ doublements, $\sum_i b_i$ triplements et n additions. Pour réduire de manière significative ce coût, nous avons introduit la notion d'expansions double-bases chaînées, ou *double-base chains* en anglais [34], qui permettent d'évaluer kP « à la Horner », en réutilisant tous les calculs intermédiaires. Pour ce faire, on représente $k > 0$ sous la forme $k = \sum_{i=1}^{n} s_i 2^{a_i} 3^{b_i}$ avec la contrainte supplémentaire que les exposants forment deux suites décroissantes : $a_1 \geq a_2 \geq$

$\cdots \geq a_n \geq 0$ et $b_1 \geq b_2 \geq \cdots \geq b_n \geq 0$. De manière plus formelle, une expansion DBNS chaînée est une suite $(C_i)_{i>0}$ qui vérifie $C_1 = 1$ et pour $i > 1$

$$C_{i+1} = 2^{u_i} 3^{v_i} C_i + s, \text{ avec } s \in \{-1, 1\} \text{ et } u_i, v_i \geq 0. \tag{3.14}$$

Une expansion DBNS chaînée calcule k si il existe $n > 0$ tel que $C_n = k$.

L'algorithme de décomposition glouton décrit plus haut s'adapte facilement au calcul de telles chaînes [35] et permet de déduire un algorithme générique de multiplication scalaire sur une courbe elliptique qui nécessite $n - 1$ additions, $a_1 = \max_i a_i$ doublements et $b_1 = \max_i b_i$ triplements. Le coût exact de cet algorithme, c'est-à-dire le nombre d'opérations (additions, multiplications, carrés, inversions) dans le corps sur lequel est défini la courbe dépend évidemment du coût des opérations de doublements, de triplement et d'addition dans ce corps. Dans [34], nous détaillons les cas des courbes génériques sur $\mathbb{F}_p$ en coordonnées Jacobiennes, et des courbes sur $\mathbb{F}_{2^m}$ en coordonnées Jacobiennes et en coordonnées affines. Le cas le plus intéressant est celui des courbes sur $\mathbb{F}_p$ car nous avons réussi à trouver un nouvel algorithme de triplement très efficace. La suite logique de ce travail, publiée dans [42] est une généralisation des expansions DBNS chaînées où on autorise un espace de chiffres un peu plus large et quelques précalculs. Cette dernière version est, à ce jour, l'algorithme le plus rapide pour calculer kP sur une courbe générique définie sur $\mathbb{F}_p$.

Chapitre 4

Autres contributions

Arithmétique des corps finis de moyenne caractéristique

Les algorithmes basés sur les courbes elliptiques [55, 60] nécessitent une arithmétique efficace sur les corps finis $\mathbb{F}_q$, où q est une puissance d'un nombre premier. La plupart des travaux concernent les corps premiers $\mathbb{F}_p$, où p est un nombre premier d'au moins 160 bits, et les extensions de type $\mathbb{F}_{2^m}$, de degré $m > 160$. Devant ce constat, nous avons décidé d'étudier la pertinence des extensions de petite et moyenne caractéristiques, c'est-à-dire les corps finis de la forme $\mathbb{F}_{p^k}$ avec $p > 3$. Avec J.-C. Bajard et C. Nègre, nous avons tout d'abord étendu l'algorithme de multiplication modulaire de Montgomery à ce type de corps. Puis, nous avons proposé une version modifiée de cet algorithme, où les calculs sont effectués sur un anneau de cardinal légèrement supérieur à l'ordre du corps fini considéré, et qui permet de nombreuses variantes. Nous avons décliné cette version modifiée de la multiplication de Montgomery en trois versions :

1. en représentant les éléments du corps $\mathbb{F}_{p^k}$ (polynômes de degré inférieur à k sur $\mathbb{F}_p$) sous une forme dite de Lagrange, c'est-à-dire, par leurs valeurs en un nombre suffisant de points plutôt que par leurs coefficients [9, 11].

2. En utilisant le théorème des restes Chinois pour distribuer les calculs dans $\mathbb{F}_p[X]/(\Psi)$ sur plusieurs anneaux $\mathbb{F}_p[X]/(\psi_i)$, où les polynômes ψ_i sont des binômes de même degré [7].

3. En utilisant une décomposition similaire où Ψ s'écrit comme le produit de trinômes de petit degré, pour les extensions de caractéristique 2 [6].

À ma connaissance, les variantes 2 et 3 correspondent aux premiers algorithmes généraux[1] de complexité sous quadratique qui ne font pas appel à la transformée de Fourier rapide (FFT).

Pour être complètement fonctionnelle dans le cadre d'un protocole ECC, notre multiplication en représentation de Lagrange se devait d'être complétée par un algorithme efficace de division. Avec J.-C. Bajard et C. Nègre, nous avons proposé une version modifiée de l'algorithme d'Euclide étendu pour le calcul de l'inverse d'un élément de $\mathbb{F}_{p^k}$ [10].

Multiplication par une constante

Toute multiplication entière peut s'exprimer à l'aide d'additions et de décalages. Le problème de la multiplication par une constante est proche de celui de la multiplication scalaire. Étant donné une constante (entière) c, comment calculer le produit $c \times x$ en minimisant le nombre d'additions ? Dans ce modèle de complexité, on suppose que les décalages sont gratuits. Cette hypothèse constitue une différence essentielle avec le problème de la multiplication scalaire

[1] C' est-à-dire, qui s'appliquent quel que soit le polynôme irréductible de degré k sur $\mathbb{F}_p$ utilisé pour définir le corps $\mathbb{F}_{p^k}$.

évoqué plus haut. Les algorithmes utilisés pour ce dernier ne sont donc pas efficaces pour le problème de la multiplication par une constante. Grâce à sa faible densité, le DBNS reste cependant un système bien adapté. Après une première étude encourageante, essentiellement expérimentale [36], j'ai proposé, avec Vassil Dimitrov et Andrew Zakaluzny, le premier algorithme de complexité sous-linéaire [37].

Plate-forme arithmétique et algorithmique pour la cryptographie

Nous avons récemment initié un projet de développement d'une bibliothèque de calcul sur les corps finis et les courbes elliptiques [51] dont l'objectif est de mesurer de manière précise et fiable plusieurs quantités au cours de l'exécution d'un algorithme cryptographique, comme le nombre de cycles, le nombre d'opérations de chaque type, la quantité de mémoire utilisée, la variation du poids de Hamming d'un registre, etc. Cette bibliothèque est développée en C++ et s'appuie sur les mécanismes de polymorphisme statique dans un souci d'efficacité et pour faciliter l'interchangeabilité de l'arithmétique au sein des protocoles. Elle sera distribuée sous licence GPL. Je serai l'encadrant principal d'une thèse orientée autour de cette bibliothèque, proposée au LIRMM pour la rentrée 2008/2009 (Plate-forme arithmétique et algorithmique pour la cryptographie ; co-encadrant : Pascal Giorgi).

Applications du RNS en cryptographie

L'efficacité de beaucoup d'algorithmes rencontrés en cryptographie asymétrique, comme RSA ou ElGamal [59], dépend en grande partie de la capacité à calculer rapidement les deux opérations que sont le produit modulaire ($x \times y \bmod n$) et l'exponentiation modulaire ($x^e \bmod n$), où les données x, y, e et n sont de grands entiers. En RNS (*Residue Number System*) [54], les entiers sont représentés par leurs restes modulo les éléments d'un ensemble de nombres premiers entre eux, appelé base de *moduli*. Le Théorème des restes Chinois (CRT) nous assure qu'il est possible de représenter tous les entiers positifs inférieurs au produit des éléments de la base des moduli. La version constructive de la preuve du CRT nous donne aussi un algorithme pour convertir un entier représenté en RNS, vers un système du numération de position classique (décimal, binaire). Un des avantages algorithmiques du RNS vient du fait que les opérations (additions, soustractions, multiplications) sur de grands entiers peuvent être effectués en parallèle sur des opérandes beaucoup plus petits (de la taille des éléments de la base des moduli).

Dans un premier temps, j'ai travaillé avec J.-C. Bajard sur la définition de cellules arithmétiques de base nécessaires à l'implantation d'un système cryptographique purement RNS, c'est-à-dire sans conversion depuis et vers le système binaire classique. Une fois découpé en blocs, le message à chiffrer ou à signer peut en effet être vu comme la représentation RNS d'un grand nombre. Cette approche permet de bénéficier des avantages du RNS tout en évitant les étapes coûteuses de conversions, depuis et vers un système classique. Nous avons étendu à l'exponentiation modulaire un algorithme de multiplication modulaire en RNS, initialement proposé par Bajard et al. [3, 4], et proposé une implantation purement RNS de l'algorithme RSA [5]. À notre connaissance, il s'agit du premier algorithme où le RNS est utilisé sans conversion. Nous avons prouvé qu'une telle solution était possible et ne compromettait en rien la sécurité du protocole cryptographique.

Cette première étude a donné lieu à une collaboration industrielle avec P.-Y. Liardet et Y. Teglia de la société ST Microelectronics sur l'utilisation du RNS pour la résistance aux attaques de type canaux cachés (*side channel attacks, SCA*) [57, 58]. Ce type d'attaques consiste à analyser diverses traces (courant, temps de calcul, radiations électromagnétiques, ...) diffusées par un dispositif électronique (carte à puce, téléphone portable, assistant personnel, ...) au cours de l'exécution d'un algorithme de chiffrement/signature. Il existe plusieurs familles de SCA : les

attaques simples qui observent une seule exécution de l'algorithme et les attaques différentielles qui analysent plusieurs traces et tentent de dévoiler un secret grâce à des outils statistiques. Une troisième classe d'attaques est basée sur l'injection de fautes [21]. Il existe plusieurs méthodes pour contrer ces différentes attaques ; en particulier la randomisation des données (message et exposant) et la randomisation des calculs. Dans [8], nous avons proposé une version randomisée de notre algorithme d'exponentiation modulaire en RNS, dont l'idée principale consiste à tirer aléatoirement les éléments de la base des moduli, parmi un ensemble prédéfini de nombres premiers entre eux, de telle sorte que l'attaquant ne puisse pas exploiter les traces émises par le dispositif. Notre algorithme permet de choisir la base de manière aléatoire au début et en cours de calcul (i.e. pendant l'exponentiation modulaire). Si l'ensemble prédéfini pour les moduli est suffisamment grand, le produit M des éléments de la base ainsi générée peut être considéré comme une valeur aléatoire. La première étape du calcul, qui consiste à calculer $xM \bmod n$ pour convertir x dans la représentation dite de Montgomery, a donc aussi pour effet de randomiser le message x. La randomisation des calculs est obtenue grâce à la nature parallèle du RNS, en faisant en sorte que les calculs effectués par une cellule arithmétique de base varient au cours d'une même exponentiation.

Arithmétique tolérante aux fautes

Lors d'un séjour d'un mois à l'université de Calgary en mars 2002, j'ai proposé un système de représentation et un algorithme permettant de détecter/corriger des erreurs de calculs pouvant intervenir dans l'évaluation de produits scalaires, très utilisés en traitement du signal. Cette approche considère les nombres comme des polynômes à plusieurs variables à coefficients sur des anneaux d'entiers (ou corps) finis [50].

Bibliographie

[1] J.-P. Allouche and J. Shallit. *Automatic Sequences*. Cambridge University Press, 2003.

[2] R. Avanzi. A note on the signed sliding window integer recoding and a left-to-right analogue. In *Selected Areas in Cryptography, SAC 2004*, volume 3357 of *Lecture Notes in Computer Science*, pages 130–143. Springer, 2005.

[3] J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 46(7) :766–776, July 1998.

[4] J.-C. Bajard, L.-S. Didier, and P. Kornerup. Modular multiplication and base extension in residue number systems. In *Proceedings 15th IEEE symposium on Computer Arithmetic, ARITH15*, pages 59–65, 2001.

[5] J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 53(6) :769–774, June 2004.

[6] J.-C. Bajard, L. Imbert, and G. A. Jullien. Parallel Montgomery multiplication in GF(2^k) using trinomial residue arithmetic. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic, ARITH17*, pages 164–171. IEEE Computer Society, 2005.

[7] J.-C. Bajard, L. Imbert, G. A. Jullien, and H. C. Williams. A CRT-based Montgomery multiplication for finite fields of small characteristic. In *Proceedings 17th IMACS World Congress, Scientific Computation, Applied Mathematics and Simulation*, pages 101–107, 2005.

[8] J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Cryptographic Hardware and Embedded Systems, CHES'04*, volume 3156 of *Lecture Notes in Computer Science*, pages 62–75. Springer, 2004.

[9] J.-C. Bajard, L. Imbert, and C. Negre. Modular multiplication in GF(p^k) using Lagrange representation. In *Progress in Cryptology, INDOCRYPT'02*, number 2551 in Lecture Notes in Computer Science, pages 275–284. Springer, 2002.

[10] J.-C. Bajard, L. Imbert, and C. Negre. Arithmetic operations in finite fields of medium prime characteristic using the Lagrange representation. *IEEE Transactions on Computers*, 55(9) :1167–1177, Sept. 2006.

[11] J.-C. Bajard, L. Imbert, C. Negre, and T. Plantard. Multiplication in GF(p^k) for elliptic curve cryptography. In *Proceedings of the 16th IEEE Symposium on Computer Arithmetic, ARITH16*, pages 181–187. IEEE Computer Society, 2003.

[12] J.-C. Bajard, L. Imbert, and T. Plantard. Modular number systems : Beyond the Mersenne family. In *Proceedings of the 11th International Workshop on Selected Areas in Cryptography, SAC'04*, volume 3357 of *Lecture Notes in Computer Science*, pages 159–169. Springer, 2005.

[13] P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In *Advances in Cryptology, CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 1986.

[14] D. J. Bernstein. Curve25519 : New Diffie-Hellman speed records. In *Proceedings of Public Key Cryptography, PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006.

[15] D. J. Bernstein and T. Lange. Faster addition and doubling on elliptic curves. In *Advances in cryptology, ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.

[16] V. Berthé. Autour du système de numération d'Ostrowski. *Bulletin of the Belgian Mathematical Society*, 8 :209–239, 2001.

[17] V. Berthé and L. Imbert. On converting numbers to the double-base number system. In *Advanced Signal Processing Algorithms, Architecture and Implementations XIV*, volume 5559 of *Proceedings of SPIE*, pages 70–78. SPIE, 2004.

[18] V. Berthé and L. Imbert. Writing integers in two bases. Research report, LIRMM, CNRS, Univ. Montpellier 2, Jan. 2008.

[19] I. F. Blake, G. Seroussi, and N. P. Smart. *Elliptic Curves in Cryptography*. Number 256 in London Mathematical Society Lecture Note Series. Cambridge University Press, 1999.

[20] I. F. Blake, G. Seroussi, and N. P. Smart. *Advances in Elliptic Curve Cryptography*. Number 317 in London Mathematical Society Lecture Note Series. Cambridge University Press, 2005.

[21] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology, EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer, 1997.

[22] A. D. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(2) :236–240, 1951. Reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[23] W. Bosma. Signed bits and fast exponentiation. *Journal de théorie des nombres de Bordeaux*, 13 :27–41, 2001.

[24] W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. the user language. *Journal of Symbolic Computation*, 24(3-4) :235–265, 1997. `http://magma.maths.usyd.edu.au/magma/`.

[25] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3) :26–33, June 1996.

[26] J. Chung and A. Hasan. More generalized Mersenne numbers. In *Selected Areas in Cryptography, SAC'03*, volume 3006 of *Lecture Notes in Computer Science*, pages 335–347. Springer, 2004.

[27] H. Cohen. Analysis of the flexible window powering algorithm. *Journal of Cryptology*, 18(1) :63–76, 2005.

[28] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.

[29] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation. In *Information and Commuunication Security, ICICS 1997*, volume 1334 of *Lecture Notes in Computer Science*, pages 282–290. Springer, 1997.

[30] R. Crandall. Method and apparatus for public key exchange in a cryptographic system. US Patent 5159632, 1992.

[31] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6) :644–654, Nov. 1976.

[32] V. Dimitrov and T. V. Cooklev. Hybrid algorithm for the computation of the matrix polynomial $I+A+\cdots+A^{N-1}$. *IEEE Transactions on Circuits and Systems I : Fundamental Theory and Applications*, 42(7) :377–380, July 1995.

[33] V. Dimitrov, J. Eskritt, L. Imbert, G. A. Jullien, and W. C. Miller. The use of the multi-dimensional logarithmic number system in DSP applications. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, ARITH15*, pages 247–254. IEEE Computer Society, 2001.

[34] V. Dimitrov, L. Imbert, and P. K. Mishra. Efficient and secure elliptic curve point multiplication using double-base chains. In *Advances in Cryptology, ASIACRYPT'05*, volume 3788 of *Lecture Notes in Computer Science*, pages 59–78. Springer, 2005.

[35] V. Dimitrov, L. Imbert, and P. K. Mishra. The double-base number system and its application to elliptic curve cryptography. *Mathematics of Computation*, 77(262) :1075–1104, 2008.

[36] V. Dimitrov, L. Imbert, and A. Zakaluzny. Sublinear constant multiplication algorithms. In *Advanced Signal Processing Algorithms, Architectures and Implementations XVI*, volume 6313 of *Proceedings of SPIE*, page 631305. SPIE, 2006.

[37] V. Dimitrov, L. Imbert, and A. Zakaluzny. Multiplication by a constant is sublinear. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic*, pages 261–268. IEEE Computer Society, 2007.

[38] V. Dimitrov and G. A. Jullien. Loading the bases : A new number representation with applications. *IEEE Circuits and Systems Magazine*, 3(2) :6–23, 2003.

[39] V. Dimitrov, G. A. Jullien, and W. C. Miller. An algorithm for modular exponentiation. *Information Processing Letters*, 66(3) :155–159, May 1998.

[40] V. Dimitrov, G. A. Jullien, and W. C. Miller. Theory and applications of the double-base number system. *IEEE Transactions on Computers*, 48(10) :1098–1106, Oct. 1999.

[41] C. Doche, T. Icart, and D. R. Kohel. Efficient scalar multiplication by isogeny decompositions. In *Public Key Cryptography, PKC'06*, volume 3958 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2006.

[42] C. Doche and L. Imbert. Extended double-base number system with applications to elliptic curve cryptography. In *Progress in Cryptology, INDOCRYPT'06*, volume 4329 of *Lecture Notes in Computer Science*, pages 335–348. Springer, 2006.

[43] H. M. Edwards. A normal form for elliptic curves. *Bulletin of the American Mathematical Society*, 44(3) :393–422, 2007.

[44] M. D. Ercegovac, L. Imbert, D. W. Matula, J.-M. Muller, and G. Wei. Improving Goldschmidt division, square root and square root reciprocal. *IEEE Transactions on Computers*, 49(7) :759–763, July 2000.

[45] M. D. Ercegovac and T. Lang. *Division and Square Root : Digit-Recurrence Algorithms and Implementations*. Kluwer Academic Publishers, Boston, 1994.

[46] M. D. Ercegovad and T. Lang. *Digital Arithmetic*. Morgan Kaufmann, 2003.

[47] C. V. Freiman. Statistical analysis of certain binary division algorithms. In *IRE Proceedings*, volume 49, pages 91–103, 1961.

[48] R. E. Goldschmidt. *Applications of division by convergence*. PhD thesis, Massachusetts Institute of Technology, June 1964.

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

[50] L. Imbert, V. Dimitrov, and G. A. Jullien. Fault-tolerant computations over replicated finite rings. *IEEE Transactions on Circuits and Systems I : Fundamental Theory and Applications*, 50(7) :858–864, July 2003.

[51] L. Imbert, A. Pereira, and A. Tisserand. A library for prototyping the computer arithmetic level in elliptic curve cryptography. In *Advanced Signal Processing Algorithms, Architectures, and Implementations XVII*, volume 6697 of *Proceedings of SPIE*, page 66970N. SPIE Ed., 2007.

[52] M. Joye and S.-M. Yen. Optimal left-to-right binary signed-digit exponent recoding. *IEEE Transactions on Computers*, 49(7) :740–748, 2000.

[53] A. Y. Khinchin. *Continued Fractions*. Dover Publications, new edition, 1997.

[54] D. E. Knuth. *The Art of Computer Programming, Vol. 2 : Seminumerical Algorithms*. Addison-Wesley, Reading, MA, third edition, 1997.

[55] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177) :203–209, Jan. 1987.

[56] N. Koblitz. CM curves with good cryptographic properties. In *Advances in Cryptography, CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 279–287. Springer, 1992.

[57] P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In *Advances in Cryptology, CRYPTO'99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[58] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology, CRYPTO'96*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.

[59] A. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography*. CRC Press, 1997.

[60] V. S. Miller. Uses of elliptic curves in cryptography. In *Advances in Cryptology, CRYPTO'85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–428. Springer, 1986.

[61] P. L. Montgomery. Evaluating recurrences of form $x_{m+n} = f(x_m, x_n, x_{m-n})$ via Lucas chains, Dec. 1983. Available at `ftp.cwi.nl:/pub/pmontgom/Lucas.ps.gz`.

[62] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170) :519–521, Apr. 1985.

[63] P. L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177) :243–264, Jan. 1987.

[64] J. A. Muir and D. R. Stinson. New minimal weight representations for left-to-right window methods. In *Topics in cryptology, CT-RSA 2005*, volume 3376 of *Lecture Notes in Computer Science*, pages 366–383. Springer, 2005.

[65] J.-M. Muller. *Arithmétique des ordinateurs*. Masson, 1989.

[66] J.-M. Muller. Algorithmes de division pour microprocesseurs : illustration à l'aide du "bug" du Pentium. *Technique et Science Informatiques*, 14(8), Oct. 1995.

[67] National Institute of Standards and Technology. *FIPS PUB 186-2 : Digital Signature Standard (DSS)*. National Institute of Standards and Technology, Jan. 2000.

[68] T. Plantard. *Arithmétique Modulaire pour la Cryptographie*. PhD thesis, Université Montpellier 2, 2005.

[69] G. W. Reitwiesner. Binary arithmetic. *Advances in Computers*, 1 :231–308, 1960.

[70] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2) :120–126, Feb. 1978.

[71] J. E. Robertson. A new class of digital division methods. *IRE Transactions on electronic computers*, C-7, Sept. 1958.

[72] R. Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of Computation*, 44 :483–494, 1985.

[73] R. Schoof. Counting points on elliptic curves over finite fields. *Journal de théorie des nombres de Bordeaux*, 7 :219–254, 1995.

[74] J. Solinas. Generalized mersenne numbers. Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 1999.

[75] J. A. Solinas. Improved algorithms for arithmetic on anomalous binary curves. Research Report CORR-99-46, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 1999. Updated version of the paper appearing in the proceedings of CRYPTO'97.

[76] The SECG Group. SEC 1 : Elliptic curve cryptography. Standard for Efficient Cryptography, Sept. 2000. `http://www.secg.org/`.

[77] The SECG group. SEC 2 : Recommended elliptic curve domain parameters. Standard for Efficient Cryptography, Sept. 2000. `http://www.secg.org/`.

[78] N. Theriault. SPA resistant left-to-right integer recodings. In *Selected areas in cryptography, SAC 2005*, volume 3897 of *Lecture Notes in Computer Science*, pages 345–358. Springer, 2006.

[79] R. Tijdeman. On the maximal distance between integers composed of small primes. *Compositio Mathematica*, 28 :159–162, 1974.

[80] L. C. Washington. *Elliptic Curves : Number Theory and Cryptography.* Chapman & Hall/CRC, 2003.

Modular Number Systems: Beyond the Mersenne Family

Jean-Claude Bajard[1], Laurent Imbert[1,2], and Thomas Plantard[1]

[1] LIRMM, CNRS UMR 5506,
161 rue Ada, 34392 Montpellier cedex 5, France
[2] ATIPS, CISaC, University of Calgary,
2500 University drive N.W, Calgary, T2N 1C2, Canada
{bajard, plantard, Laurent.Imbert}@lirmm.fr

Abstract. In SAC 2003, J. Chung and A. Hasan introduced a new class of specific moduli for cryptography, called the more generalized Mersenne numbers, in reference to J. Solinas' generalized Mersenne numbers proposed in 1999. This paper pursues the quest. The main idea is a new representation, called Modular Number System (MNS), which allows efficient implementation of the modular arithmetic operations required in cryptography. We propose a modular multiplication which only requires n^2 multiplications and $3(2n^2 - n + 1)$ additions, where n is the size (in words) of the operands. Our solution is thus more efficient than Montgomery for a very large class of numbers that do not belong to the large Mersenne family.

Keywords: Generalized Mersenne numbers, Montgomery multiplication, Elliptic curve cryptography

1 Introduction

Efficient implementation of modular arithmetic is an important prerequisite in today's public-key cryptography [6]. In the case of elliptic curves defined over prime fields, operations are performed modulo prime numbers whose size range from 160 to 500 bits [4].

For moduli p that are not of special form, Montgomery [7] or Barrett [1] algorithms are widely used. However, modular multiplication and reduction can be accelerated considerably when the modulus p has a special form. Mersenne numbers of the form $2^m - 1$ are well known examples, but they are not useful for cryptography because there are only a few primes (the first Mersenne primes are $3, 7, 31, 127, 8191, 131071, 524287, 2147483647$, etc). Pseudo-Mersenne of the form $2^m - c$, introduced by R. Crandall in [3], allow for very efficient modular reduction if c is a small integer. In 1999, J. Solinas [8] introduced the family of generalized Mersenne numbers. They are expressed as $p = f(t)$, where f is a well chosen monic integral polynomial and t is a power of 2, and lead to very fast modular reduction using only a few number of additions and subtractions.

H. Handschuh and A. Hasan (Eds.): SAC 2004, LNCS 3357, pp. 159–169, 2005.

For example, the five NIST primes listed below, recommended in the FIPS 186-2 standard for defining elliptic curves over primes fields, belong to this class[1].

$$\begin{aligned} p_{192} &= 2^{192} - 2^{64} - 1 \\ p_{224} &= 2^{224} - 2^{96} + 1 \\ p_{256} &= 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1 \\ p_{384} &= 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1 \\ p_{521} &= 2^{521} - 1 \end{aligned}$$

In 2003, J. Chung and A. Hasan, in a paper entitled "more generalized Mersenne numbers" [2], extended J. Solinas' concept, by allowing any integer for t.

In this paper we further extend the idea of defining new classes of numbers (possibly prime), suitable for cryptography. However, the resemblance with the previous works ends here. Instead of considering moduli of special form, we represent the integers modulo p in the so-called Modular Number System (MNS). By a careful choice of the parameters which define our MNS, we introduce the concept of Adapted Modular Number System (AMNS). We propose a modular multiplication which is more efficient than Montgomery's algorithm, and we explain how to define suitable prime moduli for cryptography. We provide examples of such numbers at the end of the paper.

2 Modular Number Systems

In positional number systems, we represent any nonnegative integer X in base β as

$$X = \sum_{i=0}^{k-1} d_i \, \beta^i, \tag{1}$$

where the digits d_is belong to the set $\{0, \ldots, \beta - 1\}$. If $d_{k-1} \neq 0$, we call X a k-digit base-β number.

In cryptographic applications, computations have to be done over finite rings or fields. In these cases, we manipulate representatives of equivalence classes modulo P (for simplicity we use the set of positive integers $\{0, 1, 2, \ldots, P-1\}$), and the operations are performed modulo P.

In the next definition, we extend the notion of positional number system to represent the integers modulo P.

Definition 1 (MNS). *A Modular Number System (MNS) $\mathcal{B}$ is defined according to four parameters (γ, ρ, n, P), such that all positive integers $0 \leq X < P$ can be written as*

$$X = \sum_{i=0}^{n-1} x_i \, \gamma^i \bmod P, \tag{2}$$

[1] Note that p_{521} is also a Mersenne prime.

with $1 < \gamma < P$, *and* $x_i \in \{0, \ldots, \rho - 1\}$. *The vector* $(x_0, x_1, \ldots, x_{n-1})_{\mathcal{B}}$ *denotes the representation of* X *in* $\mathcal{B}$.

In the sequel of the paper, we shall omit the subscript $(.)_{\mathcal{B}}$ when it is clear from the context, and we shall consider X either as a vector or as a polynomial (in γ). In the later, the x_is correspond to the coefficients of the polynomial (note that we use a left-to-right notation; x_0 is the constant term).

Example 1. Let us consider the MNS defined with $\gamma = 7, \rho = 3, n = 3, P = 17$. Over this system, we represent the elements of $\mathbb{Z}_{17}$ as polynomials in γ of degree at most 3 with coefficients in $\{0, 1, 2\}$ (cf. Table 1).

Table 1. The elements of $\mathbb{Z}_{17}$ in $\mathcal{B} = MNS(7, 3, 3, 17)$

0	1	2	3	4	5
0	1	2	$\gamma + 2\gamma^2$	$1 + \gamma + 2\gamma^2$	$\gamma + \gamma^2$

6	7	8	9	10	11
$1 + \gamma + \gamma^2$	γ	$1 + \gamma$	$2 + \gamma$	$2\gamma + 2\gamma^2$	$1 + 2\gamma + 2\gamma^2$

12	13	14	15	16	
$2\gamma + \gamma^2$	$1 + 2\gamma + \gamma^2$	2γ	$1 + 2\gamma$	$2 + 2\gamma$	

We remark that this system is redundant. For example, we can write $5 = 2 + \gamma^3 = \gamma + \gamma^2$, or $14 = 1 + 2\gamma^2 = 2\gamma$. However, we do not take any advantage of this property in this paper.

Definition 2 (AMNS). *A modular number system* $\mathcal{B} = MNS(\gamma, \rho, n, P)$ *is called Adapted Modular Number System (AMNS) if* $\gamma^n \bmod P = c$ *is a small integer. In this case we shall denote* $\mathcal{B} = AMNS(\gamma, \rho, n, P, c)$.

Although c is given by $\gamma^n \bmod P$, we introduce it in the AMNS definition to simplify the notations.

Note that it is not obvious (see Section 5) to prove that a given set of parameters (γ, ρ, n, P) is an MNS. Algorithm 3, presented in the next section, gives sufficient conditions to prove that this is an AMNS.

In the rest of the paper, we shall consider $\mathcal{B} = AMNS(\gamma, \rho, n, P, c)$, unless otherwise specified.

3 Modular Multiplication

As in [2], modular multiplication is performed in three steps presented in Algorithm 1.

In order to evaluate the computational complexity of Algorithm 1, let us get into more details. In the first step we evaluate

$$U(X) = \sum_{i=0}^{2n-2} u_i X^i, \text{ where } u_i = \sum_{j=0}^{i} a_i b_{i-j}, \tag{3}$$

Algorithm 1 – Modular Multiplication

Input : An AMNS $\mathcal{B} = (\gamma, \rho, n, P, c)$, and $A = (a_0, ..., a_{n-1})$, $B = (b_0, ..., b_{n-1})$
Output : $S = (s_0, ..., s_{n-1})$ such that $S = A\,B \bmod P$
1: Polynomial multiplication in $\mathbb{Z}[X]$: $U(X) \leftarrow A(X)\,B(X)$
2: Polynomial reduction: $V(X) \leftarrow U(X) \bmod (X^n - c)$
3: Coefficient reduction: $S \leftarrow CR(V)$, gives $S \equiv V(\gamma) \pmod P$

where $a_t = b_t = 0$ for $t > n-1$. We have $u_0 = a_0 b_0 < \rho^2$, $u_1 = a_0 b_1 + a_1 b_0 < 2\rho^2$, etc. Clearly, the largest coefficient is $u_{n-1} < n\rho^2$. Then, for the coefficients of degree greater than $n-1$, we have $u_n < (n-1)\rho^2, \ldots, u_{2n-2} < \rho^2$.

The cost of the first step clearly depends on the size of ρ and n. It requires n^2 products of size $\log_2(\rho)$, and $(n-1)^2$ additions of size at most $\log_2(n\rho^2)$.

In step 2, we compute

$$V(X) = \sum_{i=0}^{n-1} v_i\, X^i, \text{ where } v_i = u_i + c\, u_{i+n}. \tag{4}$$

This yields

$$v_i < cn\rho^2, \quad \text{for } i = 0 \ldots n-1. \tag{5}$$

The cost of step 2 is $(n-1)$ products between the constant c and numbers of size $\log_2(n\rho^2)$, and $(n-1)$ additions of size $\log_2(cn\rho^2)$. When c is a small constant, for example a power of 2, the $(n-1)$ products can be implemented with only $(n-1)$ shifts and additions.

In order to get a valid AMNS representation we must reduce the coefficients such that all the v_is are less than ρ. This is the purpose of the coefficient reduction.

3.1 Coefficient Reduction

For simplicity, we define $\rho = 2^{k+1}$. We reduce the elements of the vector V, obtained after step 2 of Algorithm 1, by iteratively applying Algorithm 2, presented below, which reduces numbers of size $\lceil \frac{3k}{2} \rceil$ bits to numbers of size $k+1$, i.e. less than ρ.

So, let us first consider a vector V with elements of size at most $\lceil \frac{3k}{2} \rceil$ bits. Our goal is to find a representation of V where the elements are less than ρ, i.e. of size at most $k+1$ bits.

If $V = (v_0, \ldots, v_{n-1})$, we define two vectors $\underline{V}$ and $\overline{V}$ such that

$$V = \underline{V} + \overline{V} \cdot 2^k I, \tag{6}$$

where the elements of $\underline{V}$ are less than 2^k and those of $\overline{V}$ are less than $2^{\lceil k/2 \rceil}$. In equation (6), I denotes the $n \times n$ identity matrix explicitly given by $I_{ij} = \delta_{ij}$ for $i, j = 0, \ldots, n-1$ and δ_{ij} is the Kronecker delta.

If we can express $\overline{V} \cdot 2^k I$ as a vector with elements less than 2^k, then the sum of the two vectors in (6) is less than 2^{k+1}, and gives a valid AMNS representation of V. The idea is to find a matrix M with small coefficients which satisfies

$$M \cdot (1, \gamma, \ldots, \gamma^{n-1})^T \equiv 2^k I \cdot (1, \gamma, \ldots, \gamma^{n-1})^T \pmod{P}. \tag{7}$$

Roughly speaking, the matrix M can be seen as a representation of $2^k I$ in the AMNS.

If $2^k = (\xi_0, \ldots, \xi_{n-1})_{\mathcal{B}}$, is a representation of 2^k in the AMNS, then by definition 1 we have

$$2^k \equiv \xi_0 + \xi_1\gamma + \cdots + \xi_{n-1}\gamma^{n-1} \pmod{P}. \tag{8}$$

Similarly the following congruences hold:

$$\gamma\, 2^k \equiv c\xi_{n-1} + \xi_0\gamma + \cdots + \xi_{n-2}\gamma^{n-1} \pmod{P} \tag{9}$$

$$\gamma^2\, 2^k \equiv c\xi_{n-2} + c\xi_{n-1}\gamma + \xi_0\gamma^2 + \cdots + \xi_{n-3}\gamma^{n-1} \pmod{P} \tag{10}$$

$$\vdots$$

$$\gamma^{n-1}\, 2^k \equiv c\xi_1 + c\xi_2\gamma + \cdots + c\xi_{n-1}\gamma^{n-2} + \xi_0\gamma^{n-1} \pmod{P}. \tag{11}$$

Equations (8) to (11) allow us to define the matrix

$$M = \begin{pmatrix} \xi_0 & \xi_1 & \cdots & \xi_{n-1} \\ c\xi_{n-1} & \xi_0 & \cdots & \xi_{n-2} \\ \vdots & & & \\ c\xi_1 & c\xi_2 & \cdots\, c\xi_{n-1} & \xi_0 \end{pmatrix} \tag{12}$$

which satisfies equation (7). Thus $\overline{V} \cdot 2^k I \equiv \overline{V} \cdot M \pmod{P}$, and equation (6) becomes

$$V = \underline{V} + \overline{V} \cdot M. \tag{13}$$

If we impose $c\sum_{i=0}^{n-1} \xi_i < 2^{\lfloor k/2 \rfloor}$, then the elements of the vector $\overline{V} \cdot M$ are less than 2^k. Algorithm 2 implements equation (13) to reduce the elements of V to a valid AMNS representation, i.e. with $v_i < \rho = 2^{k+1}$ for $i = 0 \ldots n-1$.

Algorithm 2 – Red$(V, \mathcal{B})$: reduction from $\lceil \frac{3k}{2} \rceil$ to $k+1$ bits

Input : $\mathcal{B} = (\gamma, \rho, n, P, c)$ an AMNS with $\rho = 2^{k+1}$; $2^k = (\xi_0, ..., \xi_{n-1})$ with $c\sum_{i=0}^{n-1} \xi_i < 2^{\lfloor k/2 \rfloor}$; a matrix M as defined in (12); a vector $V = (v_0, \ldots, v_{n-1})$ with $v_i < 2^{\lceil 3k/2 \rceil}$ for $i = 0 \ldots n-1$.

Output : $S = (s_0, ..., s_{n-1})$ with $s_i < \rho$ for all $i = 0 \ldots n-1$.

1: Define vectors $\underline{V}$ and $\overline{V}$ such that $V = \underline{V} + \overline{V} \cdot 2^k I$

2: Compute $S \leftarrow \underline{V} + \overline{V} \cdot M$

The cost of Algorithm 2 is n^2 multiplications of size $\frac{k}{2}$ and n additions of size k. However, since the M_{ij} in (12) are small constants, the n^2 products can be efficiently computed with only a small number of additions and shifts. For

example, if c and the ξ_is are small powers of 2, then we can evaluate $\overline{V} \cdot M$ with $n(n-1)$ additions. In this case the total cost for Red is n^2 additions of size k.

In order to reduce polynomials with coefficients larger than $\frac{3k}{2}$ bits, we iteratively apply the previous algorithm until all the coefficients are less than ρ. The following theorem holds.

Theorem 1. *Let us define $\mathcal{B} = AMNS(\gamma, \rho, n, P, c)$, with $\rho = 2^{k+1}$. We denote $(\xi_0, \ldots, \xi_{n-1})$ a representation of 2^k in $\mathcal{B}$, and we assume $V = (v_0, \ldots, v_{n-1})$ with $v_i < cn\rho^2$.*

If $c\sum_{i=0}^{n-1} \xi_i < 2^{\lfloor k/2 \rfloor}$ then there exists an algorithm which reduces V into a valid AMNS representation, in $\frac{k+2+\lfloor \log_2(cn) \rfloor}{\lceil k/2 \rceil - 1}$ calls to Red (algorithm 2).

Proof. After step 2 of Algorithm 1, and under the condition $\rho = 2^{k+1}$, the elements of V satisfy $v_i < 2^{2k+2}cn$. Thus $|v_i| < 2k + 3 + \lfloor \log_2(cn) \rfloor$, where $|v_i|$ denotes the size of v_i. Since each step of Red eliminates $\lceil \frac{k}{2} \rceil - 1$ bits of v_i, the number of iteration is given by the value t which satisfy the equation $2k + 3 + \lfloor \log_2(cn) \rfloor - t\left(\lceil \frac{k}{2} \rceil - 1\right) = k + 1$, i.e. $t = \frac{k+2+\lfloor \log_2(cn) \rfloor}{\lceil k/2 \rceil - 1}$. This gives $t = 2 + \frac{8+2\lfloor \log_2(cn) \rfloor}{k-2}$ is k is even, and $t = 2 + \frac{6+2\lfloor \log_2(cn) \rfloor}{k-1}$ if k is odd. □

Note that in practice, the number of iterations is very small. In the examples of section 5, the coefficient reduction step only requires 3 or 4 calls to algorithm Red. Algorithm 3 implements theorem 1.

Algorithm 3 – CR$(V, \mathcal{B})$, Coefficient reduction

Input : $\mathcal{B} = (\gamma, \rho, n, P, c)$ an AMNS with $\rho = 2^{k+1}$; $2^k = (\xi_0, ..., \xi_{n-1})$ with $c\sum_{i=0}^{n-1} \xi_i < 2^{\lfloor k/2 \rfloor}$; a vector $V = (v_0, ..., v_{n-1})$.
Output : $S = (s_0, ..., s_{n-1})$ with $s_i < \rho$ for all $i = 0 \ldots n-1$.

```
l ← max(⌊log2(v_i)⌋ + 1)
U ← V
while l > 3k/2 do
    Define U_ and U_over s.t. U = U_ + 2^{l-3k/2} · U_over
    U_over ← Red(U_over, B)
    U ← U_ + 2^{l-3k/2} · U_over
    l ← max(⌊log2(U_i)⌋ + 1)
end while
S ← Red(U, B)
```

4 Complexity Comparisons

In this section, we evaluate the number of elementary operations (word-length multiplications and additions) of our modular multiplication algorithm. Since the complexity of our algorithms clearly depends on many parameters we try to consider different interesting options. For simplicity, we assume $cn < \rho$ as this is the case in the examples presented in the next section.

For a large part, the moduli (possibly primes) we are able to generate do not belong neither to Solinas' [8] or Chung and Hasan's generalized Mersenne family [2]. Thus, we only compare our algorithm with Montgomery since it was the best known algorithm available for those numbers.

As explained in section 3, our modular multiplication requires three steps: polynomial multiplication, polynomial reduction, and coefficient reduction.

The polynomial multiplication only depends on n and ρ. If $\rho = 2^{k+1}$ ($k+1$ is the word-size), the cost of the first step is $n^2\,T_m$, where T_m is the delay of one word-length multiplication, and $(n-1)^2$ additions involving two-word operands, i.e. of cost less than $3(n-1)^2\,T_a$, where T_a is the delay of one word-length addition. Thus, the cost of step 1 is

$$n^2\,T_m + 3(n-1)^2\,T_a.$$

The polynomial reduction depends on n, ρ, and c. In the general case, it requires $(n-1)$ multiplications of size $\log_2(n\rho^2)$. However, if $c = 1, 2, 4$ (resp. $c = 3, 5, 6$) it can be implemented in n shift-and-add of three-word numbers (resp. $2n$ shift-and-add). This yields a cost of $3n\,T_a$ (resp. $6n\,T_a$). Thus, for the second step, a careful choice of c can lead to

$$3n\,T_a.$$

The coefficient reduction depends on all the parameters. The cost of algorithm Red, for $\xi_i = 0, 1, 2$ and $c = 1, 2, 4$ is $n^2\,T_a$ (it becomes $n^2 + \frac{(n-1)^2}{2}\,T_a$ if $c = 3, 5, 6$). From theorem 1, algorithm CR requires 3 calls to Red if $cn < 2^{(k-10)/2}$ (4 calls if $cn < 2^{k-10}$). Finally, step 3 requires $3n^2\,T_a$ if $cn < 2^{(k-10)/2}$, $\xi_i = 0, 1, 2$, and $c = 1, 2, 4$ (we have $8n^2\,T_a$ if $cn < 2^{k-10}$, $\xi_i = 0, 1, 2$, $c = 3, 5, 6$). As for the previous step, a good choice of c and the ξ_is gives a complexity of

$$3n^2\,T_a.$$

The important point here is that we can perform the coefficient reduction without multiplications.

To summarize, our algorithm performs the modular multiplication, where the moduli do not belong to the generalized Mersenne families – introduced by Solinas, and Chung and Hasan – in

$$n^2\,T_m + 3\left(2n^2 - n + 1\right)\,T_a.$$

This is better than Montgomery which requires $2n^2\,T_m$ (cf. [7], [5]).

In the next section we explain how we define such modulus and we give examples that reach this complexity.

5 Construction of Suitable Moduli

In this section we explain how to find γ and P which allow fast modular arithmetic.

Let us first fix some of the parameters. Since we represent numbers as polynomials of coefficients less than $\rho = 2^{k+1}$, it is advantageous to define ρ according to the word-size of the targeted architecture, i.e. by taking $k = 15, 31, 63$ for 16-bit, 32-bit, and 64-bit architectures respectively. We define n such that $(k+1)n$ roughly corresponds to the desired dynamic range. To get a very efficient reduction of the coefficients, we impose restrictions on the ξ_is, for example by only allowing values in $\{0, 1, 2\}$, and we choose very small values for c. Based on the previous choices, we now try to find suitable P and γ.

From equation (7), we deduce $V \cdot (2^k I - M) \equiv 0 \pmod{P}$, for all $V = (v_0, \ldots, v_{n-1})_{\mathcal{B}}$. Thus, it is clear that the determinant

$$d = \left|2^k I - M\right| \equiv 0 \pmod{P}. \tag{14}$$

All the divisors of d, including d itself, can be chosen for P. If we need P to be prime, we can either try to find a prime factor of the determinant which is large enough (this is easier than factorization since it suffices to eliminate the small prime factors up to an arbitrary bound), or consider only the cases where the determinant is already a prime.

We remark that γ is a root, modulo P, of both $\gamma^n - c$ and $2^k - \sum_{i=0}^{n-1} \xi_i \gamma^i$. Thus γ is also a root of $\gcd(\gamma^n - c, 2^k - \sum_{i=0}^{n-1} \xi_i \gamma^i) \bmod P$.

5.1 Generating Primes for Cryptographic Applications

For elliptic curve defined over prime fields, P must be a prime of size at least 160 bits.

Let us assume a 16-bit architecture. We fix $\rho = 2^{16}$, and we see if we can generate good primes P with $n = 11$. Note that $nk = 176$ does not guaranty 176-bit primes for P. In practice, the candidates we obtain are slightly smaller. We impose strong restrictions on the other parameters, by allowing only $\xi_i \in \{0, 1\}$, and $2 \leq c \leq 6$.

As an example, we consider $c = 3$, and $2^k = (1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1)_{\mathcal{B}}$, which correspond to the polynomial $1 + x^5 + x^8 + x^9 + x^{10}$. Using (14), we compute

$$d = 467523550650744744856027134573563377107161910767327,$$

which has

$$P = 7924127977131266861966561602941752154264730638 53$$

as a prime factor of size 160 bits.

Then, we compute a root of $\gcd(x^{11} - 3, 2^{15} - 1 - x^5 - x^8 - x^9 - x^{10})$ modulo P, and we obtain

$$\gamma = 474796736496801627149092588633773724051936841406.$$

We have investigated different set of parameters and applied the same technique to define suitable prime moduli. In Table 2, we give the number of such primes and the corresponding parameters.

Table 2. Number of primes P greater than 2^{160} for use in elliptic curve cryptography, and the corresponding AMNS parameters

$k+1$	n	c	ξ_i	Number of primes of size ≥ 160 bits
16	11	$\{2,3\}$	$\{0,1\}$	132
16	11	$\{2,3,4,5,6\}$	$\{0,1\}$	306
16	11	2	$\{0,1,2\}$	3106
16	11	$\{2,3,4,5,6\}$	$\{0,1,2\}$	≥ 7416 (a)
32	6	$\{2,3,4,5,6\}$	$\{0,1\}$	≥ 12 (b)
32	6	$\{2,3,4,5,6\}$	$\{0,1,2\}$	≥ 87 (b)
64	16	$\{2,3,4,5,6\}$	$\{0,1\}$	≥ 1053 (b)

(a): the determinant d was already a prime in 7416 cases. We did not try to factorize it in the other cases.
(b): computation interrupted.

6 Others Operations in an AMNS

In this section we briefly describe the other basic operations in AMNS in order to provide a fully functional system for cryptographic applications. We present methods for converting numbers between binary and AMNS, as well as solutions for addition and subtraction. In the context of elliptic curve cryptography, it is important to notice that, except for the inversion which can be performed only once at the very end of the computations if we use projective coordinates, all the operations can be computed within the AMNS. Thus conversions are only required at the beginning and at the end of the process.

6.1 Conversion from Binary to AMNS

Theorem 2. *If X is an integer such that $0 \leq X < P$, given in classical binary representation, then a representation of X in the AMNS $\mathcal{B}$ is obtained with at most $2(n-1) + \frac{6(n-1)}{k-2}$ calls to Red.*

Proof. We simply remark that $P < 2^{n(k+1)}$ and that the size of the largest coefficient of X is reduced by $\lceil \frac{k}{2} \rceil - 1$ bits after each call to Red. Thus the reduction of $0 \leq X < P$ requires at least $\frac{(n-1)(k+1)}{\lceil \frac{k}{2} \rceil - 1}$ iterations, or more precisely $2(n-1) + \frac{6(n-1)}{k-2}$ is k is even, and $2(n-1) + \frac{4(n-1)}{k-1}$ if k is odd. □

We use theorem 2 by applying the coefficient reduction CR (Algorithm 3) to the vector $(X, 0, \ldots, 0)$.

6.2 Conversion from AMNS to Binary

Given $X = (x_0, \ldots, x_{n-1})_{\mathcal{B}}$, we have to evaluate $X = \sum_{i=0}^{n-1} x_i \gamma^i \bmod P$. The binary representation of X can be obtained with Horner's scheme

$$X = x_0 + \gamma\,(x_1 + \gamma\,(x_2 + \cdots + \gamma\,(x_{n-2} + \gamma\, x_{n-1}) \cdots)) \bmod P.$$

Since γ is of the same order of magnitude as P, the successive modular multiplications must be evaluated with Barrett or Montgomery algorithms. The cost of the conversion is thus at most $3n^3\,T_m$.

6.3 Addition, Subtraction

Given $X = (x_0, \ldots, x_{n-1})_{\mathcal{B}}$ and $Y = (y_0, \ldots, y_{n-1})_{\mathcal{B}}$, the addition is simply given by $S = (x_0 + y_0, \ldots, x_{n-1} + y_{n-1})_{\mathcal{B}^+}$, where $\mathcal{B}^+$ denotes an extension of the AMNS $\mathcal{B}$ where the elements are not necessarily less than ρ. Since the input vectors of our modular multiplication algorithm do not need to have their elements less than ρ, this is a valid representation. However, if the reduction to $\mathcal{B}$ is required, it can be done thanks to algorithm CR.

Subtraction $X - Y$ is performed by adding X and the negative of Y. We use $Z = (z_0, \ldots, z_{n-1})_{\mathcal{B}+}$ as a representation of 0 in $\mathcal{B}^+$, i.e. with $z_i > \rho$ for $i = 0 \ldots n-1$. From (12) and (13) we have

$$Z = \sum_{i=0}^{n-1} z_i \gamma^i \equiv 0 \bmod P,$$

with $z_i = 3\left[2^k - \left(\sum_{j=0}^{i} \xi_j + c \sum_{j=i+1}^{n-1} \xi_j\right)\right]$.

The negative of Y is thus given in $\mathcal{B}^+$ by the vector $(z_0 - y_0, \ldots, z_{n-1} - y_{n-1})_{\mathcal{B}+}$. For the reduction in $\mathcal{B}$, the same remark as for the addition applies.

7 Conclusions

In this paper we defined a new family of moduli suitable for cryptography. In that sense, this research can be seen as an extension of the works by J. Solinas, and J. Chung and A. Hasan. We introduced a new system of representation for the integers modulo P, called Adapted Modular Number System (AMNS), and we proposed a modular multiplication in AMNS which is more efficient than Montgomery. We explained how to construct an AMNS which lead to moduli suitable for fast modular arithmetic, and we explicitly provided examples of primes for cryptographic sizes. Future researches on this subject will be dedicated to the problem of defining an AMNS for a given number p, and to the exploration of the potential advantages of the redundancy of this representation.

Acknowledgments

This work was done during L. Imbert leave of absence at the university of Calgary, with the ATIPS[2] and CISaC[3] laboratories. It was also partly supported by the French ministry of education and research under the ACI 2002, "OpAC, Opérateurs arithmétiques pour la Cryptographie", grant number C03-02.

[2] Advanced Technology Information Processing Systems, `www.atips.ca`
[3] Centre for Information Security and Cryptography, `cisac.math.ucalgary.ca`

References

1. P. Barrett. Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In A. M. Odlyzko, editor, *Advances in Cryptology - Crypto '86*, volume 263 of *LNCS*, pages 311–326. Springer-Verlag, 1986.
2. J. Chung and A. Hasan. More generalized mersenne numbers. In M. Matsui and R. Zuccherato, editors, *Selected Areas in Cryptography – SAC 2003*, volume 3006 of *LNCS*, Ottawa, Canada, August 2003. Springer-Verlag. (to appear).
3. R. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent number 5159632, 1992.
4. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer-Verlag, 2004.
5. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
6. A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 1997.
7. P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
8. J. Solinas. Generalized mersenne numbers. Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 1999.

Arithmetic Operations in Finite Fields of Medium Prime Characteristic using the Lagrange Representation

Jean-Claude Bajard, *Member, IEEE,* Laurent Imbert, *Member, IEEE,* and Christophe Nègre

***Abstract*— In this paper we propose a complete set of algorithms for the arithmetic operations in finite fields of prime medium characteristic. The elements of the fields $\mathbb{F}_{p^k}$ are represented using the newly defined Lagrange representation, where polynomials are expressed using their values at sufficiently many points. Our multiplication algorithm, which uses a Montgomery approach, can be implemented in $O(k)$ multiplications and $O(k^2 \log k)$ additions in the base field $\mathbb{F}_p$. For the inversion, we propose a variant of the extended Euclidean GCD algorithm, where the inputs are given in the Lagrange representation. The Lagrange representation scheme and the arithmetic algorithms presented in the present work represent an interesting alternative for elliptic curve cryptography.**

***Index Terms*— Finite field arithmetic, optimal extension fields, Newton interpolation, Euclidean algorithm, elliptic curve cryptography**

I. Introduction

Finite field arithmetic is an important prerequisite for many scientific applications. The main motivation of this work is in the context of elliptic curve cryptography (ECC), which was proposed independently by Koblitz [1] and Miller [2] in 1985, as an alternative to the existing group-based algorithms [3]. Since then, an immense amount of research has been dedicated to securing and accelerating its implementations. ECC has quickly received a lot of attention because of smaller key-length and increased theoretical robustness (there is no known sub-exponential algorithm to solve the ECDLP problem, which is the foundation of ECC). The relatively small key-size (the security provided by a 160-bit key is equivalent to a 80-bit symmetric-key for block ciphers or a 1024-bit RSA modulus) is a major advantage for devices with limited hardware resources such as smartcards, cell phones or PDAs. As a result ECC has kept receiving commercial acceptance and has been included in numerous standards such as IEEE 1363 [4] and NIST FIPS 186.2 [5]. These standards recommend carefully chosen elliptic curves allowing for secure and efficient implementations over either (large) prime or binary fields. With most of the research following the standard recommendations, alternatives solutions and obvious areas of interest have only received very little attention.

The most studied alternatives to the recommended prime and binary fields are the optimal extension fields (OEF) $\mathbb{F}_{p^k}$, proposed by Bailey and Paar in [6], [7], where p is a prime of the form $2^n - c$, with $|c| \leq n/2$, and there exists an irreducible polynomial over $\mathbb{F}_p$ of the form $X^k - \omega$, with $\omega \in \mathbb{F}_p$. If $c = \pm 1$, then the OEF is said to be of *Type I* and if $\omega = 2$ then the OEF is said to be of *type II*. (See [8] for more details and examples of optimal extension fields of types I and II.)

Elliptic curves defined over fields of characteristic three have recently been considered by Smart and Westwood [9]. Their conclusion is that such curves "could offer greater performance than currently perceived by the community." In 1999 already, Smart proposed a generalization of Solinas work [10] on anomalous binary curves (best known as Koblitz curves [11]) and explained how to use a Frobenius expansion method to speed up the scalar multiplication over fields of odd characteristic.

In this paper, we propose a complete set of arithmetic operations in finite extension fields of medium prime characteristic. We consider the fields $\mathbb{F}_{p^k} \simeq \mathbb{F}_p[X]/(N)$, where N is a monic irreducible polynomial of degree k, called the reduction polynomial. In other words, this isomorphism tells us that the elements of $\mathbb{F}_{p^k}$ can be expressed as the set of all polynomials of degree at most $k-1$, with coefficients in $\mathbb{F}_p$. The arithmetic operations (addition, multiplication) are carried out using polynomial arithmetic modulo N.

Compared to the previous works, the novelty comes from the fact that the operands are represented in the so-called *Lagrange representation* (LR); which means that our polynomials are not represented by their coefficients, but rather using their values at sufficiently many points. We define the Lagrange representation and present the basic operations in LR in Section II. After recalling Montgomery and the OEF algorithms for multiplication over a finite field $\mathbb{F}_{p^k}$ in Section III, we propose a modified Montgomery algorithm for the multiplication modulo N in Section IV. We propose several variants and optimization strategies which can lead to very efficient implementations. For example, under certain conditions, it only requires a linear number of multiplications in $\mathbb{F}_p$. In Section V, we propose a Lehmer-based GCD algorithm [12] for computing the inverse of an element $A \in \mathbb{F}_{p^k}$ modulo N. We discuss the advantages of the Lagrange representation in the context of ECC in Section VI.

J.-C. Bajard is with the LIRMM, CNRS/UM2, 161 rue Ada, 34392 Montpellier cedex 5, FRANCE.

L. Imbert is with the LIRMM, CNRS/UM2, 161 rue Ada, 34392 Montpellier cedex 5, FRANCE; and with the ATIPS labs. and the CISaC, University of Calgary, 2500 University drive N.W., T2N 1N4, Calgary, AB, CANADA.

C. Nègre is with the Université de Perpignan, 52 Av. Paul Alduy, 66860 Perpignan Cedex, FRANCE

II. LAGRANGE REPRESENTATION

In this section, we define the Lagrange representation (LR) and we briefly present the basic operations in LR, namely addition/subtraction, multiplication[1] and the conversions between the coefficient based and the Lagrange representations.

The Lagrange representation scheme can be defined by considering a special case of the Chinese Remainder Theorem (CRT). Let us consider the polynomial $\Psi(X) = \prod_{i=1}^{k}(X-e_i)$, where $e_i \in \mathbb{F}_p$ for $1 \leq i \leq k$, and $e_i \neq e_j$ for $i \neq j$. (Note that this clearly implies $k < p$; we precise that, since we shall also need $\Psi'(X) = \prod_{i=1}^{k}(X - e'_i)$ such that $\gcd(\Psi, \Psi') = 1$ for our multiplication algorithm, the condition will become $2k < p$.) For any arbitrary $U \in \mathbb{F}_p[X]$, we have $U \bmod (X - e_i) = U(e_i)$. By extension, the ring isomorphism given by the Chinese Remainder Theorem

$$\begin{array}{ccc} \mathbb{F}_p[X]/(\Psi) & \longrightarrow & \mathbb{F}_p[X]/(X-e_1) \times \cdots \times \mathbb{F}_p[X]/(X-e_k) \\ U & \longmapsto & (U \bmod (X-e_1), \ldots, U \bmod (X-e_k)) \end{array}, \tag{1}$$

is the evaluation map of the polynomial U at all points $e_1, \ldots, e_k$: $U \mapsto (U(e_1), \ldots, U(e_k))$. Moreover, if $\deg U < k$, then the polynomial

$$U(X) = \sum_{i=1}^{k} u_i \, l_i(X) \tag{2}$$

is the (unique) Lagrange interpolation polynomial satisfying $u_i = U(e_i)$ for $1 \leq i \leq k$, where the l_i's are the Lagrange interpolants such that, for all $1 \leq i \leq k$,

$$l_i(X) = \prod_{j=1, j\neq i}^{k} \frac{X - e_j}{e_i - e_j} \ . \tag{3}$$

In this case, the CRT is equivalent to Lagrange interpolation theorem. In fact, it is useful to think of the CRT as a generalization of interpolation. What both the CRT and Lagrange interpolation theorem tell us is that the interpolation polynomial is unique modulo $\Psi = \prod_{i=1}^{k}(X - e_i)$, so that there is exactly one polynomial $U \in \mathbb{F}_p[X]$ of degree less than $\deg \Psi = k$, which satisfies $U(e_i) = u_i$ for $i = 1, \ldots, k$. In the following, we use this property in order to represent the elements of $\mathbb{F}_{p^k}$.

Definition 1 (Lagrange representation): Let $U \in \mathbb{F}_p[X]$ with $\deg U < k$, and $\Psi = \prod_{i=1}^{k}(X - e_i)$, where $e_i \in \mathbb{F}_p$ for $1 \leq i \leq k$ and $e_i \neq e_j$ for $i \neq j$. If $u_i = U(e_i)$ for $1 \leq i \leq k$, we define the so-called Lagrange representation (LR) of U modulo Ψ as

$$\mathrm{LR}_\Psi(U) = (u_1, \ldots, u_k). \tag{4}$$

One recognized advantage of the CRT is the fact that the costly arithmetic modulo Ψ can be split into several independent arithmetic units, each performing its arithmetic modulo a very simple polynomial (in our case, a binomial of degree one); thus leading to straightforward parallel implementations. For example, additions, subtraction and multiplications in the ring $\mathbb{F}_p[X]/(\Psi)$ (i.e., modulo Ψ) can be performed independently for each modulus. Indeed, if $\mathrm{LR}_\Psi(U) = (u_1, \ldots, u_k)$ and $\mathrm{LR}_\Psi(V) = (v_1, \ldots, v_k)$, then we have

$$\mathrm{LR}_\Psi(U \diamond V) = (u_1 \diamond v_1, \ldots, u_k \diamond v_k), \tag{5}$$

where $\diamond$ belongs to $\{+, -, \times\}$ and $u_i \diamond v_i$ is performed over $\mathbb{F}_p$, i.e., modulo p. Note that the CRT also provides a natural, implicit way to perform the arithmetic modulo Ψ, that we shall exploit in our field multiplication algorithm presented in Section IV.

Since $\deg \Psi = \deg N = k$, field additions and subtractions are equivalent to their ring counterparts, and can be easily computed using (5). The conversion from the coefficient based representation into LR is the evaluation map of a polynomial at many points, with all the operations performed in $\mathbb{F}_p$. The conversion back from LR to the coefficient based representation is an interpolation step that can be computed using (2) and (3). Fast multipoint evaluation and fast interpolation methods are covered in details in [13, chap. 10]. Note that, since all the arithmetic operations can be performed in LR, the conversions steps are only required at the very beginning and the very end of the algorithms and do not affect the global computational cost. In some cases, it is even possible to avoid these conversions steps by performing all the computations in the Lagrange representation (see Section VI).

III. BACKGROUND

In this section, we briefly recall Montgomery modular multiplication algorithm for integers, and its straightforward extension to finite extension fields. Then, we present a modified Montgomery multiplication algorithm, where the elements of the finite field $\mathbb{F}_{p^k}$ are represented in the Lagrange Representation (LR).

A. Montgomery multiplication in $\mathbb{F}_{p^k}$

Montgomery modular multiplication for integers [14] – which, given a, b, n and r such that $\gcd(r, n) = 1$, computes $abr^{-1} \bmod n$ without performing any division – has been generalized to binary fields $\mathbb{F}_{2^k}$ by Koç and Acar [15]. Their solution is a direct adaptation of the original Montgomery algorithm, where the polynomial X^k plays the role of the Montgomery factor r. Given $A, B \in \mathbb{F}_{2^k}$, it computes $ABX^{-k} \bmod N$, where N is the monic irreducible polynomial of degree k in $\mathbb{F}_2[X]$ which defines the field.

We first remark that Koç and Acar's algorithm easily extends to any extension field $\mathbb{F}_{p^k}$. In the polynomial basis representation, the elements of $\mathbb{F}_{p^k}$ can be modeled as the polynomials in $\mathbb{F}_p[X]$ of degree at most $k-1$. Let N be a monic irreducible polynomial of degree k chosen as the reduction polynomial. We define $\Psi = X^k$, such that $\gcd(\Psi, N) = 1$. Then, given $A, B \in \mathbb{F}_p[X]/(N)$, Algorithm 1 can be used to compute $AB\Psi^{-1} \bmod N$.

In this case, choosing $\Psi = X^k$ seems to be a perfect choice, since the reduction modulo X^k (in Step 1) and the division by X^k (in Step 2) are easily implemented. Indeed, given two polynomials $U, V \in \mathbb{F}_p[X]$, with $\deg U, \deg V < k$, we compute $(U \times V) \bmod X^k$ by ignoring the coefficients of $U \times V$ of order larger than $k-1$. Similarly, $(U \times V)/X^k$ is given by the coefficients of $(U \times V)$ of order greater than

[1] A ring multiplication, different from the field operation.

Algorithm 1 Montgomery Multiplication over $\mathbb{F}_{p^k}$

Input: $A, B \in \mathbb{F}_p[X]$, with $\deg A, \deg B \leq k-1$; a monic irreducible polynomial $N \in \mathbb{F}_p[X]$, with $\deg N = k$; $\Psi = X^k$

Output: $AB\Psi^{-1} \bmod N$

1: $Q = -A \times B \times N^{-1} \bmod \Psi$
2: $R = (A \times B + Q \times N)/\Psi$

or equal to k. These computations easily express in terms of matrix operations.

Let us define

$$N = n_0 + n_1 X + \cdots + n_{k-1} X^{k-1} + X^k,$$

and N', the inverse of N modulo X^k, as

$$N' = N^{-1} \bmod X^k = n'_0 + n'_1 X + \cdots + n'_{k-1} X^{k-1}.$$

In Step 1 of Algorithm 1, we compute $Q = -ABN^{-1} \bmod \Psi$ as

$$Q = -\begin{pmatrix} n'_0 & 0 & \dots & 0 \\ n'_1 & n'_0 & \dots & 0 \\ \vdots & & \ddots & \\ n'_{k-1} & n'_{k-2} & \dots & n'_0 \end{pmatrix} \begin{pmatrix} a_0 & 0 & \dots & 0 \\ a_1 & a_0 & \dots & 0 \\ \vdots & & \ddots & \\ a_{k-1} & a_{k-2} & \dots & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{k-1} \end{pmatrix}. \tag{6}$$

Similarly, we evaluate $R = (A\,B + Q\,N)/\Psi$ as

$$R = \begin{pmatrix} 0 & a_{k-1} & \dots & a_2 & a_1 \\ & & & & \\ 0 & 0 & \ddots & & a_2 \\ \vdots & & & \ddots & \\ 0 & 0 & \dots & 0 & a_{k-1} \\ 0 & 0 & \dots & 0 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{k-2} \\ b_{k-1} \end{pmatrix} + \begin{pmatrix} 1 & n_{k-1} & \dots & n_2 & n_1 \\ & & & & \\ 0 & 1 & \ddots & & n_2 \\ \vdots & & \ddots & \ddots & \\ 0 & 0 & \dots & 1 & n_{k-1} \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ \vdots \\ q_{k-2} \\ q_{k-1} \end{pmatrix}. \tag{7}$$

Note that, because N is a monic polynomial of degree k, the diagonal of the second matrix in (7) is composed of ones.

The number of arithmetic operations over $\mathbb{F}_p$ is easily determined. The computation of Q in (6) requires $k(k+1)$ multiplications, and $k(k-1)$ additions, whereas R in (7) is computed in $k(k-1)$ multiplications, and $\frac{(k-1)(k-2)}{2} + \frac{k(k-1)}{2} + (k-1)$ additions. If M and A denote the costs of one multiplication and one addition in $\mathbb{F}_p$ respectively, the total cost of Algorithm 1 is

$$2k^2\,M + (2k^2 - 2k)\,A. \tag{8}$$

For most applications (including ECC) the finite field is fixed and we can reasonably assume that the reduction polynomial N and its inverse modulo X^k are known in advance. In this case, the multiplications by the n_i's and n'_i's in (6) and (7) can be simplified, using optimized algorithms for multiplication by a constant and by constant vectors [16]. The global cost of Algorithm 1 becomes

$$k^2\,M + k^2\,CM + (2k^2 - 2k)\,A, \tag{9}$$

where CM denotes the cost of one multiplication by a constant in $\mathbb{F}_p$.

B. Optimal Extension Fields

Optimal extension fields (OEFs) have been introduced by Bailey and Paar in [6], [7]. The main idea is to select the (prime) characteristic p of the field to closely match the underlying hardware characteristics and to simplify the arithmetic modulo p (for example, the Mersenne prime $p = 2^{31} - 1$ is a good choice for 32-bit architectures). Following the same idea, the reduction polynomial N of degree k is chosen to simplify the reduction modulo N as much as possible. The following definition is taken from [8].

Definition 2 (OEF): An *optimal extension field* (OEF) is a finite field $\mathbb{F}_{p^k}$ such that:

1) $p = 2^n - c$ for some integers n and c with $\log_2 |c| \leq n/2$; and
2) there exists an irreducible polynomial $f(X) = X^m - \omega$ in $\mathbb{F}_p[X]$.

If $c \in \{\pm 1\}$, then the OEF is said to be of *Type I* (p is a Mersenne prime if $c = 1$); if $\omega = 2$, the OEF is said to be of *Type II*.

Examples of OEFs and discussions on how to find irreducible polynomials of the required form are given in [8]. The multiplication of A and B can be performed by an ordinary polynomial multiplication in $\mathbb{Z}[X]$, along with coefficient reductions in $\mathbb{F}_p$, followed by a reduction by the polynomial f. The number of reduction modulo p can be reduced by accumulation strategies on the coefficients of $C = AB$. As pointed out in [8], "the arithmetic resembles that commonly used in prime-field implementations, and multiplication cost in $\mathbb{F}_{p^k}$ is expected to be comparable to that in a prime field $\mathbb{F}_q$ where $q \simeq p^k$ and which admits fast reduction (e.g., the NIST-recommended primes)." This means that the multiplication in $\mathbb{F}_{p^k}$ can be performed in $O(k^2)$ multiplications in $\mathbb{F}_p$. When k is small (as in the case of finite fields used for ECC), fast multiplication algorithms, like Karatsuba-Ofman methods, are not likely to give faster implementations.

In the next section, we propose a modified Montgomery algorithm in $\mathbb{F}_{p^k}$ which, under certain conditions, only requires $O(k)$ multiplications in $\mathbb{F}_p$.

IV. Modified Montgomery multiplication in Lagrange representation

In this section we first modify Algorithm 1 by allowing the polynomial Ψ to be any polynomial of degree k satisfying $\gcd(\Psi, N) = 1$; and by replacing the division by Ψ in Step 2 by a multiplication by Ψ^{-1} modulo another given polynomial Ψ'. Note that this operation is possible only if $\gcd(\Psi, \Psi') = 1$. Then we analyze a special case, where Ψ, Ψ' are the products of first-degree polynomials. Algorithm 2 computes $AB\Psi^{-1} \bmod N$, for any relatively prime polynomials Ψ and Ψ' of degree k satisfying $\gcd(\Psi, N) = 1$ and $\gcd(\Psi, \Psi') = 1$.

Remarks: In Step 1, the notation, $A \times B \bmod (\Psi \times \Psi')$ simply means that we compute both $AB \bmod \Psi$ and $AB \bmod \Psi'$. Also, assume that N_Ψ^{-1} denotes the polynomial $N^{-1} \bmod \Psi$ of degree $\leq k-1$. We remark that Q computed in Step 2 is a polynomial of degree $\leq k-1$, whereas the polynomial $-ABN_\Psi^{-1}$ is of degree $\leq 3k-3$. Since the computations

Algorithm 2 Modified Montgomery Multiplication over $\mathbb{F}_{p^k}$

Input: $A, B \in \mathbb{F}_p[X]$, with $\deg A, \deg B \leq k-1$; a monic irreducible polynomial $N \in \mathbb{F}_p[X]$, with $\deg N = k$; Ψ, Ψ', with $\deg \Psi = \deg \Psi' = k$, and $\gcd(\Psi, \Psi') = \gcd(\Psi, N) = 1$

Output: $AB\Psi^{-1} \bmod N$

1: $T \leftarrow A \times B \bmod (\Psi \times \Psi')$
2: $Q \leftarrow T \times (-N^{-1}) \bmod \Psi$
3: $R \leftarrow (T + Q \times N) \times \Psi^{-1} \bmod \Psi'$

performed in Step 3 have to be carried out modulo Ψ', we have to compute $Q \bmod \Psi'$ from the knowledge of Q only. We note that it is impossible to compute $-ABN_{\Psi}^{-1} \bmod \Psi'$ exactly, but only the polynomial $(-ABN_{\Psi}^{-1} \bmod \Psi) \bmod \Psi'$, denoted $\tilde{Q}$ here. Since Q and $\tilde{Q}$ differ by a multiple of Ψ, both the values $(T + QN)$ and $(T + \tilde{Q}N)$ are multiples of Ψ and the multiplication by Ψ^{-1} modulo Ψ' gives the correct result (see Lemma 1). With this in mind, we shall abusively use Q in the following. Furthermore, if the result R computed in Step 3 has to be reused for another modular multiplication, as in the performance of an exponentiation, we must also compute $R \bmod \Psi$ from $R \bmod \Psi'$. We address the problem of converting polynomials between different Lagrange representations in Sections IV-A and IV-B.

Lemma 1: Algorithm 2 is correct; it returns $AB\Psi^{-1} \bmod N$.

Proof: In Steps 1 and 2, we compute Q, such that $(AB + QN)$ is a multiple of Ψ. Indeed, we have $(AB + QN) \equiv (AB - ABN^{-1}N) \equiv 0 \pmod{\Psi}$. This implies that there exists a polynomial f such that $(AB + QN) = f\Psi$, with $\deg f \leq k-1$. Now, in step 3, we compute R modulo Ψ'. We have $(AB+QN)\Psi^{-1} \equiv (f\Psi)\Psi^{-1} \equiv f \pmod{\Psi'}$. Since $\deg \Psi' = k > \deg f$, we have $(AB+QN)\Psi^{-1} \bmod \Psi' = f$. Since $\deg N \geq k$, we have $R = f = AB\Psi^{-1} \bmod N$ which concludes the proof. □

Of course, Algorithm 2 is advantageous, only if one can define polynomials Ψ, Ψ' such that the arithmetic operations modulo Ψ and Ψ' are easy to implement. The proposed solution takes advantage of the Lagrange representation (see Section II).

Let $E = \{e_1, \ldots, e_k\}$ and $E' = \{e'_1, \ldots, e'_k\}$, be such that $E \cap E' = \emptyset$ and $e_i, e'_i \in \mathbb{F}_p$, for $1 \leq i \leq k$ (in other words, the e_i's and e'_i's are all distinct). We define $\Psi = \prod_{i=1}^{k}(X - e_i)$ and $\Psi' = \prod_{i=1}^{k}(X - e'_i)$, two polynomials of degree k such that $\gcd(\Psi, \Psi') = \gcd(\Psi, N) = 1$. We assume that the inputs A, B are given (or converted) into both LR_{Ψ} and $\mathrm{LR}_{\Psi'}$. We further suppose that the precomputed values are also known in Lagrange representation (modulo Ψ and/or Ψ'), more precisely, we need $\mathrm{LR}_{\Psi}(-N^{-1}) = (\tilde{n}_1, \ldots, \tilde{n}_k)$, $\mathrm{LR}_{\Psi'}(N) = (n'_1, \ldots, n'_k)$ and $\mathrm{LR}_{\Psi'}(\Psi^{-1}) = (\zeta_1, \ldots, \zeta_k)$.

As mentioned earlier, the arithmetic modulo Ψ (resp. Ψ') is automatically and implicitly carried out in Lagrange representation by computing modulo the $(X - e_i)$ for $1 \leq i \leq k$ (resp. modulo the $(X - e'_i)$ for $1 \leq i \leq k$). In the next two sections, we address the problem of converting a polynomial from LR_{Ψ} to $\mathrm{LR}_{\Psi'}$ (the reverse conversion is identical). We consider both Lagrange and Newton's interpolation formulæ and we propose some implementation strategies that can be used to speed-up the implementation in both cases.

A. Lagrange interpolation

Assume that E, E', Ψ and Ψ' are defined as above. If $\mathrm{LR}_{\Psi}(U) = (u_1, \ldots, u_k)$, then $\mathrm{LR}_{\Psi'}(U) = (u'_1, \ldots, u'_k)$ is given by Lagrange interpolation theorem (or the CRT), by computing

$$u'_i = \sum_{j=1}^{k} u_j \left(\prod_{t=1, t\neq j}^{k} \frac{e'_i - e_t}{e_j - e_t} \right), \text{ for } 1 \leq i \leq k. \tag{10}$$

The computations can be easily expressed as a matrix-vector product. By defining the $k \times k$ constant matrix Ω with elements

$$\omega_{i,j} = \prod_{t=1, t\neq j}^{k} \frac{e'_i - e_t}{e_j - e_t}, \quad \text{for } 1 \leq i, j \leq k \tag{11}$$

we have $\mathrm{LR}_{\Psi'}(U) = \Omega \times \mathrm{LR}_{\Psi}(U)$, or equivalently

$$\begin{pmatrix} u'_1 \\ u'_2 \\ \vdots \\ u'_k \end{pmatrix} = \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \ldots & \omega_{1,k} \\ \omega_{2,1} & \omega_{2,2} & \ldots & \omega_{2,k} \\ \vdots & & \ddots & \\ \omega_{k,1} & \omega_{k,2} & \ldots & \omega_{k,k} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{pmatrix}. \tag{12}$$

Similarly, for the reverse conversion from $\mathrm{LR}_{\Psi'}$ to LR_{Ψ}, we define the $k \times k$ constant matrix Ω' with elements

$$\omega'_{i,j} = \prod_{t=1, t\neq j}^{k} \frac{e_i - e'_t}{e'_j - e'_t}, \quad \text{for } 1 \leq i, j \leq k \tag{13}$$

and we compute $\mathrm{LR}_{\Psi}(U) = \Omega' \times \mathrm{LR}_{\Psi'}(U)$.

The complexity of Lagrange interpolation is equal to

$$k^2\, CM + k(k-1)\, A. \tag{14}$$

Hence, in this case, the total cost of Algorithm 2 is

$$2k\, M + (2k^2 + 3k)\, CM + (2k^2 - k)\, A. \tag{15}$$

If one uses a general multiplier for the constant multiplications, then we must assume that $CM = M$ and a sequential implementation of our algorithm does not compare favorably neither against the Montgomery approach (see Algorithm 1) or an OEF implementation. However, because the number of real multiplications is reduced from k^2 to $2k$, hardware implementations can take advantage of Lefèvre's multiplication by integer constants [16] and of Boullis and Tisserand's approach for hardware multiplication by constant matrices [17]. These methods, based on number recoding and dedicated common subexpression factorization algorithms, have been implemented on FPGA for several applications. Based on their results, it is not unreasonable to expect savings of 20-to-40 percent in both area and time for the present application as well; but of course, a precise analysis has to be done to support this claim[2].

The operation count given by (15) does not take into account the fact that some optimizations strategies are applicable.

[2]This work is part of a currently underway joint project with A. Tisserand.

A first optimization we want to point out, which does not come from the Lagrange representation, concerns the base field arithmetic – i.e. the arithmetic modulo p – and the fact that complete reduction modulo p is not always necessary. Actually, it is only needed at the very end of the computational task. For all intermediate computations, several approaches are possible: partial reduction to maintain the values congruent to p and less than $2p$ (especially easy when p is a Mersenne prime), accumulation into two registers (of machine word size) assuming p fits into one register, or three registers can be employed, exactly as mentioned in [8, Example 2.56, page 66] for OEFs. In the following, we will only refer to reduction, or partial reduction modulo p, but for practical implementations, the best options have to be considered.

As for the OEFs, the cost of Algorithm 2 can be significantly reduced by looking for suitable sets of parameters. With OEFs, c and ω are the only parameters that can be adjusted (see Section III-B). Our algorithm gives us more freedom; it allows for $2k$ values (the elements of E and E') to be adjusted. However, this higher degree of freedom makes the optimization process more difficult. Next, we present various optimizations strategies that can be applied to reduce the cost of Algorithm 2 in time and/or space.

By noting that the reduction polynomial N does not directly influence the complexity of our algorithm, it is possible to define the points of E' such that $N = \Psi' + 1$ is irreducible.[3] It is very easy to find such polynomials. Indeed, for given prime p and $k > 0$, the number of irreducible polynomials of the form $\prod_{i=1}^{k}(X - e_i) + 1$ is equal to $\binom{p}{k}$. When p^k is not to small, the probability for a monic uniformly random monic polynomial of degree k in $\mathbb{F}_p[X]$ to be irreducible is close to $1/k$. If we assume that our specific polynomials satisfy this estimate, the number of irreducible polynomials of this form is close to $\binom{p}{k}/k$. Using exhaustive search, we have been able to verify this estimate for small fields (for example, the exact number of irreducible polynomial of degree 3 in $\mathbb{F}_{101}[X]$ of this form is equal to 56661, whereas $\binom{101}{3}/3 = 55550$). For larger extensions, however, we do not know whether it is still valid. In the context of ECC, the degree k of the extension is usually small (see Table III) and we believe that the estimate is correct.

If the points of this of E' are chosen such that $N = \Psi' + 1$ is irreducible, we have $\mathrm{LR}_{\Psi'}(N) = (1, \ldots, 1)$ and Step 3 of Algorithm 2 can be rewritten $R \leftarrow (T + Q) \times \Psi^{-1} \bmod \Psi'$. It saves $k\,CM$ and more importantly, it makes it possible to evaluate $r_i = (t_i + q'_i)\zeta_i \bmod p$ with only one reduction (or partial reduction) instead of two, by allowing the partial result $(t_i + q'_i)\zeta_i$ to be stored into two registers; or without any reduction if one considers three registers.

Using the same idea, it is possible to define E such that $\mathrm{LR}_{\Psi}(-N^{-1}) = (\tilde{n}_1, \ldots, \tilde{n}_k)$ is composed of small integers. Hence, the multiplications by $\tilde{n}_i$ can be replaced by a few number of shifts and additions, and the reduction is greatly simplified as $t_i \times \tilde{n}_i$ fits into a single register plus a few bits. Using a greedy algorithm, we have been able to find such polynomials and the corresponding sets of points. For example, for $p = 8191$ and $k = 13$, the polynomial

$$N = X(X-1)\ldots(X-10)(X-2089)(X-8189)+1,$$

is the first irreducible polynomial (given by our greedy algorithm) such that there exists $k = 13$ points which satisfy $|\tilde{n}_i| \leq 3$; these points are given by

$$E = \{1259, 1872, 1989, 3215, 3667, 3791, 3798, \\ 4197, 4408, 4589, 4615, 4900, 6461\}.$$

The first irreducible polynomial such that $|\tilde{n}_i| \leq 4$ for $1 \leq i \leq 13$ was even faster obtained:

$$N = X(X-1)\ldots(X-11)(X-1558)+1, \\ E = \{140, 286, 950, 1315, 1928, 2293, 2936, \\ 3086, 3619, 5187, 5828, 7374, 7417\}.$$

Another possible optimization is to look for $\tilde{n}_i$'s which are small powers of two. For example, with

$$N = X(X-1)\ldots(X-10)(X-1879)(X-8189)+1, \\ E = \{269, 1036, 1086, 1205, 1484, 2093, 2672, \\ 3151, 3517, 3839, 4111, 6944, 7651\},$$

we have $|\tilde{n}_i| \in \{1, 2, 4\}$ for $1 \leq i \leq k$.

Although it seems to be a difficult task, the freedom in the selection of the points of E and E' can be further exploited by trying to optimize the interpolation matrices Ω and Ω'. For example, one can try to construct matrices with as many small values (possibly 1 or small powers of 2 in absolute value) as possible. The only partial results we have at the moment, based on exhaustive search for small fields, seem difficult to generalize to larger extension fields. However, other matrix optimizations are still possible. In [18], we have detected symmetries between the elements of Ω and Ω' that can contribute to simplified, smaller architectures. The following Lemma holds (see [18] for a proof):

Lemma 2: Assume $e_i = 2i$ and $e'_i = 2i + 1$. Then, from (11) and (13) we have

$$\omega_{i,j} = \prod_{t=1, t\neq j}^{k} \frac{2i+1-2t}{2j-2t}, \quad \text{and}$$

$$\omega'_{i,j} = \prod_{t=1, t\neq j}^{k} \frac{2i-(2t+1)}{2j+1-(2t+1)} \ .$$

Hence, for every $1 \leq i, j \leq k$, we have

$$\omega_{i,j} = \omega'_{k+1-i,k+1-j} \ . \tag{16}$$

Lemma 2 tells us that the operation $\mathrm{LR}_{\Psi}(R) = \Omega' \times \mathrm{LR}_{\Psi'}(R)$, which has to be performed after Step 3 of Algorithm 2, can be replaced by $\overline{\mathrm{LR}_{\Psi}(R)} = \Omega \times \overline{\mathrm{LR}_{\Psi'}(R)}$ (with the matrix Ω instead of Ω'), where $\overline{U}$ denotes the vector composed of the elements of U in the reverse order: $\overline{U} = (u_k, \ldots, u_1)$. In other words, we compute:

$$\begin{pmatrix} r'_k \\ r'_{k-1} \\ \vdots \\ r'_1 \end{pmatrix} = \begin{pmatrix} \omega_{1,1} & \omega_{1,2} & \ldots & \omega_{1,k} \\ \omega_{2,1} & \omega_{2,2} & \ldots & \omega_{2,k} \\ \vdots & & \ddots & \\ \omega_{k,1} & \omega_{k,2} & \ldots & \omega_{k,k} \end{pmatrix} \begin{pmatrix} r_k \\ r_{k-1} \\ \vdots \\ r_1 \end{pmatrix}.$$

[3] Note that N is necessarily monic in this case as $N = \prod_{i=1}^{k}(X - e'_i) + 1 = X^k + \cdots + n_0$.

Let us consider a small example. We define $p = 89$, $k = 5$ and we consider the sets $E = \{2, 4, 6, 8, 10\}$ and $E' = \{1, 3, 5, 7, 9\}$. We compute the constant interpolation matrix Ω that we are going to use for the two interpolation steps:

$$\Omega = \begin{pmatrix} 56 & 44 & 85 & 57 & 26 \\ 26 & 15 & 37 & 3 & 9 \\ 9 & 70 & 16 & 36 & 48 \\ 48 & 36 & 16 & 70 & 9 \\ 9 & 3 & 37 & 15 & 26 \end{pmatrix}$$

Let $N = X^5 + 2X + 1$ be the irreducible polynomial defining the field $\mathbb{F}_{89^5}$. We need the following predefined constant vectors:

$$\begin{aligned} \mathrm{LR}_{\Psi'}(N) &= (4, 72, 21, 1, 61) \\ \mathrm{LR}_{\Psi}(N^{-1}) &= (77, 61, 60, 27, 83) \\ \mathrm{LR}_{\Psi'}(\Psi^{-1}) &= (55, 39, 87, 2, 50). \end{aligned}$$

Now, assume the inputs $A = 17X^4 + 6X + 35$ and $B = 59X^2 + 42X + 11$ are given in LR representation. We have

$$\mathrm{LR}_{\Psi}(A) = (52, 50, 31, 28, 16), \mathrm{LR}_{\Psi'}(A) = (58, 6, 10, 43, 20),$$
$$\mathrm{LR}_{\Psi}(B) = (64, 55, 73, 29, 12), \mathrm{LR}_{\Psi'}(B) = (23, 45, 5, 81, 6).$$

In Step 1, we compute $T = A \times B \bmod \Psi \times \Psi'$:

$$\mathrm{LR}_{\Psi}(T) = (35, 80, 38, 11, 14), \mathrm{LR}_{\Psi'}(T) = (88, 3, 50, 12, 31).$$

Then, in Step 2, we compute

$$\mathrm{LR}_{\Psi}(Q) = (64, 15, 34, 59, 84)$$

that we interpolate modulo Ψ' by computing $\mathrm{LR}_{\Psi'}(Q) = \Omega \times \mathrm{LR}_{\Psi}(Q)$ to get

$$\mathrm{LR}_{\Psi'}(Q) = (43, 75, 49, 53, 53).$$

Next, we evaluate $R = (T + Q \times N) \times \Psi^{-1} \bmod \Psi'$:

$$\mathrm{LR}_{\Psi'}(R) = (60, 54, 67, 41, 63)$$

and we convert it back modulo Ψ using the same matrix Ω, by computing $\overline{\mathrm{LR}_{\Psi}(R)} = \Omega \times \overline{\mathrm{LR}_{\Psi'}(R)}$:

$$\mathrm{LR}_{\Psi}(R) = (21, 13, 77, 5, 1).$$

One can easily check that the result is equal to $AB\Psi^{-1} \bmod N = 2X^4 + 15X^3 + 74X^2 + 49X + 9$ in the Lagrange representation.

B. Newton interpolation

Assume that E, E', Ψ and Ψ' are defined as above and $LR_{\Psi}(Q) = (q_1, \ldots, q_k)$. In order to compute $LR_{\Psi'}(Q) = (q'_1, \ldots, q'_k)$ using Newton's interpolation, we can precompute $k-1$ constants $C_j = ((e_j - e_1)(e_j - e_2) \ldots (e_j - e_{j-1}))^{-1} \bmod p$, for $2 \le j \le k$ and we can evaluate $(\hat{q}_1, \ldots, \hat{q}_k)$ by setting

$$\begin{cases} \hat{q}_1 = q_1 \bmod p, \\ \hat{q}_2 = (q_2 - \hat{q}_1)\, C_2 \bmod p, \\ \hat{q}_3 = (q_3 - (\hat{q}_1 + (e_3 - e_1)\hat{q}_2))\, C_3 \bmod p, \\ \quad \vdots \\ \hat{q}_k = (q_k - (\hat{q}_1 + (e_k - e_1)(\hat{q}_2 + \ldots \\ \qquad + (e_k - e_{k-2})\hat{q}_{k-1}) \ldots))\, C_k \bmod p. \end{cases} \tag{17}$$

As we shall see further, computing the $\hat{q}_i$'s under this form allows for very interesting optimizations and can lead to very efficient implementation, with only a linear number of field multiplications. For parallel implementation, however, it is also possible to compute

$$\hat{q}_i = \big(\ldots ((q_i - \hat{q}_1)(e_i - e_1)^{-1} - \hat{q}_2)(e_i - e_2)^{-1} - \cdots - \hat{q}_{i-1} \big)(e_i - e_{i-1})^{-1} \bmod p, \tag{18}$$

for $2 \le i \le k$. For more details, see the discussion about the *mixed-radix representation* in [19, pp 290–293] and exercise 5.11 in [13, page 125].

Once the $\hat{q}_i$'s have been computed using (17) or (18), the polynomial

$$Q = \hat{q}_1 + \hat{q}_2\psi_1 + \hat{q}_3\psi_1\psi_2 + \cdots + \hat{q}_k\psi_1 \ldots \psi_{k-1} \tag{19}$$

satisfies the conditions

$$\deg Q \le k - 1, \quad Q(e_i) \equiv q_i \pmod p \quad \text{for } 1 \le i \le k. \tag{20}$$

We then evaluate $\mathrm{LR}_{\Psi'}(Q) = (q'_1, \ldots, q'_k)$ using Horner's rule. For $1 \le i \le k$, we compute $q'_i = Q \bmod (X - e'_i) = Q(e'_i)$. From (19), we have

$$q'_i = \big((\ldots (\hat{q}_k(e'_i - e_{k-1}) + \hat{q}_{k-1})(e'_i - e_{k-2}) + \cdots + \hat{q}_2)(e'_i - e_1) + \hat{q}_1 \big) \bmod p. \tag{21}$$

If the C_j are precomputed and if we do not take into account the cost of computing the values $(e_i - e_j)$, the complexity of (17) is equal to $k(k-1)/2\, CM + k(k-1)/2\, A$. Under the same assumptions, the computations in (21), performed for $1 \le i \le k$, require $k(k-1)\, CM + k(k-1)\, A$. The total cost of Newton interpolation is thus $3k(k-1)/2\, CM + 3k(k-1)/2\, A$, which at first, seems very inefficient. However, as for Lagrange interpolation, this general complexity estimate can be significantly reduced by carefully choosing the points of interpolation.

Let us consider the first $2k$ integers: we define $E = \{0, \ldots, k-1\}$ and $E' = \{k, \ldots, 2k-1\}$. In this case (17) rewrites

$$\begin{cases} \hat{q}_1 = q_1 \bmod p, \\ \hat{q}_2 = (q_2 - \hat{q}_1) C_2 \bmod p, \\ \hat{q}_3 = (q_3 - (\hat{q}_1 + 2\hat{q}_2)) C_3 \bmod p, \\ \quad \vdots \\ \hat{q}_k = (q_k - (\hat{q}_1 + (k-1)(\hat{q}_2 + (k-2)(\hat{q}_3 + \ldots \\ \qquad + 2\hat{q}_{k-1}) \ldots))) C_k \bmod p. \end{cases} \tag{22}$$

By taking a closer look at (22), we notice that it requires $k-3$ multiplications by 2, $k-2$ multiplications by 3, …, two multiplications by $k-2$ and one multiplication by $k-1$. Since the largest constant is equal to $k-1$, and k is usually small (see Table III), all these operations can be readily computed with only a few number of shifts and additions. Thus, only $k-1$ constant multiplications by the C_i's are actually needed.

The same applies for (21): the $k(k-1)$ constant multiplications by numbers of the form $(e'_i - e_j)$ can be performed

with a small number of additions and shifts. More precisely, we have to compute

$$\begin{cases} q'_1 = ((\dots(\hat{q}_k \times 2 + \hat{q}_{k-1}) \\ \qquad \times 3 + \dots + \hat{q}_2) \times k + \hat{q}_1) \bmod p, \\ q'_2 = ((\dots(\hat{q}_k \times 3 + \hat{q}_{k-1}) \\ \qquad \times 4 + \dots + \hat{q}_2) \times (k+1) + \hat{q}_1) \bmod p, \\ \quad \vdots \\ q'_k = ((\dots(\hat{q}_k \times (k+1) + \hat{q}_{k-1}) \\ \qquad \times (k+2) + \dots + \hat{q}_2) \times (2k-1) + \hat{q}_1) \bmod p, \end{cases} \tag{23}$$

which requires one multiplication by 2, two multiplications by $3, \dots, k-1$ multiplications by k, $k-1$ multiplications by $k+1, \dots,$ two multiplications by $2k-2$ and one multiplication by $2k-1$. As before, since the largest constant is equal to $2k-1$ and k is small, these operations can be evaluated with only a few number of shifts and additions. For $2 \leq k \leq 23$ (see Table III), the numbers of additions required in the multiplications by the constants $c = 1, 2, \dots, 2k-1$ are given in Table I. We remark that 43 is the first number in the range which requires three additions. We also note that the non-adjacent form (NAF) does not always give the optimal number of addition; for example the multiplication by $45 = (10\bar{1}0\bar{1}01)_2$ can be done with three additions if one considers the NAF, or with only two if one considers its factorization $45 = 9 \times 5$.

c	$\#A$	c	$\#A$	c	$\#A$
1	0	16	0	31	1
2	0	17	1	32	0
3	1	18	1	33	1
4	0	19	2	34	1
5	1	20	1	35	2
6	1	21	2	36	1
7	1	22	2	37	2
8	0	23	2	38	2
9	1	24	1	39	2
10	1	25	2	40	1
11	2	26	2	41	2
12	1	27	2	42	2
13	2	28	1	43	3
14	1	29	2	44	2
15	1	30	1	45	2

TABLE I
NUMBER OF ADDITION ($\#A$) REQUIRED IN THE MULTIPLICATION BY SOME SMALL CONSTANTS c

Moreover, if we assume that p fits in a single machine word and the $\hat{q}_i$'s are also reduced to fit into a single word, then the q'_i's can be computed with a single reduction (or partial reduction) modulo p, by allowing the partial result (before reduction) to be accumulated into two machine words. Let w denote the size (in bits) of one machine word. Since $q'_i = Q(e'_i)$, for $1 \leq i \leq k$, we remark that the size of the largest summand in (19) is equal to

$$w + \left\lfloor \log_2 \prod_{j=1}^{k-1} (e'_i - e_j) \right\rfloor + 1 \ .$$

Moreover, since k terms need to be added to compute the q'_i's, and by noticing that the largest value before reduction in (23) is q'_k, we have

$$|q'_i| < w + \sum_{t=k}^{2k-1} \lfloor \log_2 t \rfloor + k - 1,$$

where $|q|$ denotes the size of q (in bits). Hence, accumulation into two machine words is possible if the following condition is satisfied:

$$\sum_{t=k}^{2k-1} \lfloor \log_2 t \rfloor + k - 1 \leq w \ . \tag{24}$$

As an example, if we consider 32-bit registers ($w = 32$), with $p = 2^{31} - 1$ and $k = 7$, then we have $\sum_{t=7}^{13} \lfloor \log_2 t \rfloor + 7 - 1 = 26 \leq w = 32$, and condition (24) is satisfied.

The asymptotic complexity of the multiplication by a constant k is an open problem. Although Lefèvre conjectured it to be $O((\log k)^{0.85})$, if we consider the best known complexity of $O(\log k)$ additions, the cost of Newton interpolation in the context of Algorithm 2 becomes

$$k - 1\, CM + O(k^2 \log k)\, A, \tag{25}$$

and the total cost of Algorithm 2 is

$$2k\, M + (4k-1)\, CM + O(k^2 \log k)\, A \ . \tag{26}$$

Even if one uses a general multiplier for the multiplications by the large constants, our algorithm shows a better asymptotic complexity than the multiplication algorithms suggested for the OEFs [7], [8], with only a linear number of multiplications. Its complexity is

$$O(k)\, M + O(k^2 \log k)\, A \ . \tag{27}$$

V. INVERSION

In this section, we present an algorithm for computing the inverse of an element A in $\mathbb{F}_{p^k}$ given in Lagrange representation. We use the same notations as before: N is a monic irreducible polynomial of degree k in $\mathbb{F}_p[X]$ and the elements of $\mathbb{F}_{p^k}$ are the polynomials in $\mathbb{F}_p[X]$ of degree less than or equal to $k-1$.

A. Polynomial GCD and inverse computation

Let us start with the more general case of polynomials defined over a field K. If A, B are two polynomials in $K[X]$ with $B \neq 0$, then there exists (unique) polynomials Q and R in $K[X]$ such that

$$A = QB + R \text{ and either } R = 0 \text{ or } \deg R < \deg B \ . \tag{28}$$

We define the polynomial GCD of A and B, not both zero, as a polynomial of greatest degree that divides both A and B. If the polynomial G satisfies this definition, then so does any polynomial of the form uG, where u is a unit[4] in $K[X]$. In other words, there is a set of greatest common divisors of A and B, each one being a unit multiple of the others. The ambiguity is removed by considering that "the" greatest

[4]In our case, the units will be the elements of $\mathbb{F}_p^*$.

common divisor of A and B is the (unique) monic polynomial of greatest degree which divides both A and B.

As for integers, the Bézout identity holds: for A, B not both equal to 0, there exists polynomials U and V such that

$$AU + BV = \gcd(A, B) \ . \tag{29}$$

The polynomials U and V are called the Bézout coefficients of A and B. It is well known that the extended Euclidean algorithm can be used to compute the inverse of an element in K; from (29) we have $U \equiv A^{-1} \pmod B$ and $V \equiv B^{-1} \pmod A$. Many variants of the extended Euclidean algorithm for polynomials have been reported in the literature [19], [20], [21], [22]. A very thorough complexity analysis can be found in [13, pp 46–53]. The presented algorithm is based on the classical Euclidean loop: while $B \neq 0$, $\gcd(A, B) = \gcd(B, R)$, where $R = A \bmod B$ is given by (28). The initial polynomials A, B and the partial quotients, remainders and Bézout coefficients are kept monic to save some operations in the polynomial division. If $\deg A = n \geq \deg B = m$, then the algorithm requires at most $m + 2$ inversions and $\frac{13}{2}nm + O(n)$ additions and multiplications in K. Since we are only interested in one of the Bézout coefficients for the inversion, and because one of the input polynomial is already monic in the case of finite field inversion, the complexity can be reduced to $\frac{9}{2}nm + O(n)$ additions and multiplications, plus at most $m + 1$ inversions.

B. Extended Euclidean algorithm for polynomials in LR

In this section we propose an inversion algorithm, based on the extended Euclidean algorithm for polynomials defined over $\mathbb{F}_{p^k}$, where the input polynomials are given in the Lagrange representation. Given $A \in \mathbb{F}_p[X]$ with $\deg A \leq k - 1$ and $N \in \mathbb{F}_p[X]$, a monic irreducible polynomial of degree k, we compute $A^{-1} \bmod N$. More precisely, algorithm 4 (below) receives $\mathrm{LR}_\Psi(A)$ and $\mathrm{LR}_\Psi(N)$ and returns $\mathrm{LR}_\Psi(A^{-1} \bmod N)$. We first notice that N, which is a polynomial of degree k, cannot be represented in Lagrange representation with only k values; its exact representation would require $k + 1$ values. However, by considering $\mathrm{LR}_\Psi(N)$, we have the exact (unique) representation of $N \bmod \Psi$, which is sufficient to compute $\mathrm{LR}_\Psi(A^{-1} \bmod N) = (A^{-1} \bmod N) \bmod \Psi$.

The main drawback of the Lagrange representation for performing a polynomial division is the ignorance of the degree and coefficients of the polynomials we are manipulating. To bypass this problem, we propose an algorithm which computes the degree and leading coefficient of a polynomial U given in the Lagrange representation.

Assume $U \in \mathbb{F}_p[X]$ with $\deg U < k$ is given in LR; i.e., $\mathrm{LR}_\Psi(U) = (u_1, \ldots, u_k)$. From (2) and (3) we remark that

$$\begin{aligned} U(X) &= \sum_{i=1}^{k} u_i \prod_{j=1, j\neq i}^{k} \frac{X - e_j}{e_i - e_j} \\ &= \sum_{i=1}^{k} \frac{u_i}{\prod_{j=1,j\neq i}^{k}(e_i - e_j)} \prod_{j=1,j\neq i}^{k} (X - e_j) \\ &= \sum_{i=1}^{k} \frac{u_i}{\prod_{j=1,j\neq i}^{k}(e_i - e_j)} X^{k-1} + \cdots \end{aligned} \tag{30}$$

Thus, the coefficient of degree $k - 1$ of U is given by

$$\ell(U) = \sum_{i=1}^{k} u_i \left(\prod_{j=1, j\neq i}^{k} (e_i - e_j) \right)^{-1} \bmod p \ . \tag{31}$$

Thanks to Lagrange interpolation theorem, we know that if $\deg U < k$, it is uniquely defined by $(u_1, \ldots, u_k)$. Hence, if $\deg U < k - 1$, we clearly get $\ell(U) = 0$ in (31). A straightforward solution to find the degree and leading coefficient of U in this case, is to repeat the process for the degrees $k-2$, $k-3$, etc, until one finds a non null coefficient. If (31) tells us that $\deg U \neq k-1$, we know that $k-1$ values are sufficient to define U uniquely, and we can consider any subset of E of size $k - 1$ to compute $\ell(U)$. In Algorithm 3, the sum in (31) is evaluated for $1 \leq i \leq t$, where t is initially set to $m + 1$ and m is the largest possible degree for U, and decremented by 1 each time the tested coefficient is equal to 0. At the end, the degree of U is equal to $t - 1$.

Algorithm 3 Leading term – LT(U,m)

Precomputed: $\zeta_{i,t} = \left(\prod_{j=1,j\neq i}^{t} (e_i - e_j) \right)^{-1} \bmod p$, for $i \leq t$ and $t = 1, \ldots, k$
Input: A polynomial U of degree at most $m \leq k - 1$ given in Lagrange representation: $\mathrm{LR}_\Psi(U) = (u_1, \ldots, u_k)$
Output: (d, c) where $d = \deg U$ and $c = \ell(U)$, such that $U = cX^d + \cdots$

```
1: if m = 0 then
2:    c ← u_1
3: else
4:    t ← m + 1
5:    c ← 0
6:    while c = 0 do
7:       for i ← 1 to t do
8:          c ← c + u_i ζ_{i,t} mod p
9:       if c = 0 then
10:         t ← t − 1
11: return (t − 1, c)
```

We assume that the values $\zeta_{i,t} = \left(\prod_{j=1,j\neq i}^{t} (e_i - e_j) \right)^{-1} \bmod p$ for $1 \leq i \leq t \leq k$ are precomputed. This requires the storage of $k(k+1)/2$ integers less than p. If $\deg U = m \leq k - 1$, the cost of $LT(U, m)$ (in terms of the number of operations in $\mathbb{F}_p$) is

$$(m + 1)\, CM + m\, A. \tag{32}$$

In Algorithm 4, all the variables U_i, V_i are represented in the Lagrange representation. The variables $d(U), \ell(U)$ denote the degree and leading coefficient of U respectively. We use a polynomial version of Lehmer's Euclidean GCD algorithm [12], [19], [22] where one step of each polynomial division is performed; i.e., if $\deg U \geq \deg V$ and $t = \deg(U) - \deg(V)$, then we compute $q = \ell(U)/\ell(V)$ and $R = U - qX^t V$. The process is repeated until a zero remainder is encountered.

To illustrate our inversion algorithm in LR, we consider a small example, using the following parameters: $p = 17$, $k = 3$,

U_1	U_3	V_1	V_3	$d(U_3)$	$\ell(U_3)$	t	q
$(1,1,1)$	$(5,10,3)$	$(0,0,0)$	$(5,4,4)$	2	11	-1	
$(0,0,0)$	$(5,4,4)$	$(1,1,1)$	$(5,10,3)$			1	14
$(3,6,9)$	$(3,13,14)$			2	4	0	5
$(15,1,4)$	$(12,14,16)$			1	2	-1	
$(1,1,1)$	$(5,10,3)$	$(15,1,4)$	$(12,14,16)$			1	14
$(12,7,3)$	$(7,9,11)$			1	2	0	1
$(14,6,16)$	$(12,12,12)$			0	12		
$(4,9,7)$	$(1,1,1)$						

TABLE II

ITERATIONS OF EXTENDED EUCLID'S ALGORITHM 4 IN LR, WITH $\mathrm{LR}_\Psi(A) = (5,10,3)$ AND $\mathrm{LR}_\Psi(N) = (5,4,4)$

Algorithm 4 Inversion over $\mathbb{F}_{p^k}$ in LR

Precomputed: $X_t = \mathrm{LR}_\Psi(X^t)$, for $0 \le t \le k$
Input: $\mathrm{LR}_\Psi(A) = (a_1, \ldots, a_k)$ and $\mathrm{LR}_\Psi(N) = (n_1, \ldots, n_k)$ such that $\gcd(A, N) = 1$,
Output: $\mathrm{LR}_\Psi(A^{-1} \bmod N)$.
1: $(U_1, U_3) \leftarrow (\mathrm{LR}_\Psi(1), \mathrm{LR}_\Psi(A))$
2: $(V_1, V_3) \leftarrow (\mathrm{LR}_\Psi(0), \mathrm{LR}_\Psi(N))$
3: $(d(V_3), \ell(V_3)) \leftarrow (k, 1)$ $\{N$ is monic of degree $k\}$
4: $(d(U_3), \ell(U_3)) \leftarrow LT(U_3, k-1)$ $\{\deg U_3 \le k-1\}$
5: **while** $U_3 \neq 0$ **do**
6: $t \leftarrow d(U_3) - d(V_3)$
7: **if** $t < 0$ **then**
8: $(U_1, U_3) \leftrightarrow (V_1, V_3)$
9: $(d(U_3), \ell(U_3)) \leftrightarrow (d(V_3), \ell(V_3))$
10: $t \leftarrow -t$
11: $q \leftarrow \ell(U_3)\,\ell(V_3)^{-1} \bmod p$
12: $U_1 \leftarrow U_1 - q\,X_t V_1$
13: $U_3 \leftarrow U_3 - q\,X_t V_3$
14: $(d(U_3), \ell(U_3)) \leftarrow LT(U_3, d(U_3) - 1)$
15: **return** U_1

$E = \{1,2,3\}$ and $N = X^3 + 3X^2 + 1$. We compute the inverse of $A = 11X^2 + 6X + 5$ modulo N in Lagrange representation. We have $\mathrm{LR}_\Psi(A) = (5,10,3)$ and $\mathrm{LR}_\Psi(N) = (5,4,4)$. Note that $\mathrm{LR}_\Psi(N) = \mathrm{LR}_\Psi(N \bmod \Psi) = \mathrm{LR}_\Psi(9X^2 + 6X + 7)$. The initialization step gives $\mathrm{LR}_\Psi(U_1) = (1,1,1)$, $\mathrm{LR}_\Psi(U_3) = (5,10,3)$, $\mathrm{LR}_\Psi(V_1) = (0,0,0)$ and $\mathrm{LR}_\Psi(V_3) = (5,4,4)$. We know that $d(V_3) = d(N) = 3$ and $\ell(V_3) = \ell(N) = 1$. The iterations of Algorithm 4 are summarized in Table II. We remark that $\gcd(A, N) = 1$ (given in LR by U_3) and that the inverse of A modulo N, given by U_1, is $\mathrm{LR}_\Psi(A^{-1} \bmod N) = (4,9,7)$. It is easy to verify that $A^{-1} \bmod N$ is equal to $5X^2 + 7X + 9$, which evaluated at $\{1,2,3\}$ gives the same result.

Let us now evaluate the complexity of Algorithm 4. Since $\deg R < \deg U$, the number of (partial) division steps is at most $\deg(N \bmod \Psi) + \deg A = 2k - 2$. If we omit the calls to $LT(U, m)$ for now, each iteration requires: one inversion plus one multiplication for the computation of q in step 10; plus $3k$ multiplications and $2k$ additions for the computations of U_1 and U_3 in steps 11 and 12 (we need k multiplications for qX_t, and $2k$ for qX_tV_j for $j = 1, 3$). How many calls to $LT(U_3, m)$ do we have? Since $\deg U_3, \deg V_3 \le k - 1$ and both U_3 and V_3 have to be reduced (their value are swapped whenever $t < 0$) to polynomials of degree zero, there are exactly two calls to $LT(U_3, i)$, for $1 \le i \le k - 1$ (we note that the calls to $LT(U, 0)$ are free). The total complexity is thus $2k - 2$ inversions plus $(2k-2)M + (2k-2)(3kM + 2kA) + 2\sum_{i=1}^{k-1} LT(U_3, i)$

$$\begin{aligned} &= (6k^2 - 4k - 2)M + (4k^2 - 4k)A \\ &\qquad + 2\sum_{i=1}^{k-1}(i+1)CM + 2\sum_{i=1}^{k-2} i\,A \\ &= (6k^2 - 4k - 2)M + (5k^2 - 5k)A + (k^2 + k - 2)CM, \end{aligned} \tag{33}$$

which can be simplified to $2k-2$ inversions, plus $12k^2 - 8k - 4$ operations in $\mathbb{F}_p$.

A more careful analysis shows that some operations can be saved. Since the degree of U_3 is decreasing from $k-1$ to to 0, it is not necessary to perform the computations in step 12 ($U_3 - qX_tV_3$) for all k values representing $\mathrm{LR}_\Psi(U_3) = (u_1, \ldots, u_k)$. Note that qX_t and U_1 in step 11 must always be computed entirely; i.e., for all k values. In the worst case, the degree of U_3 is decremented by one every two iterations. Thus we can save $1\,M + 1\,A$ for the first two iterations; $2\,M + 2\,A$ for the next two; and so on, up to $(k-1)\,M + (k-1)\,A$ for the last two iterations. This represent a saving of $(k^2 - k)\,M + (k^2 - k)\,A$. Eventually, the total cost of Algorithm 4, is $2k-2$ inversions plus

$$(5k^2 - 3k - 2)\,M + (4k^2 - 4k)\,A + (k^2 + k - 2)\,CM, \tag{34}$$

or equivalently $10k^2 - 6k - 4$ additions and multiplications in $\mathbb{F}_p$.

Compared to the extended Euclidean algorithm presented in Section V-A whose complexity is k inversions and $\frac{9}{2}k^2 + O(k)$ operations in $\mathbb{F}_p$, our inversion algorithm requires roughly twice more operations. This is mainly due to the fact that, using the Lehmer approach, we are performing twice as many Euclidean Euclidean steps. Unfortunately, the Lagrange representation does not allow us to perform a complete polynomial division at each iteration. For hardware implementations, the parallel nature of the Lagrange representation might compensate this extra cost if several processing units are used. In the next section, we justify the interest of being able to perform an inversion in the Lagrange representation in the context of elliptic curve cryptography.

p	form of p	k	l
59	$2^6 - 2^2 - 1$	29	170
67	$2^6 + 3$	29 ... 31	175 ... 188
73	$2^6 + 2^3 + 1$	29 ... 31	179 ... 191
127	$2^7 - 1$	23 ... 61	160 ... 426
257	$2^8 + 1$	23 ... 73	184 ... 584
503	$2^9 - 2^3 - 1$	19 ... 61	170 ... 547
521	$2^9 + 2^3 + 1$	19 ... 61	171 ... 550
8191	$2^{13} - 1$	13 ... 43	168 ... 558
65537	$2^{16} + 1$	11 ... 37	176 ... 592
131071	$2^{17} - 1$	11 ... 31	186 ... 526
524287	$2^{19} - 1$	11 ... 31	208 ... 588
2147483647	$2^{31} - 1$	7 ... 19	216 ... 588
2305843009213693951	$2^{61} - 1$	3 ... 7	182 ... 426

TABLE III

GOOD CANDIDATES FOR p AND k SUITABLE FOR ELLIPTIC CURVE CRYPTOGRAPHY AND THE CORRESPONDING KEY LENGTHS

VI. DISCUSSIONS

For ECC, we usually prefer p and k to be prime. From a security point of view, it is not clear yet whether curves defined over such extension fields render the system less secure. Except for a family of well defined weak curves, the best known approaches to solve the ECDLP are generic algorithms, such as Pollard's Rho method [23]. Some attempts has recently been made to solve the ECDLP for curves defined over small extension fields. In [24], Gaudry proposed a solution which is asymptotically faster than Pollard's Rho when the degree of the extension is equal to zero mod 3 or 4. Explicitly, Gaudry's attack "can solve an elliptic curve discrete logarithm problem defined over $\mathbb{F}_{q^3}$ in time $O(q^{4/3})$, with a reasonably small constant; and an elliptic problem over $\mathbb{F}_{q^4}$ or a genus 2 problem over $\mathbb{F}_{q^2}$ in $O(q^{3/2})$ with a larger constant." In our case, we are only interested in k being a prime. With the light brought by these last results, it is thus preferable to avoid the case $k = 3$. The following Table give some good candidates for p and k and the corresponding key length $l = \lfloor \log_2(p^k) \rfloor$ in bits. For each prime p, we give the form of p and the smallest and largest primes k satisfying the condition $p > 2k$ required for our multiplication. For large p, we only give the extensions which lead to key sizes smaller than 600 bits. The number of possible combinations for the primes p and k is huge. It is of course impossible to list all of them. Since the form of the reduction polynomial does not directly influence the complexity of our multiplication algorithm, we can select a prime p even if there is no "good" reduction polynomial of the desired degree. For example, it is possible to choose $F_3 = 257$ and $F_4 = 65537$, the fourth and fifth Fermat primes, as well as all the Mersenne primes starting from $127 = 2^7 - 1$. Note that the only type II OEF for Mersenne primes up to $2^{89} - 1$ is obtained for $p = 2^{13} - 1$ (see [8]).

Our inversion algorithm requires twice as many operations as the classical Euclidean GCD algorithm for polynomials over a finite field. For ECC, this is not a very serious issue since projective coordinates can be used to avoid all the inversions except one at the end of the point multiplication; i.e., the computation of the point $kP = P + \cdots + P$ (k times), where k is a large integer and P is a point on the curve. (See [8] or [25] for more details about elliptic curve arithmetic.) Furthermore, we remark that all the computations of an ECC protocol (ECDH for example) could be performed in the Lagrange representation. Once Alice and Bob have agreed on a set of parameters (finite field, elliptic curve and base point P on the curve), and have converted the coordinates of P in the Lagrange representation, then, all the computations and exchanges of information could be done in LR. For ECDH, we further notice that there would be no need to perform an interpolation at the end since the results they both get in LR are identical. If k_1 and k_2 are their secret random scalars, they both end up with the point $k_1 k_2 P$ whose coordinates, given in LR, can be considered as several[5] sets of k integers (elements of $\mathbb{F}_p$) as in the coefficient-based representation. In this context, we believe that it is advantageous to be able to perform an inversion in the Lagrange representation using, for example, the extended GCD algorithm presented in Section V.

VII. CONCLUSIONS

In this paper, we have presented a complete set of arithmetic operations for finite fields of the form $\mathbb{F}_{p^k}$. The elements of the field are modeled as polynomials of degree less than k by their values as sufficiently many points (instead of their coefficients). This representation scheme is called the Lagrange representation. Our multiplication, which is a modified Montgomery algorithm, works for $p > 2k$ and can be implemented with only a linear number of multiplication in $\mathbb{F}_p$. The inversion is performed using a variant of the extended Euclidean algorithm, where the degree and leading term of the coefficients of the polynomials manipulated in LR have to be computed at each iteration. The Lagrange representation is particularly attractive for ECC algorithms (with projective coordinates to reduce the number of inversions) since all the computations and exchange of information can possibly be performed within this system.

ACKNOWLEDGMENTS

This work was conducted during Dr. Imbert's leave of absence at the University of Calgary (Canada) with the ATIPS labs. (dept. of Elec. and Comp. Eng.) and the CISaC (dept.

[5]The number of coordinates depends on the type of projective coordinate.

of Mathematics and Statistics). It was funded by the Canadian NSERC Strategic Grant #73-2048, *Novel Implementation of Cryptographic Algorithms on Custom Hardware Platforms* and from the grant *ACI Sécurité Informatique, Opérateurs Cryptographiques et Arithmétique Matérielle* from the French ministry of education and research.

REFERENCES

[1] N. Koblitz, "Elliptic curve cryptosystems," *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, Jan. 1987.
[2] V. S. Miller, "Uses of elliptic curves in cryptography," in *Advances in Cryptology – CRYPTO '85*, ser. Lecture Notes in Computer Science, H. C. Williams, Ed., vol. 218. Springer-Verlag, 1986, pp. 417–428.
[3] A. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. CRC Press, 1997.
[4] IEEE, *IEEE 1363-2000 Standard Specifications for Public-Key Cryptography*, 2000.
[5] National Institute of Standards and Technology, *FIPS PUB 186-2: Digital Signature Standard (DSS)*. National Institute of Standards and Technology, Jan. 2000.
[6] D. Bailey and C. Paar, "Optimal extension fields for fast arithmetic in public-key algorithms," in *Advances in Cryptography – CRYPTO '98*, ser. Lecture Notes in Computer Science, H. Krawczyk, Ed., vol. 1462. Springer-Verlag, 1998, pp. 472–485.
[7] ——, "Efficient arithmetic in finite field extensions with application in elliptic curve cryptography," *Journal of Cryptology*, vol. 14, no. 3, pp. 153–176, 2001.
[8] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2004.
[9] N. P. Smart and E. J. Westwood, "Point muliplication on ordinary elliptic curves over fields of characteristic three," *Applicable Algebra in Engineering, Communication and Computing*, vol. 13, no. 6, pp. 485–497, Apr. 2003.
[10] J. A. Solinas, "Improved algorithms for arithmetic on anomalous binary curves," Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, Research Report CORR-99-46, 1999, updated version of the paper appearing in the proceedings of CRYPTO '97.
[11] N. Koblitz, "CM curves with good cryptographic properties," in *Advances in Cryptography – CRYPTO '91*, ser. Lecture Notes in Computer Science, vol. 576. Springer-Verlag, 1992, pp. 279–287.
[12] D. H. Lehmer, "Euclid's algorithm for large numbers," *American Mathematical Monthly*, vol. 45, no. 4, pp. 227–233, 1938.
[13] J. Von Zur Gathen and J. Gerhard, *Modern Computer Algebra*. Cambridge University Press, 1999.
[14] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
[15] Ç. K. Koç and T. Acar, "Montgomery multiplication in $\mathrm{GF}(2^k)$," *Designs, Codes and Cryptography*, vol. 14, no. 1, pp. 57–69, Apr. 1998.
[16] V. Lefèvre, "Multiplication by an integer constant," INRIA, Research Report 4192, May 2001.
[17] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1271–1282, Oct. 2005.
[18] J.-C. Bajard, L. Imbert, C. Nègre, and T. Plantard, "Multiplication in $\mathrm{GF}(p^k)$ for elliptic curve cryptography," in *Proceedings 16th IEEE symposium on Computer Arithmetic*. IEEE Computer Society, 2003, pp. 181–187.
[19] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3rd ed. Reading, MA: Addison-Wesley, 1997.
[20] R. Crandall and C. Pomerance, *Prime Numbers. A Computational Perspective*. Springer-Verlag, 2001.
[21] J. Sorenson, "Two fast GCD algorithms," *Journal of Algorithms*, vol. 16, no. 1, pp. 110–144, 1994.
[22] ——, "An analysis of Lehmer's Euclidean algorithm," in *Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation – ISSAC '95*. ACM Press, 1995, pp. 254–258.
[23] J. M. Pollard, "Monte Carlo methods for index computation mod p," *Mathematics of Computation*, vol. 32, no. 143, pp. 918–924, July 1978.
[24] P. Gaudry, "Index calculus for abelian varieties and the elliptic curve discrete logarithm problem," Oct. 2004, preprint.
[25] R. M. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.

Jean-Claude Bajard received the Ph.D. degree in computer science in 1993 from the École Normale supérieure de Lyon (ENS), France. He taught mathematics in high school from 1979 to 1990 and served as a research and teaching assistant at the ENS in 1993. From 1994 to 1999, he was an assistant professor at the Université de Provence, Marseille, France. He joined the Université Montpellier 2, Montpellier, France in 1999 where he is currently holding a professor position. Dr. Bajard is a member of the ARITH group in the department of computer science of the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM). His research interests include computer arithmetic and cryptography. He is a member of the IEEE.

Laurent Imbert received the M.S. and Ph.D. degrees in computer science from the Université de Provence, Marseille, France in 1997 and 2000 respectively. He was a Post-Doctoral fellow at the University of Calgary, Alberta, Canada from October 2000 to August 2001. Since October 2001, he has been with the ARITH group of the department of computer science at the Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM), Montpellier, France, where he holds a senior researcher position for the French National Centre for Scientific Research (CNRS). His research interests includes efficient implementation of cryptographic systems, algorithmic number theory and computer arithmetic. He is a member of the IEEE and the International Association for Cryptographic Research (IACR).

Christophe Nègre received his M.S. degree in Mathematics and his Ph.D. in Computer Science from the Université Montpellier 2, Montpellier, France in 2001 and 2004 respectively. From September 2004 to October 2005, he was a research and teaching assistant. Since November 2005, he has been an assistant professor with the DALI group at the Université de Perpignan, France. His research interests are in cryptography and number theory.

MATHEMATICS OF COMPUTATION
Volume 77, Number 262, April 2008, Pages 1075–1104
S 0025-5718(07)02048-0
Article electronically published on December 11, 2007

THE DOUBLE-BASE NUMBER SYSTEM AND ITS APPLICATION TO ELLIPTIC CURVE CRYPTOGRAPHY

VASSIL DIMITROV, LAURENT IMBERT, AND PRADEEP K. MISHRA

Abstract. We describe an algorithm for point multiplication on generic elliptic curves, based on a representation of the scalar as a sum of mixed powers of 2 and 3. The sparseness of this so-called double-base number system, combined with some efficient point tripling formulae, lead to efficient point multiplication algorithms for curves defined over both prime and binary fields. Side-channel resistance is provided thanks to side-channel atomicity.

1. Introduction

Since its discovery by Miller [38] and Koblitz [33] in 1985, Elliptic Curve Cryptography (ECC) has been the subject of a vast amount of publications. Of particularly interest is the quest for fast and side-channel resistant implementations. ECC bases its theoretical robustness on the Elliptic Curve Discrete Logarithm Problem (ECDLP), for which no subexponential algorithm is known. The main operation of any ECC protocol is to compute the point $[n]P = P + \cdots + P$ (n times), for $n \in \mathbb{Z}$ and P a point on the curve. This operation, called the point or scalar multiplication, is the most time consuming and must be carefully implemented. Adaptations of fast exponentiation algorithms [23] have been proposed. The double-and-add algorithm, an adaptation of the square-and-multiply exponentiation method, can be used to compute $[n]P$ in $\log n$ point doublings and $(\log n)/2$ point additions on average. Since the opposite of a point $(-P)$ is easily computed, signed digit representations allow one to reduce the number of point additions: the Non Adjacent Form (NAF), also known as the modified Booth recoding, requires $(\log n)/3$ point additions on average. Window methods (w-NAF) can be used to further reduce the number of additions to $(\log n)/(w+1)$, at the extra cost of a small amount of precomputations (one needs to precompute the points jP for $j = 1, 3, \ldots, 2^{w-1} - 1$; the points $\pm jP$ are used in the point multiplication algorithm). Methods based on efficiently computable endomorphisms on special curves, such as Koblitz curves, are also very attractive. Since the original submission of this manuscript, several interesting papers have been published, which merge the properties of the double-base number system and the efficiently computable endomorphisms on these curves [17, 3].

Received by the editor June 5, 2006 and, in revised form, February 26, 2007.

2000 *Mathematics Subject Classification.* Primary 14H52; Secondary 14G50, 68R99.

Key words and phrases. ECC, point multiplication, double-base number system, side-channel atomicity.

This work was funded by the NSERC Strategic Grant: Novel Implementation of Cryptographic Algorithms on Custom Hardware Platforms.

This work was done during the second author's leave of absence from the CNRS at the University of Calgary.

In this paper, we propose a scalar multiplication algorithm based on a representation of the scalar n as a sum of mixed powers of two coprime integers $p.q$, called the *double-base number system* (DBNS). The inherent sparseness of this representation scheme leads to fewer point additions than other classical methods. For example, let $p = 2, q = 3$ and n be a randomly chosen 160-bit integer. Then one needs only about 22 summands to represent it, as opposed to 80 in standard binary representation and 53 in the non-adjacent form. Although this sparseness does not immediately lead to algorithmic improvements, it outlines one of the main features of this number system and serves as a good starting point for potential applications in cryptography. Double-base representations have recently attracted curiosity in the cryptographic community: Avanzi, Ciet and Sica have investigated double-bases in the case of Koblitz curves, by letting one of the bases be an algebraic number [13, 4]. In [19], Doche et al. proposed a very efficient tripling algorithm[1] for a particular family of curves by using isogeny decompositions; in this context, finding short double-base expansions is also of primary importance. Very recently, Doche and Imbert proposed an extension of the idea which lead to significant speedups in double-base point multiplications for generic curves [20]. This is achieved by considering double-base expansions with digit sets larger than $\{-1, 0, 1\}$.

This paper is an extension of the author's paper at Asiacrypt 2005 [16]. The present version contains a more detailed presentation of the double-base number system, including a theorem on the number of double-base representations for a given positive integer, and some numerical results that illustrate the properties of this encoding scheme, particularly its redundancy and sparseness. It also gives more details on the most important step of the greedy approach used for the conversion from binary, i.e., finding the best approximation of a given integer of the form $p^a q^b$. An efficient alternative solution, which requires some precomputed values to be stored in lookup tables, is presented in [20].

In order to best exploit the sparse and ternary nature of this representation scheme, we also propose new formulae for some useful point operations (tripling, quadrupling, etc.) for generic elliptic curves. We consider curves defined over $\mathbb{F}_p$ with Jacobian coordinates, and curves over $\mathbb{F}_{2^m}$ with both affine and Jacobian coordinates. Some of these formulae are already present in [16]. The derivations are given with more details in the present paper.

Since their discovery by Kocher [35, 34], side-channel attacks (SCA) have become the most serious threat for cryptographic devices. Therefore, protection against various kinds of SCA (power analysis, electromagnetic attacks, fault attacks, etc.) has become a major issue and an interesting area of research. Several countermeasures have been proposed in the literature. We refer interested readers to [8, 2] for details. In this work we consider a solution proposed by Chavalier-Mames et al. called side-channel atomicity [10]. The field operations used in the ADD and DBL curve operations are rearranged and divided into small identical groups, called atomic blocks. These blocks all contain the same operations, in the very same order, to become indistinguishable from the side-channel information leaked to the adversary. Therefore, the trace of a computation composed of a series of ADD and DBL looks like a series of atomic blocks; the adversary cannot distinguish which block belongs to which operation from the side-channel information. Thus the sequence

[1]Note that it is possible to trade one multiplication for a squaring in their formula.

of execution of the curve operations is blinded. This effectively resists simple power attacks.

The sequel of the paper is organized as follows: In Section 2, we introduce the double-base number system, its main properties, and some related problems in number theory and combinatorics. We briefly recall the basics of elliptic curve cryptography and the costs of the classical curve operations in Section 2.2. In Section 3, we present several new curve formulae for the operations that arise in the DBNS point multiplication algorithm presented in Section 4. Finally, we compare our algorithm with several other methods in Section 5.

2. Background

2.1. **The double-base number system.** In this section, we present the main properties of the double-base number system, along with some numerical results in order to provide the reader with some intuitive ideas about this representation scheme and the difficulty of some underlying open problems. We have intentionally omitted the proofs of previously published results. The reader is encouraged to check the references for more details.

We will need the following definitions.

Definition 1 (***S***-integer)**.** Given a set of primes S, an S-integer is a positive integer whose prime factors all belong to S.

Definition 2 (double-base number system)**.** Given p, q, two relatively prime positive integers, the double-base number system (DBNS) is a representation scheme into which every positive integer n is represented as the sum or difference of $\{p, q\}$-integers, i.e., numbers of the form $p^a q^b$:

$$n = \sum_{i=1}^{l} s_i\, p^{a_i} q^{b_i}, \quad \text{with } s_i \in \{-1, 1\} \text{ and } a_i, b_i \geq 0. \tag{1}$$

The size, or length, of a DBNS expansion is equal to the number of terms l in (1). In the following, we will only consider expansions of n as sums of $\{2, 3\}$-integers; i.e., DBNS with $p = 2, q = 3$.

Whether one considers signed ($s_i = \pm 1$) or unsigned ($s_i = 1$) expansions, this representation scheme is highly redundant. For instance, if we assume unsigned double-base representations only, we can prove that 10 has exactly 5 different DBNS representations, 100 has exactly 402 different DBNS representations, $1,000$ has exactly $1,295,579$ different DBNS representations, etc. The following theorem holds.

Theorem 1. *Let n be a positive integer. The number of unsigned DBNS representations of n is given by the following recursing function. $f(1) = 1$ and for $n \geq 1$,*

$$f(n) = \begin{cases} f(n-1) + f(n/3) & \text{if} \quad n \equiv 0 \pmod 3, \\ f(n-1) & \text{otherwise.} \end{cases} \tag{2}$$

Proof. Let us consider the diophantine equation

$$n = h_0 + 3h_1 + 9h_2 + \cdots + 3^k h_k, \tag{3}$$

where $k = \lfloor \log_3(n) \rfloor$ and $h_i \geq 0$ for $i = 0, \ldots, k$. Let $h^{(m)} = (h_0^{(m)}, \ldots, h_k^{(m)})$ be the m-th solution of (3). By substituting each $h_i^{(m)}$ into (3) with its (unique) binary representation, we obtain a specific partition of n as the sum of numbers of the form

$2^a 3^b$. Our problem thus reduces to counting the number of solutions $g(n)$ of (3). This is a very classical integer partition problem, which is known to be associated with the following generating function (see [45] for example):

$$G(z) = \frac{1}{(1-z)(1-z^3)\dots(1-z^{3^k})}. \tag{4}$$

We will prove that (2) admits the same generating function; i.e., that $f(n) = g(n) = [z^n]G(z)$, where the symbol $[z^n]G(z)$ denotes the coefficient of degree n in the series $G(z)$.

Let $F(z) = \sum_{n=1}^{\infty} z^n f(n)$ be the generating function associated with (2). We find that

$$\begin{aligned} F(z) &= \sum_{n=1}^{\infty} z^{3n} f(3n) + \sum_{\substack{n=1 \\ 3\nmid n}}^{\infty} z^n f(n) \\ &= \sum_{n=1}^{\infty} z^{3n} \left(f(3n-1) + f(n)\right) + \sum_{\substack{n=2 \\ 3\nmid n}}^{\infty} z^n f(n-1) + z f(1) \\ &= z + \sum_{n=2}^{\infty} z^n f(n-1) + \sum_{n=1}^{\infty} z^{3n} f(n) \\ &= z + zF(z) + F(z^3). \end{aligned}$$

Thus

$$F(z) = \frac{z}{1-z} + \frac{z^3}{(1-z)(1-z^3)} + \frac{z^9}{(1-z)(1-z^3)(1-z^9)} + \cdots. \tag{5}$$

By noticing that, for $n \geq 1$, the coefficient of z^n in the series $z/(1-z)$ is equal to the coefficient of z^n in the series $1/(1-z)$ and by expressing all terms in (5) with denominator $\prod_i (1-z^{3^i})$, we obtain that

$$[z^n]\, F(z) = [z^n]\, \frac{1}{(1-z)(1-z^3)\dots(1-z^{3^k})} = [z^n]\, G(z), \tag{6}$$

which concludes the proof. □

It is quite clear that the above theorem also applies to numbers of the form $2^a s^b$, where s is an odd integer greater than 1. In this case, the number of solutions of the corresponding partition problem is given by a function similar to (2), where 3 is replaced by s. $\hat{f}(1) = 1$ and for $n \geq 1$,

$$\hat{f}(n) = \begin{cases} \hat{f}(n-1) + \hat{f}(n/s) & \text{if} \quad n \equiv 0 \pmod{s}, \\ \hat{f}(n-1) & \text{otherwise.} \end{cases}$$

Apparently, Mahler was the first to consider the problem of finding good approximations of $\hat{f}(n)$ in his work from 1940 on the Mordel's functional equation [37]. He proved that $\log \hat{f}(n) \approx \dfrac{(\log n)^2}{2 \log s}$. In 1953, Pennington [40] obtained a very good approximation of $\log \hat{f}(sn)$, which gives us an extremely accurate estimation of the number of partitions of n as the sum of $\{2, s\}$-integers:

$$\hat{f}(sn) = e^{O(1)} \left(\frac{n^{C_1 \log n} n^{C_2} \log n^{C_1 \log \log n}}{n^{2C_1 \log \log n} \log n^{C_3 \log n}} \right),$$

where C_1, C_2, C_3 are explicitconstants depending only on s

Theorem 1 tells us that there exist very many ways to represent a given integer in DBNS. Some of these representations are of special interest, most notably the ones that require the minimal number of $\{2,3\}$-integers; that is, an integer can be represented as the sum of l terms, but cannot be represented with $(l-1)$ or fewer. These so-called *canonic* representations are extremely sparse. For example, 127 has 783 different unsigned representations, among which 6 are canonic requiring only three $\{2,3\}$-integers. An easy way to visualize DBNS numbers is to use a two-dimensional array (the columns represent the powers of 2 and the rows represent the powers of 3) into which each non-zero cell contains the sign of the corresponding term. For example, the six canonic representations of 127 are given in Table 1.

TABLE 1. The six canonic unsigned DBNS representations of 127

$2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 108 + 18 + 1$

	1	2	4
1	1		
3			
9		1	
27			1

$2^2 3^3 + 2^4 3^0 + 2^0 3^1 = 108 + 16 + 3$

	1	2	4	8	16
1					1
3	1				
9					
27			1		

$2^5 3^1 + 2^0 3^3 + 2^2 3^0 = 96 + 27 + 4$

	1	2	4	8	16	32
1			1			
3						1
9						
27	1					

$2^3 3^2 + 2^1 3^3 + 2^0 3^0 = 72 + 54 + 1$

	1	2	4	8
1	1			
3				
9				1
27		1		

$2^6 3^0 + 2^1 3^3 + 2^0 3^2 = 64 + 54 + 9$

	1	2	4	8	16	32	64
1							1
3							
9	1						
27		1					

$2^6 3^0 + 2^2 3^2 + 2^0 3^3 = 64 + 36 + 27$

	1	2	4	8	16	32	64
1							1
3							
9			1				
27	1						

Some numerical facts provide a good impression about the sparseness of the DBNS. The smallest integer requiring three $\{2,3\}$-integers in its unsigned canonic DBNS representation is 23; the smallest integer requiring four $\{2,3\}$-integers in its unsigned canonic DBNS representation is 431. Similarly, the next smallest integers requiring five, six and seven $\{2,3\}$-integers are $18,431$, $3,448,733$, and $1,441,896,119$, respectively. The next record-setter would be most probably bigger than one trillion.

If one considers signed representations, then the theoretical difficulties in establishing the properties of this number system dramatically increase. To wit, it is possible to prove that the smallest integer that cannot be represented as the sum or difference of two $\{2,3\}$-integers is 103. The next limit is most probably 4985, but to prove it rigorously, one has to show that none of the following exponential diophantine equations has a solution.

Conjecture 1. *The diophantine equations*

$$\pm 2^a 3^b \pm 2^c 3^d \pm 2^e 3^f = 4985$$

do not have solutions in integers.

One way to tackle this problem would be to extend the results from Skinner [41] on the diophantine equation $ap^x + bq^y = c + dp^z q^w$, to the case where a, b, c, d are not necessarily positive integers. Deriving similar results for a four-term equation (that is, proving that a given number does not admit a signed DBNS representation with 4 terms) seems, however, to be a much more difficult problem.

Finding one of the canonic DBNS representations in a reasonable amount of time, especially for large integers, seems to be a very difficult task. Fortunately, one can use a greedy approach to find a fairly sparse representation very quickly. Given $n > 0$, Algorithm 1 below returns a signed DBNS representation for n. Although it sometimes fails in finding a canonic representation,[2] it is very easy to implement and, more importantly, it guarantees an expansion of sublinear length. Indeed, one of the most important theoretical results about the double-base number system is the following theorem from [18]. It gives us an estimate for the number of terms that one can expect to represent a positive integer.

Algorithm 1 Greedy algorithm

Input A positive integer n
Output The sequence of triples $(s_i, a_i, b_i)_{i\geq 0}$ such that $n = \sum_i s_i 2^{a_i} 3^{b_i}$ with $s_i \in \{-1, 1\}$ and $a_i, b_i \geq 0$

```
s ← 1
while n ≠ 0 do
    Find the best approximation of n of the form z = 2^a 3^b
    print (s, a, b)
    if n < z then
        s ← −s
    n ← |n − z|
```

Theorem 2. *Algorithm* 1 *terminates after* $k \in O(\log n / \log\log n)$ *steps.*

Sketch of proof. (See [18] for a complete proof). Clearly, we have $k \in O(\log n)$ by taking the 2-adic or 3-adic expansions of n. A result by Tijdeman [44] states that there exists an absolute constant C such that there is always a number of the form $2^a 3^b$ between $n - n/(\log n)^C$ and n. Let $n = n_0 > n_1 > n_2 > \cdots > n_l > n_{l+1}$ be the sequence of integers obtained via Algorithm 1. Clearly, for all $i = 0, \ldots, l$, it satisfies $n_i = 2^{a_i} 3^{b_i} + n_{i+1}$ with $n_{i+1} < n_i/(\log n_i)^C$ By defining

[2]The smallest example is 41; the canonic representation is $32+9$, whereas the greedy algorithm returns $41 = 36 + 4 + 1$.

$l = l(n)$ such that $n_l > f(n) \geq n_{l+1}$, we obtain that $k = l(n) + O(\log f(n))$. The proof is completed by showing that the function $f(n) = \exp(\log n/\log\log n)$ gives $l(n) \in O(\log n/\log\log n)$. □

The complexity of the greedy algorithm mainly depends on the complexity of step 3: finding the $\{2,3\}$-integer which best approximates n. The problem can be reformulated in terms of linear forms of logarithms. For the best default approximation, one has to find two integers $a, b \geq 0$ such that

$$a\log 2 + b\log 3 \leq \log n, \tag{7}$$

and such that no other integers $a', b' \geq 0$ give a better left approximation to $\log n$.

In [7], Berthè and Imbert proposed an algorithm based on Ostrowski's number system [1] for real numbers [6]; a number system associated with the series $(q_i\alpha - p_i)_{i\geq 0}$, where $(p_i/q_i)_{i\geq 0}$ is the series of the convergents of the continued fraction expansion of an irrational number $\alpha \in]0,1[$. In this system, every real number $-\alpha \leq \beta < 1-\alpha$ can be uniquely written as

$$\beta = \sum_{i=1}^{+\infty} b_i(q_{i-1}\alpha - p_{i-1}), \tag{8}$$

where

$$\begin{cases} 0 \leq b_1 \leq a_1 - 1, \text{ and } 0 \leq b_i \leq a_i \text{ for } i > 1, \\ b_i = 0 \text{ if } b_{i+1} = a_{i+1}, \\ b_i \neq a_i \text{ for infinitely many even and odd integers.} \end{cases}$$

The algorithm presented in [7] uses the fact that β can be approximated modulo 1 by numbers of the form $A\alpha$; the best successive approximations being given by the series

$$A_j = \sum_{i=1}^{j} b_i q_{i-1}. \tag{9}$$

By setting $\alpha = \log 2/\log 3$, and $\beta = \{\log n/\log 3\}$ (where $\{\}$ denotes the fractional part), the solution of our problem is given by $a = \sum_{i=1}^{m} b_i q_{i-1}$ and $b = \lfloor\beta\rfloor - \sum_{i=1}^{m} b_i p_{i-1}$, where m is the largest integer such that $b \geq 0$.

One can prove that the algorithm proposed in [7] to find the best approximation of n of the form $2^a 3^b$ has complexity $O(\log\log n)$. Since the greedy algorithm finishes in $O(\log n/\log\log n)$ iterations, its complexity is thus in $O(\log n)$. It is important to remark that for a recoding algorithm between two additive number systems, we cannot do better.

2.2. **Elliptic curve cryptography.**

Definition 3. An elliptic curve E over a field K, denoted by E/K, is defined by its Weierstraß equation

$$E/K : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \tag{10}$$

where $a_1, a_2, a_3, a_4, a_6 \in K$ and Δ, the discriminant of E, is different from 0.

In practice, the general equation (10) can be greatly simplified by applying admissible changes of variables. If the characteristic of K is not equal to 2 and 3, one can rewrite it as

$$y^2 = x^3 + a_4x + a_6, \tag{11}$$

where $a_4, a_6 \in K$, and $\Delta = 4a_4^3 + 27a_6^2 \neq 0$.

When the characteristic of K is equal to 2, the *ordinary* or *non-supersingular* form[3] of an elliptic curve is given by

$$y^2 + xy = x^3 + a_2x^2 + a_6, \tag{12}$$

where $a_2, a_6 \in K$ and $\Delta = a_6 \neq 0$.

The set $E(K)$ of K-rational points on an elliptic curve E/K consists of the affine points (x, y) satisfying (10) along with the special point $\mathcal{O}$ called the *point at infinity*. It forms an abelian group, where the operation (denoted additively) is defined by the well-known law of chord and tangent (see [2] for details). Given P, Q on the curve, the group law slightly differs as to whether one considers the computation of $P + Q$ with $P \neq \pm Q$ or the computation of $P + P = [2]P$. We talk about point addition (ADD) and point doubling (DBL).

There exist many ways to represent the points of $E(K)$. In affine coordinates ($\mathcal{A}$), both the ADD and DBL operations involve expensive field inversions (to compute the slope of the chord/tangent). In order to avoid these inversions, several inversion-free systems of coordinates have been proposed. The choice of such a system has to be made according to several parameters including memory constraints and the relative cost between one field inversion and one field multiplication, often called the $[i]/[m]$ ratio. For binary fields, several works report a ratio $[i]/[m]$ between 3 and 10 depending on the implementation options (see [15, 26]). For prime fields, however, this ratio is more difficult to estimate precisely. In [22], Fong et al. consider that $[i]/[m] > 40$ on general-purpose processors. In fact, different experiments can provide very different results. If, for example, one uses GMP [24] to compare the cost of these two operations over large prime fields, one does not necessarily notice a huge difference in terms of computational time. This is due to the fact that the multiplication and reduction (modulo p) algorithms implemented in GMP are generic; i.e. they do not take into account the possible special form of the modulus. When implementing ECC/HECC algorithms, it is a good idea to use primes that allow fast modular arithmetic, such as those recommended by the NIST [39], the SEC Group [43], or more generally the primes belonging to what Bajard et al. called the Mersenne family [42, 11, 5]. In these cases, the multiplication becomes much more efficient than the inversion. In hardware implementations using inversion-free systems, the space for the inverter is often saved and the single final inversion is done using Fermat's little theorem. Although the overhead due to inversions in less dramatic for curves defined over $\mathbb{F}_{2^m}$, affine coordinates are not necessarily the best choice in practice, especially for software implementations [15]. In this paper we consider projective coordinates for curves defined over $\mathbb{F}_p$ and both affine and projective for curves defined over $\mathbb{F}_{2^m}$. More exactly, we use Jacobian coordinates ($\mathcal{J}$), a special class of projective coordinates, where the point $(X : Y : Z)$ corresponds to the affine point $(X/Z^2, Y/Z^3)$ when $Z \neq 0$. The point at infinity is represented as $(1 : 1 : 0)$. The opposite of $(X : Y : Z)$ is $(X : -Y : Z)$. Clearly there exist infinitely many points in the projective space which correspond to the same affine point. We use the common abusive notation $(X : Y : Z)$ to represent any representative of the equivalence class given by the relation of projection.

As we shall see, our DBNS-based point multiplication algorithm uses several basic operations (addition, doubling, tripling, etc.). In Sections 2.2.1 and 2.2.2, we recall the complexity of some of these curve operations, expressed in terms of the

[3]By opposition to supersingular curves of the form $y^2 + a_3y = x^3 + a_4x + a_6$ with $a_3 \neq 0$.

number of elementary operations in the field K. The interested reader is encouraged to check the literature [27, 2] for detailed descriptions of these algorithms. We use $[i]$, $[s]$ and $[m]$ to denote the cost of one inversion, one squaring and one multiplication, respectively. We always leave out the cost of field additions. For curves defined over $\mathbb{F}_p$ it is widely assumed that $[s] = 0.8[m]$. It is therefore a good idea to trade multiplications in favor of squarings whenever possible. However, as we shall see, our algorithms can be protected against SCA using side-channel atomicity [10]. In such cases, because squarings and multiplications must be performed using the same multiplier in order to be indistinguishable, we have to consider that $[s] = [m]$. For curves defined over binary fields, however, squarings are free (when normal bases are used to represent the elements of $\mathbb{F}_{2^m}$) or of negligible cost (squaring is a linear operation in polynomial basis); the complexity is thus mainly driven by the numbers of inversions and multiplications.

2.2.1. *Elliptic curves defined over $\mathbb{F}_p$.* When Jacobian coordinates are used the addition ($\text{ADD}^{\mathcal{J}}$) and doubling ($\text{DBL}^{\mathcal{J}}$) operations require $12[m]+4[s]$ and $4[m]+6[s]$, respectively. The cost of $\text{DBL}^{\mathcal{J}}$ can be reduced to $4[m] + 4[s]$ when $a_4 = -3$ (Brier and Joye [9] proved that most randomly chosen curves can be mapped to an isogeneous curve with $a_4 = -3$). Also, if one of the points is given in affine coordinates ($Z = 1$), then the cost of the so-called *mixed addition* ($\mathcal{J} + \mathcal{A} \to \mathcal{J}$) reduces to $8[m]+3[s]$. Several algorithms have been proposed for repeated doublings (the computation of $[2^w]P$) in the case of binary fields [25, 36]. For prime fields, an algorithm was proposed by Itoh et al. in [28], which is more efficient than w invocations of $\text{DBL}^{\mathcal{J}}$. In the general case ($a_4 \neq -3$) it requires $4w[m]+(4w+2)[s]$. When $a_4 = -3$ the cost of w-$\text{DBL}^{\mathcal{J}}$ is exactly the same as the cost of w doublings; i.e. $4w[m] + 4w[s]$. In Table 2, we summarize the complexity of these different elliptic curve operations. In the third column, we give the minimum number of registers required to achieve the corresponding complexities. See [2, chapter 13] for a complete description.

TABLE 2. Elliptic curve operations in Jacobian coordinates for curves defined over $\mathbb{F}_p$

Curve operation	Complexity	# Registers
$\text{DBL}^{\mathcal{J}}$	$4[m] + 6[s]$	6
$\text{DBL}^{\mathcal{J},\, a_4=-3}$	$4[m] + 4[s]$	5
$\text{ADD}^{\mathcal{J}}$	$12[m] + 4[s]$	7
$\text{ADD}^{\mathcal{J}+\mathcal{A}}$	$8[m] + 3[s]$	7
w-$\text{DBL}^{\mathcal{J}}$	$4w[m] + (4w + 2)[s]$	7

2.2.2. *Elliptic curves defined over $\mathbb{F}_{2^m}$.* For curves defined over $\mathbb{F}_{2^m}$ we give the cost of the most common operations in affine and Jacobian coordinates,[4] as they both have practical interest. Note that we only consider ordinary curves (for details concerning *supersingular* curves, see [2]).

[4]We do not consider Lopez-Dahab coordinates, which are also very attractive for curves defined over binary fields, simply because we did not find any good tripling formula in this system of coordinate.

Affine coordinates: The addition ($\text{ADD}^{\mathcal{A}}$) and doubling ($\text{DBL}^{\mathcal{A}}$) operations can be computed with the same number of field operations in $1[i] + 1[s] + 2[m]$. In [21], Eisenträger et al. proposed efficient formulae for tripling ($\text{TPL}^{\mathcal{A}}$), double-and-add ($\text{DA}^{\mathcal{A}}$) and triple-and-add ($\text{TA}^{\mathcal{A}}$). By trading some inversions for a small number of multiplications, their results have been further improved by Ciet et al. [12] when $[i]/[m] > 6$. We summarize the complexities of each of these operations in Table 4, with the break-even points between the two options for $\text{DA}^{\mathcal{A}}$, $\text{TPL}^{\mathcal{A}}$ and $\text{TA}^{\mathcal{A}}$.

TABLE 3. Elliptic curve operations in affine coordinates for curves defined over $\mathbb{F}_{2^m}$

Curve operation	[21]	[12]	break-even point
$\text{DBL}^{\mathcal{A}}$	$1[i] + 1[s] + 2[m]$		–
$\text{ADD}^{\mathcal{A}}$	$1[i] + 1[s] + 2[m]$		–
$\text{DA}^{\mathcal{A}}$	$2[i] + 2[s] + 3[m]$	$1[i] + 2[s] + 9[m]$	$[i]/[m] = 6$
$\text{TPL}^{\mathcal{A}}$	$2[i] + 2[s] + 3[m]$	$1[i] + 4[s] + 7[m]$	$[i]/[m] = 4$
$\text{TA}^{\mathcal{A}}$	$3[i] + 3[s] + 4[m]$	$2[i] + 3[s] + 9[m]$	$[i]/[m] = 5$

Jacobian coordinates: The use of Jacobian coordinates for curves defined over $\mathbb{F}_{2^m}$ was proposed by Hankerson et al. in [26], after noticing that their software implementation[5] using affine coordinates was leading to a ratio $[i]/[m] \simeq 10$. In the general case, an addition requires $16[m] + 3[s]$; it reduces to $11[m] + 3[s]$ if one of the points is given in affine coordinates. The doubling operation can be computed in $5[m] + 5[s]$, including one multiplication by a_6 (see [2, chapter 13] for detailed algorithms).

TABLE 4. Elliptic curve operations in Jacobian coordinates for curves defined over $\mathbb{F}_{2^m}$

Curve operation	Complexity	# Registers
$\text{DBL}^{\mathcal{J}}$	$5[m] + 5[s]$	5
$\text{ADD}^{\mathcal{J}}$	$16[m] + 3[s]$	10
$\text{ADD}^{\mathcal{J}+\mathcal{A}}$	$11[m] + 3[s]$	–

We note that, although the doubling algorithm can be computed using 5 multiplications and 5 squarings, it requires 6 atomic blocks if one considers (s, m, a)-blocks (i.e., blocks composed of 1 squaring, 1 multiplication and 1 addition in that order). Since squarings are almost free over $\mathbb{F}_{2^m}$, it is much better to consider (s, s, m, a)-blocks, as it indeed allows one to perform a doubling in 5 blocks; i.e. in 5 multiplications. We express both the doubling and the mixed addition with (s, s, m, a)-blocks in Tables 15 and 16 of Appendix B.

3. New curve arithmetic formulae

This section is devoted to new, efficient curve operations, which have been defined in order to best exploit the sparseness and the ternary nature of the DBNS representation. Namely, we give formulae for:

[5]For hardware implementation, however, affine coordinates seem to be the best choice.

- point tripling, consecutive triplings, as well a very specific consecutive doublings following (consecutive) tripling(s) in Jacobian coordinates for curves defined over $\mathbb{F}_p$,
- point tripling, point quadrupling (QPL$^{\mathcal{A}}$) and combined quadruple-and-add (QA$^{\mathcal{A}}$) in affine coordinates for curves defined over $\mathbb{F}_{2^m}$,
- point tripling (TPL$^{\mathcal{J}}$) in Jacobian coordinates for curves defined over $\mathbb{F}_{2^m}$.

3.1. **Curves defined over $\mathbb{F}_p$ using Jacobian coordinates.** In this section, we derive equations to obtain an efficient point tripling formula (TPL$^{\mathcal{J}}$) in Jacobian coordinates for curves defined over $\mathbb{F}_p$. (This formula was already present in [16].) Then, we explain how some field operations can be saved when several triplings (w-TPL$^{\mathcal{J}}$) have to be computed, or when several doublings have to be computed right after one or more triplings (w-TPL$^{\mathcal{J}}$/w'-DBL$^{\mathcal{J}}$). As we shall see in Section 4, this very specific operation occurs quite often in our scalar multiplication algorithm.

To simplify, we start with affine coordinates. Let $P = (x_1, y_1) \in E(K)$ be a point on an elliptic curve E defined by (11). By definition, we have $[2]P = (x_2, y_2)$, where

$$\lambda_1 = \frac{3x_1^2 + a_4}{2y_1}, \quad x_2 = \lambda_1^2 - 2x_1, \quad y_2 = \lambda_1(x_1 - x_2) - y_1 \ . \tag{13}$$

We can compute $[3]P = [2]P + P = (x_3, y_3)$, by evaluating λ_2 (the slope of the chord between the points $[2]P$ and P) as a function of x_1 and y_1 only. We have

$$\begin{aligned} \lambda_2 &= \frac{y_2 - y_1}{x_2 - x_1} \\ &= -\lambda_1 - \frac{2y_1}{x_2 - x_1} \\ &= -\frac{3x_1^2 + a_4}{2y_1} - \frac{8y_1^3}{(3x_1^2 + a_4)^2 - 12x_1y_1^2} \ . \end{aligned} \tag{14}$$

We further remark that

$$\begin{aligned} x_3 &= \lambda_2^2 - x_1 - x_2 \\ &= \lambda_2^2 - x_1 - \lambda_1^2 + 2x_1 \\ &= (\lambda_2 - \lambda_1)(\lambda_2 + \lambda_1) + x_1 \end{aligned} \tag{15}$$

and

$$\begin{aligned} y_3 &= \lambda_2(x_1 - x_3) - y_1 \\ &= -\lambda_2(\lambda_2 - \lambda_1)(\lambda_2 + \lambda_1) - y_1 \ . \end{aligned} \tag{16}$$

Thus $[3]P = (x_3, y_3)$ can be computed directly from x_1, y_1 without evaluating the intermediate values x_2 and y_2.

By replacing x_1 and y_1 by X_1/Z_1^2 and Y_1/Z_1^3 respectively, we obtain the following point tripling formula in Jacobian coordinates. Let $P = (X_1 : Y_1 : Z_1)$ be a point on the curve $\neq \mathcal{O}$. Then the point $[3]P = (X_3 : Y_3 : Z_3)$ is given by

$$\begin{aligned} X_3 &= 8Y_1^2(T - ME) + X_1E^2, \\ Y_3 &= Y_1\left(4(ME - T)(2T - ME) - E^3\right), \\ Z_3 &= Z_1E, \end{aligned} \tag{17}$$

where $M = 3X_1^2 + a_4Z_1^4$, $E = 12X_1Y_1^2 - M^2$ and $T = 8Y_1^4$.

The cost of (17) is $10[m] + 6[s]$. If one uses side-channel atomicity to resist SCA, this is equivalent to $16[m]$. We express the TPL$^{\mathcal{J}}$ algorithm in terms of atomic blocks in Table 13 of Appendix A. In comparison, computing $[3]P$ using the doubling and addition algorithms from [10], expressed as a repetition of atomic blocks, costs $10[m] + 16[m] = 26[m]$.

As we shall see, consecutive triplings; i.e., expressions of the form $[3^w]P$, occur quite often in our point multiplication algorithm. From (17), we remark that the computation of the intermediate value $M = 3X_1^2 + a_4Z_1^4$ requires $1[m] + 3[s]$ (we omit the multiplication by 3). If we need to compute $[9]P$, we then have to evaluate $M' = 3X_3^2 + a_4Z_3^4$. Since $Z_3 = Z_1E$, we have $a_4Z_3^4 = a_4Z_1^4E^4$ (where $E = 12X_1Y_1^2 - M^2$), and $a_4Z_1^4$ and E^2 have already been computed. Hence, using these precomputed subexpressions, we can compute $M' = 3X_3^2 + (a_4Z_1^4)(E^2)^2$, with $1[m] + 2[s]$. The same technique can be applied to save one multiplication for each subsequent tripling. Thus, we can compute 3^wP with $(15w+1)[m]$, which is always better than w invocations of the tripling algorithm. The atomic blocks version of w-TPL$^{\mathcal{J}}$ is given in Table 14 of Appendix A. Note that the idea of reusing a_4Z^4 for multiple doublings was first proposed by Cohen et al. in [14], where modified Jacobian coordinates are proposed.

From Table 2, DBL$^{\mathcal{J}}$ normally requires $4[m] + 6[s]$, or equivalently 10 blocks of computation if side-channel atomicity is used. The scalar multiplication algorithm presented in the next section very often[6] requires a w'-DBL$^{\mathcal{J}}$ to be computed right after a w-TPL$^{\mathcal{J}}$. This is due to the fact that we impose some conditions on the double-base expansion of the scalar k (see details in Section 4). When this occurs, it is possible to save $1[s]$ for the first DBL$^{\mathcal{J}}$ using subexpressions computed for the last tripling. The next $(w'-1)$-DBL$^{\mathcal{J}}$ are then computed with $(4w'-4)[m]+(4w'-4)[s]$. (The details of these algorithms are given in Appendix A.) We summarize the complexities of these curve operations in Table 5, together with the number of registers required in each case.

TABLE 5. Tripling algorithms in Jacobian coordinates for curves defined over $\mathbb{F}_p$

Curve operation	Complexity	# Registers
TPL$^{\mathcal{J}}$	$6[s] + 10[m]$	8
w-TPL$^{\mathcal{J}}$	$(4w + 2)[s] + (11w - 1)[m]$	10
w-TPL$^{\mathcal{J}}$/w'-DBL$^{\mathcal{J}}$	$(11w + 4w' - 1)[s] + (4w + 4w' + 3)[m]$	10

3.2. **Curves defined over $\mathbb{F}_{2^m}$ using affine coordinates.** In this section, we propose new affine formulae for the computation of $[3]P$, $[4]P$ and $[4]P \pm Q$ for curves defined over $\mathbb{F}_{2^m}$.

Let us first recall the equations for the doubling operation. Given $P = (x_1, y_1)$, $P \neq -P$, we have $[2]P = (x_2, y_2)$, where

$$\lambda_1 = x_1 + \frac{y_1}{x_1}, \quad x_2 = \lambda_1^2 + \lambda_1 + a_2, \quad y_2 = \lambda_1(x_1 + x_2) + x_2 + y_1. \tag{18}$$

[6]The only exceptions occur when the expansion of k contains a series of consecutive $\{2, 3\}$-integers with identical binary exponents.

We shall compute $[3]P = (x_3, y_3)$ as $[3]P = P + [2]P$. From (18), we obtain after simplifications

$$x_3 = (\lambda_1 + \lambda_2 + 1)^2 + \lambda_2 + \lambda_1 + 1 + x_1, \quad y_3 = \lambda_2(x_1 + x_3) + x_3 + y_1, \tag{19}$$

where $\lambda_1 = x_1 + y_1/x_1$ and $\lambda_2 = (y_1 + y_2)/(x_1 + x_2)$. In order to reduce the number of field operations for the computations of x_3 and y_3, we want to get a convenient expression for $(\lambda_1 + \lambda_2 + 1)$. We start by expressing $(x_1 + x_2)$ in terms of x_1 only. We have:

$$x_1 + x_2 = x_1 + \lambda_1^2 + \lambda_1 + a_2 = \frac{x_1^4 + (y_1^2 + x_1 y_1 + a_2 x_1^2)}{x_1^2}.$$

From (12), since P is on the curve, we define $\alpha = x_1^4 + x_1^3 + a_6$ such that

$$x_1 + x_2 = \frac{\alpha}{x_1^2}. \tag{20}$$

Now, going back to the expression for λ_2, we have:

$$\begin{aligned} \lambda_2 &= \lambda_1 + \frac{x_2}{x_1 + x_2} \\ &= \lambda_1 + \frac{x_1}{x_1 + x_2} + 1 \\ &= \lambda_1 + \frac{x_1^3}{\alpha} + 1 \qquad (21) \\ &= \lambda_1 + \frac{x_1^4 + a_6}{\alpha} \qquad (22) \\ &= \frac{\alpha(x_1^2 + y_1) + x_1^4 + a_6}{x_1 \alpha}, \end{aligned}$$

$$\lambda_2 = \frac{\beta}{x_1 \alpha}, \tag{23}$$

where $\beta = \alpha(x_1^2 + y_1) + x_1^4 + a_6$. From (21), we remark that

$$\lambda_1 + \lambda_2 + 1 = \frac{x_1^3}{\alpha}. \tag{24}$$

Replacing (23) and (24) in (19), we finally get

$$x_3 = \left(\frac{x_1^4}{x_1 \alpha}\right)^2 + \frac{x_1^4}{x_1 \alpha} + x_1, \tag{25}$$

$$y_3 = \frac{\beta}{x_1 \alpha}(x_1 + x_3) + x_3 + y_1. \tag{26}$$

It is easy to see that computing α requires $1[m] + 2[s]$; 1 extra $[m]$ gives β and $1/(x_1\alpha)$ is computed with another $1[m] + 1[i]$. The total cost for this new point tripling is thus $1[i] + 3[s] + 6[m]$. This is always better than the formula proposed in [12] (it saves $1[m] + 1[s]$) and becomes faster than the equation from [21] as soon as $[i] \geq 3[m]$ (see Table 3 for the exact costs of these previous methods).

For the quadrupling operations, the trick used in [21] by Eisenträger et al., which consists in evaluating only the x-coordinate of $[2]P$ when computing $[2]P \pm Q$, can also be applied to speed-up the quadrupling ($\text{QPL}^{\mathcal{A}}$) operation. From (19), we compute $[4]P = [2]([2]P) = (x_4, y_4)$ as

$$\lambda_2 = x_2 + \frac{y_2}{x_2}, \quad x_4 = \lambda_2^2 + \lambda_2 + a_2, \quad y_4 = \lambda_2(x_1 + x_4) + x_4 + y_1. \tag{27}$$

We observe that the computation of y_2 can be avoided by evaluating λ_2 as

$$\lambda_2 = \frac{x_1^2}{x_2} + \lambda_1 + x_2 + 1. \tag{28}$$

As a result, computing $[4]P$ over binary fields requires $2[i] + 3[s] + 3[m]$. Compared to two consecutive doublings, it saves one field multiplication. Note that we are working in characteristic 2 and thus squarings are free or of negligible cost.

For the $\mathrm{QA}^{\mathcal{A}}$ operation, we evaluate $[4]P \pm Q$ as $[2]([2]P) \pm Q$ using one doubling ($\mathrm{DBL}^{\mathcal{A}}$) and one double-and-add ($\mathrm{DA}^{\mathcal{A}}$), resulting in $3[i]+3[s]+5[m]$. This is always better than applying the previous trick one more time by computing $((((P+Q)+P)+P)+P)$ in $4[i] + 4[s] + 5[m]$; or evaluating $[3]P + (P + Q)$ which requires $4[i] + 4[s] + 6[m]$.

In [12], Ciet et al. have improved an algorithm by Guajardo and Paar [25] for the computation of $[4]P$; their new method requires $1[i] + 5[s] + 8[m]$. Based on their costs, $\mathrm{QA}^{\mathcal{A}}$ is best evaluated as $([4]P) \pm Q$ using one quadrupling ($\mathrm{QPL}^{\mathcal{A}}$) followed by one addition ($\mathrm{ADD}^{\mathcal{A}}$) in $2[i] + 6[s] + 10[m]$. In Table 6 below, we summarize the costs and break-even points between our new formulae and the algorithms proposed in [12]. With such small break-even points, however, it remains unclear which formulae will give the best overall performance in practical situations.

TABLE 6. Tripling and quadrupling algorithms in affine coordinates for curves defined over $\mathbb{F}_{2^m}$

Operation	present work	[12]	break-even point
$\mathrm{TPL}^{\mathcal{A}}$	$1[i] + 3[s] + 6[m]$	$1[i] + 4[s] + 7[m]$	–
$\mathrm{QPL}^{\mathcal{A}}$	$2[i] + 3[s] + 3[m]$	$1[i] + 5[s] + 8[m]$	$[i]/[m] = 5$
$\mathrm{QA}^{\mathcal{A}}$	$3[i] + 3[s] + 5[m]$	$2[i] + 6[s] + 10[m]$	$[i]/[m] = 5$

3.3. **Curves defined over $\mathbb{F}_{2^m}$ using Jacobian coordinates.** As pointed out by Hankerson et al. in [26], affine coordinates might not be the best option for software implementations. In this section, we propose a tripling algorithm for curves defined over $\mathbb{F}_{2^m}$ using Jacobian coordinates. (We did not find an efficient tripling formula using Lopez-Dahad coordinates.)

Let us first recall the doubling formula. If $P = (X_1 : Y_1 : Z_1)$, we compute $[2]P = (X_2 : Y_2 : Z_2)$ as

$$\begin{aligned} X_2 &= B + a_6 C^4, \\ Z_2 &= X_1 C^2, \\ Y_2 &= BZ_2 + (A + D + Z_2)X_2, \end{aligned}$$

where $A = X_1^2$, $B = A^2$, $C = Z_1^2$ and $D = Y_1 Z_1$.

For the point tripling operation, we compute $[3]P = P + [2]P = (X_3 : Y_3 : Z_3)$ by deriving, for example, the addition formula from [2]. We easily obtain

$$Z_3 = (X_1 Z_2^2 + X_2 Z_1^2) Z_1 Z_2 = (X_1^3 Z_1^2 + X_2) Z_2 Z_1^3 = F Z_1^3,$$

with $E = (X_1 Z_1^2 + X_2)$ and $F = E Z_2$. We then compute X_3 as

$$\begin{aligned} X_3 &= a_2 Z_3^2 + (Y_1 Z_2^3 + Y_2 Z_1^3)(Y_1 Z_2^3 + Y_2 Z_1^3 + Z_3) + (X_1 Z_2^2 + X_2 Z_1^2)^3 \\ &= a_2 F^2 Z_1^6 + (Y_1 X_1^3 Z_1^6 + Y_2 Z_1^3)(Y_1 X_1^3 Z_1^6 + Y_2 Z_1^3 + F Z_1^3) + (X_1^3 Z_1^4 + X_2 Z_1^2)^3 \\ &= H Z_1^6, \end{aligned}$$

where $H = (a_2 F^2 + G(G+F) + E^3)$ and $G = (Y_1 X_1^3 Z_1^3 + Y_2)$. Finally, Y_3 can be computed as

$$\begin{aligned} Y_3 &= (Y_1 Z_2^3 + Y_2 Z_1^3 + Z_3) X_3 + ((X_1 Z_2^2 + X_2 Z_1^2) Z_1)^2 ((Y_1 Z_2^3 + Y_2 Z_1^3) X_2 \\ &\quad + (X_1 Z_1^2 + X_2 Z_1^2) Z_1 Y_2) \\ &= (Y_1 X_1^3 Z_1^3 + Y_2 + F) H Z_1^9 + (X_1 Z_1^2 + X_2)^2 ((X_1^3 Y_1 Z_1^3 + Y_2) X_2 \\ &\quad + (X_1^3 Z_1^2 + X_2) Y_2) Z_1^9 \\ &= I Z_1^9, \end{aligned}$$

where $I = (G+F)H + E^2(GX_2 + EY_2)$.

Using the fact that $[3]P = (X_3 : Y_3 : Z_3) = (HZ_1^6 : IZ_1^9 : FZ_1^3) = (H : I : F)$, the operation count is $15[m] + 7[s]$, including two multiplications by a_2 and a_6. In terms of memory, it requires 12 registers. We note that computing $[3]P$ as $[2]P + P$; i.e. using one doubling followed by one addition, would cost $21[m] + 8[s]$. We give an atomic version of this algorithm in Table 17 of Appendix B.

4. Scalar multiplication and double-base chains

In this section, we present a generic scalar multiplication algorithm which takes advantage of the properties of the double-base number and the efficient curve formulae presented in the previous sections. This generic algorithm can be easily adapted to different cases; we give the complexities for curves defined over $\mathbb{F}_p$ using Jacobian coordinates and for curves defined over $\mathbb{F}_{2^m}$ using both affine and Jacobian coordinates.

Everything would be easy and we would have nothing else to say if it was possible to use the greedy algorithm presented in Section 2.1 for the conversion. Unfortunately, in order to reduce the number of doublings and/or triplings, our algorithm requires the scalar n to be represented in a particular double-base form. More precisely, we need to express $n > 0$ as $n = \sum_{i=1}^{l} s_i 2^{a_i} 3^{b_i}$, with $s_i \in \{-1, 1\}$, where the exponents form two decreasing sequences; i.e., $a_1 \geq a_2 \geq \cdots \geq a_l \geq 0$ and $b_1 \geq b_2 \geq \cdots \geq b_l \geq 0$. More formally, we endow the set of $\{2,3\}$-integers with the order $\precsim$ induced by the product order on $\mathbb{N}^2$:

$$2^a 3^b \precsim 2^{a'} 3^{b'} \Leftrightarrow a \leq a', b \leq b'. \tag{29}$$

These particular DBNS representations allow us to expand n in a Horner-like fashion such that all partial results can be reused during the computation of $[n]P$. In fact, such a double-base expansion for n defines a double-base chain computing n.

Definition 4 (Double-base chain)**.** Given $n > 0$, a sequence $(C_i)_{i>0}$ of positive integers satisfying:

$$C_1 = 1, \quad C_{i+1} = 2^{u_i} 3^{v_i} C_i + s, \text{ with } s \in \{-1, 1\} \tag{30}$$

for some $u_i, v_i \geq 0$, and such that $C_l = n$ for some $l > 0$, is called a double-base chain computing n. The length l of a double-base chain is equal to the number of $\{2,3\}$-integers in (1) used to represent the integer n.

Note that it is always possible to find a double-base chain computing n; the binary representation is a special case. In fact, this particular DBNS representation is also highly redundant. Counting the exact number of DBNS representations which satisfy these conditions is per se a very intriguing problem. Let $g(n)$ denote the number of (unsigned) double-base chains computing n. Clearly, since the binary expansion is a trivial case, one has $g(3n) \geq 1 + g(u)$, and thus $g(3^n) \geq n + 1$. If $\overline{g}(n) = \frac{1}{n}\sum_{t=0}^{u} g(t)$, we conjecture that, for large n, one has $\log n < \overline{g}(n)$; and maybe $\lim_{n\to\infty} \frac{\log n}{\overline{g}(n)} = 0$. Moreover, it is possible to prove that $g(n) = 1$, if and only if, either $n \in \{0, 1, 2\}$ or $n = 2^a 3 - 1$, for $a \geq 1$.

If necessary, such a specific DBNS representation of any w-bit positive integer n can be computed using Algorithm 2 below; a modified version of the greedy algorithm which takes into account the order $\precsim$ on the exponents.

Algorithm 2 Greedy algorithm with restricted exponents

Input n, a w-bit positive integer; $a_{max}, b_{max} > 0$, the largest allowed binary and ternary exponents

Output The sequence $(s_i, a_i, b_i)_{i>0}$ such that $n = \sum_{i=1}^{l} s_i 2^{a_i} 3^{b_i}$, with $2^{a_i} 3^{b_i} \precsim 2^{a_{i-1}} 3^{b_{i-1}}$ for $i \geq 1$

1: $s \leftarrow 1$
2: **while** $n > 0$ **do**
3: define $z = 2^a 3^b$, the best approximation of n with $0 \leq a \leq a_{max}$ and $0 \leq b \leq b_{max}$
4: **print** (s, a, b)
5: $a_{max} \leftarrow a, \quad b_{max} \leftarrow b$
6: **if** $n < z$ **then**
7: $s \leftarrow -s$
8: $n \leftarrow |n - z|$

Two important parameters of this algorithm are the upper bounds for the binary and ternary exponents in the expansion of n, called a_{max} and b_{max}, respectively. Clearly, we have $a_{max} < \log_2(n) < w$ and $b_{max} < \log_3(n) \approx 0.63w$. Our experiments showed that using these utmost values for a_{max} and b_{max} does not result in short expansions. Indeed, when the best approximation of a given integer of the form $2^a 3^b$ is either close to a power of 2 (i.e. b is small) or close to a power of 3 (i.e. a is small), the resulting double-base chains are likely to be the binary or the balanced ternary expansions. We want to avoid this phenomenon by selecting a_{max}, b_{max} such that $2^{a_{max}} 3^{b_{max}}$ is slightly greater than n, and a_{max} is not too large/small compared to b_{max}. The optimal values for a_{max}, b_{max} seem difficult to determine in the general case as they clearly depend (but not only) on the relative cost between the doubling and the tripling operations. Instead, we consider the following heuristic which leads to good results in practice: if $n = (n_{w-1} \dots n_1 n_0)_2$ is a randomly chosen w-bit integer (i.e. $n_{w-1} \neq 0$), we initially set $a_{max} = x$ and $b_{max} = y$, where $2^x 3^y$ is a very good, non-trivial (i.e. $y \neq 0$) approximation of 2^w. Then, in order to get sequences of exponents satisfying the conditions $2^{a_i} 3^{b_i} \precsim 2^{a_{i-1}} 3^{b_{i-1}}$ for $i \geq 1$, the new largest exponents are updated according to the values of a and b obtained in step 3.

We can now present a generic point multiplication algorithm which can be easily adapted to various cases depending on the field over which the curve is defined and the curve operations we have at our disposal.

Algorithm 3 Generic DBNS Scalar Multiplication

Input An integer $n = \sum_{i=1}^{l} s_i 2^{a_i} 3^{b_i}$, with $s_i \in \{-1, 1\}$, and such that $2^{a_i} 3^{b_i} \precsim 2^{a_{i-1}} 3^{b_{i-1}}$ for $i \geq 1$; and a point $P \in E(K)$
Output the point $[n]P \in E(K)$
1: $Q \leftarrow [s_1]P$
2: **for** $i = 1, \ldots, l-1$ **do**
3: $\quad u_i \leftarrow a_i - a_{i+1}, \quad v_i \leftarrow b_i - b_{i+1}$
4: $\quad Q \leftarrow [3^{v_i}]Q$
5: $\quad Q \leftarrow [2^{u_i}]Q$
6: $\quad Q \leftarrow Q + [s_{i+1}]P$
7: **Return** Q

The complexity of Algorithm 3 depends on the number of doublings, triplings and mixed additions that have to be performed: the total number of additions is equal to the length l of the double-base expansion of n, and the number of doublings and triplings are equal to $a_1 \leq a_{max}$ and $b_1 \leq b_{max}$, respectively. However, the complexity can be more precisely evaluated if one considers the exact cost of each iteration, by counting the exact number of field operations (inversions, multiplications and squarings) required in steps 4 to 6.

In fact, given $n > 0$, Algorithm 3 immediately gives us a double-base chain for n. Let W_i be the exact number of curve operations required to compute $[C_i]P$ from $[C_{i-1}]P$. We clearly have $C_1 = 1$ and $W_1 = 0$ (we set Z to P or $-P$ at no cost in step 1). Hence, the total cost for computing $[n]P$ from input point P is given by

$$W_l = \sum_{i=1}^{l} W_i. \tag{31}$$

In the next three sections, we consider three cases: curves defined over $\mathbb{F}_p$ using Jacobian coordinates, and curves defined over $\mathbb{F}_{2^m}$ using both affine and Jacobian coordinates. Extensions to other cases, for example for curves defined over fields of characteristic three, can be easily derived.

4.1. **Curves defined over $\mathbb{F}_p$ with Jacobian coordinates.** In this case, steps 4 and 5 can be implemented using the w-TPL$^{\mathcal{J}}$/w'-DBL$^{\mathcal{J}}$ operation presented in Section 4.1. When $u_i = 0$ or $v_i = 0$ in Step 3, w-TPL$^{\mathcal{J}}$ or w-DBL$^{\mathcal{J}}$ are called instead. The addition is always a mixed addition (ADD$^{\mathcal{J}+\mathcal{A}}$). We have:

$$W_i = v_i\text{-TPL}^{\mathcal{J}}/u_i\text{-DBL}^{\mathcal{J}} + \text{ADD}^{\mathcal{J}+\mathcal{A}}. \tag{32}$$

The total cost is given by (31).

4.2. **Curves defined over $\mathbb{F}_{2^m}$ with Jacobian coordinates.** In this case we did not find any way to save some operations for consecutive doublings and/or triplings. The addition is always a mixed addition. We have:

$$W_i = v_i \times \text{TPL}^{\mathcal{J}} + u_i \times \text{DBL}^{\mathcal{J}} + \text{ADD}^{\mathcal{J}+\mathcal{A}}. \tag{33}$$

Again, the total cost is given by (31).

4.3. **Curves defined over $\mathbb{F}_{2^m}$ with affine coordinates.** In this case, the algorithm can be further optimized in order to take advantage of the quadrupling and combined quadruple-and-add algorithms presented in Section 3.2. Algorithm 4 below is an adaptation of our generic algorithm.

Algorithm 4 DBNS scalar multiplication for curves over $\mathbb{F}_{2^m}$ using affine coordinates

Input An integer $n = \sum_{i=1}^{l} s_i\, 2^{a_i} 3^{b_i}$, with $s_i \in \{-1, 1\}$, and such that $2^{a_i} 3^{b_i} \precsim 2^{a_{i-1}} 3^{b_{i-1}}$ for $i \geq 1$; and a point $P \in E(K)$

Output the point $[n]P \in E(K)$

1: $Q \leftarrow [s_1]P$
2: **for** $i = 1, \ldots, l-1$ **do**
3: $\quad u_i \leftarrow a_i - a_{i+1}, \quad v_i \leftarrow b_i - b_{i+1}$
4: $\quad$ **if** $u_i = 0$ **then**
5: $\quad\quad Q \leftarrow [3]([3^{v_i - 1}]Q) + [s_{i+1}]P$
6: $\quad$ **else**
7: $\quad\quad Q \leftarrow [3^{v_i}]Q$
8: $\quad\quad Q \leftarrow [4^{\lfloor (u_i - 1)/2 \rfloor}]Q$
9: $\quad\quad$ **if** $u_i \equiv 0 \pmod 2$ **then**
10: $\quad\quad\quad Q \leftarrow [4]Q + [s_{i+1}]P$
11: $\quad\quad$ **else**
12: $\quad\quad\quad Q \leftarrow [2]Q + [s_{i+1}]P$
13: **Return** Q

We remark that although $l - 1$ additions are required to compute $[n]P$, we never actually use the addition operation (ADD$^{\mathcal{A}}$); simply because we combine each addition with either a doubling (step 13), a tripling (step 6) or a quadrupling (step 11), using the DA$^{\mathcal{A}}$, TA$^{\mathcal{A}}$ and QA$^{\mathcal{A}}$ operations. Note also that the TA$^{\mathcal{A}}$ operation for computing $[3]P \pm Q$ is only used in step 6, when $u_i = 0$. Another approach of similar cost is to start with all the quadruplings plus one possible doubling when u_i is odd, and then perform $v_i - 1$ triplings followed by one final triple-and-add.

The expression for W_i is a little more complicated; we have:

$$W_i = \delta_{u_i,0}\left((v_i - 1)\,T + TA\right) + (1 - \delta_{u_i,0})\left(v_i\,T + \left\lfloor \frac{u_i - 1}{2} \right\rfloor Q + \delta_{|u_i|_2,0}\,QA + \delta_{|u_i|_2,1}\,DA\right), \tag{34}$$

where $\delta_{i,j}$ is the Kronecker delta such that $\delta_{i,j} = 1$ if $i = j$ and $\delta_{i,j} = 0$ if $i \neq j$, and $|u_i|_2$ denotes $u_i \bmod 2$.

5. Comparisons and experimental results

In this section, we illustrate the efficiency of the proposed variants of the generic algorithm by providing experimental results and comparisons with classical methods (double-and-add, NAF, w-NAF) and some recently proposed algorithms: a ternary/binary approach from [12] for curves defined over binary fields using affine coordinates; and two algorithms from Izu et al. published in [29] and [31] for curves

defined over prime fields. In the latter, we consider the protected version of our algorithm, combined with Joye and Tymen's randomization technique to counteract differential attacks [32].

If we assume that n is a randomly chosen integer, it is well known that the double-and-add algorithm requires $\log n$ doublings and $\log n/2$ additions on average. Using the NAF representation, the average density of non-zero digits is reduced to 1/3. More generally, for w-NAF methods, the average number of non-zero digits is roughly equal to $\log n/(w+1)$. Unfortunately, it seems very difficult to give such a theoretical estimate for double-base chains. When the exponents do not have to satisfy any other conditions than being positive integers, it can be proved [18] that the greedy algorithm returns expansions of length $O(\log n/\log\log n)$. However, for double-base chains, the rigorous determination of this complexity leads to tremendously difficult problems in transcendental number theory and exponential Diophantine equations and is still an open problem. Therefore, in order to estimate the average number of $\{2,3\}$-integers required to represent n, and to precisely evaluate the complexity of our point multiplication algorithms, we have performed several numerical experiments, over 10000 randomly chosen 160-bit integers. The results are presented below.

5.1. **Curves defined over $\mathbb{F}_p$ with Jacobian coordinates.** We report results for 160-bit integers. If the classic methods are used in conjunction with side-channel atomicity (which implies $[s]=[m]$), the average cost of the double-and-add method can be estimated to $159\times 10+80\times 11=2470[m]$; similarly, the NAF and 4-NAF methods require, on average, $2173[m]$ and $1942[m]$, respectively. The results of our numerical experiments in the case of double-base chains are presented in Table 7.

TABLE 7. Average complexity of our scalar multiplication algorithm obtained using 10000 randomly chosen 160-bit integers for different values a_{max}, b_{max} and curves defined over $\mathbb{F}_p$ using Jacobian coordinates

a_{max}	b_{max}	l	Complexity	$[s]=[m]$	$[s]=0.8[m]$
57	65	44.09	$758.25[s]+1236.60[m]$	1994.86	1843.20
76	53	37.23	$770.23[s]+1132.45[m]$	1902.69	1748.64
95	41	36.63	$812.24[s]+1072.48[m]$	1884.73	1722.28
103	36	38.39	$840.44[s]+1061.33[m]$	1901.78	1733.69

In order to compare our algorithm with the side-channel resistant algorithms presented in [29, 31, 30], we also give the uniform costs in terms of the equivalent number of field multiplications in the last two columns. Note that, if side-channel atomicity is used to prevent simple analysis, squarings cannot be optimized and must be computed using a general multiplier; one must therefore consider $[s]=[m]$. In the last column of Table 7, we also give the complexity in terms of the equivalent number of multiplications assuming $[s]=0.8[m]$.

In Table 8, we summarize the costs of several scalar multiplication algorithms. In order to present fair comparisons, we add the extra cost of Joye and Tymen's randomization technique ($41[m]$ assuming $[i]=30[m]$) which can be used to resist differential analysis. The figures for the algorithms from Izu, Möller and Takagi are taken from [29] and [31] assuming Coron's randomization technique which turns

out to be more efficient in their case. The cost of our algorithm is taken from the third row of Table 7, with $a_{max} = 95$ and $b_{max} = 41$, as these values lead to the best operation count.

TABLE 8. Comparison of different scalar multiplication algorithms protected against simple and differential analysis

Algorithm	Complexity (#[m])
double-and-add	2511
NAF	2214
4-NAF	1983
Izu, Möller, Takagi 2002 [29]	2449
Izu, Takagi 2005 [31]	2629
DBNS	1926

We remark that the DBNS algorithm requires fewer operations than the other methods. It represents a gain of 23.29% over the double-and-add, 13% over the NAF, 2.8% over 4-NAF, 21.35% over [29] and 26.7% over [31]. Moreover, it does not require precomputations like the 4-NAF and the algorithms from Izu et al.

5.2. **Curves defined over $\mathbb{F}_{2^m}$ with Jacobian coordinates.** As noticed in Section 3.3, the cost of our tripling is almost equivalent to that of one doubling followed by a mixed addition. From our numerical experiments, we remark that the average length l of the double-base chains obtained with the greedy algorithm for different values a_{max}, b_{max} lies between $2\log n/9$ and $\log n/3$. Unfortunately, with such an efficient doubling formula, even the shortest double-base chains result in too many additions (about 40 for 160-bit scalar) and too many triplings to defeat algorithms based on doublings only. Therefore, an optimized algorithm is very likely to behave like the NAF algorithm which only uses doublings and mixed additions; on average the NAF requires $5[m] \times 159 + 11[m] \times 53 = 1378[m]$. This is confirmed by our numerical results presented in Table 9. The best results are not obtained for shortest chains but for the expansions which minimize the number of triplings (and maximize the number of doublings).

TABLE 9. Average complexity of our DBNS point multiplication algorithm obtained using 10000 randomly chosen 160-bit integers for different values a_{max}, b_{max} and curves defined over $\mathbb{F}_{2^m}$ using Jacobian coordinates

a_{max}	b_{max}	l	Complexity (#[m] only)
57	65	44.09	$1708[m]$
76	53	37.23	$1566[m]$
95	41	36.63	$1478[m]$
103	36	38.39	$1459[m]$
156	3	52.41	$1374[m]$
159	1	53.10	$1370[m]$

For curves over $\mathbb{F}_{2^m}$ and Jacobian coordinates, the double-base approach will thus only become a serious alternative if one can find a better tripling formula, or an algorithm leading to shorter double-base chains.

5.3. **Curves defined over $\mathbb{F}_{2^m}$ with affine coordinates.** We summarize our experimental results and the comparisons with other classical methods in Table 10. These results have been obtained with the curve operations that have the best complexity when the ratio $[i]/[m]$ is small. Indeed, when the relative cost of an inversion increases, inversion-free coordinates become rapidly more interesting (see [26]).

For completeness, we also give the operation counts for a recent ternary/binary algorithm presented in [12], which is based on the following recursive decomposition: if $n \equiv 0$ or $3 \pmod 6$, return $[3]([n/3]P)$; if $n \equiv 2$ or $4 \pmod 6$, return $[2]([k/2]P)$; if $n \equiv 1 \pmod 6$, i.e., $n = 6m + 1$, return $[2]([3m]P) + P$; if $n \equiv 5 \pmod 6$, i.e., $n = 6m - 1$, return $[2]([3m]P) - P$. The recursion stops whenever $n = 1$ and returns P. Applying this recursive decomposition to any positive scalar n leads, of course, to a double-base chain which does satisfy the requirements that the ternary and binary exponents form two decreasing sequences. But, thanks to the huge redundancy of the DBNS, it is possible to seek better representations having the same properties on the exponents and, at the same time, leading to shorter chains and reduced complexity.

TABLE 10. Average complexity and comparisons with other methods of our DBNS point multiplication algorithm obtained using 10000 randomly chosen 160-bit integers and different values a_{max}, b_{max} for curves defined over $\mathbb{F}_{2^m}$ using affine coordinates

a_{max}	b_{max}	l	Complexity	$[i] = 3[m]$	$[i] = 10[m]$
57	65	44.09	$228.69[i] + 242.32[s] + 335.66[m]$	1022	2066
76	53	37.23	$216.17[i] + 235.09[s] + 324.84[m]$	973	1932
95	41	36.63	$211.74[i] + 235.86[s] + 322.38[m]$	958	1898
103	36	38.39	$211.15[i] + 237.50[s] + 322.46[m]$	956	1906
Double-and-add			$244.31[i] + 244.82[s] + 407.09[m]$	1139	2847
NAF			$217.22[i] + 217.91[s] + 380.63[m]$	1031	2550
Ternary/Binary			$222.11[i] + 222.84[s] + 353.04[m]$	1019	2573

We remark that our algorithm requires fewer inversions and multiplications than the other methods. In order to clarify the comparison, we report, in the last two columns of Table 10, the cost in terms of the equivalent number of multiplications assuming $[i] = 3[m]$ and $[i] = 10[m]$. In the first case, our algorithm represents a speed-up of about 16% over the double-and-add, 7% over the NAF and 6% over the ternary/binary approach proposed in [12]. In the more realistic case (for software implementations) $[i] = 10[m]$, the speed-ups are even more important; 33% over the double-and-add, 25% over the NAF and 26% over the ternary/binary approach.

6. CONCLUSIONS

In this paper, we proposed several variants of a generic point multiplication algorithm based on the representation of n as $\sum_i \pm 2^{a_i} 3^{b_i}$. Among many nice properties, this representation scheme, called the double-base number system, offers the advantage of being very sparse. In the context of our scalar multiplication algorithm, the extra condition that the sequences of exponents must decrease does not allow us to claim sublinearity for the length of the double-base chains. However,

we provided extensive numerical evidence that demonstrates the efficiency of this approach compared to existing methods of similar nature.

Acknowledgments

The authors are very thankful to the anonymous reviewers for their very useful comments and for suggesting to us the improved tripling formula in affine coordinates for curves over binary fields. They also thank Nicolas Meloni for the improved tripling formula in Jacobian coordinates for curves over binary fields.

Appendix A. Curves defined over $\mathbb{F}_p$ using Jacobian coordinates

In this appendix, we give the algorithms for $\text{DBL}^{\mathcal{J}}$ (including the case when a doubling is performed right after a tripling), w-$\text{DBL}^{\mathcal{J}}$, $\text{TPL}^{\mathcal{J}}$ and w-$\text{TPL}^{\mathcal{J}}$, expressed in atomic blocks, for curves defined over $\mathbb{F}_p$, with Jacobian coordinates.

Table 11. The $\text{DBL}^{\mathcal{J}}$ algorithm in atomic blocks. When $\text{DBL}^{\mathcal{J}}$ is called right after w-$\text{TPL}^{\mathcal{J}}$, the blocks Δ_2, Δ_3 and Δ_4 can be replaced by the blocks Δ_2' and Δ_3' to save one multiplication

$\text{DBL}^{\mathcal{J}}$ / $\mathbb{F}_p$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1)$
Output: $[2]P = (X_3 : Y_3 : Z_3) = (R_1 : R_2 : R_3)$
Init: $R_1 = X_1,\ R_2 = Y_1,\ R_3 = Z_1$

Δ_1	$R_4 = R_1 \times R_1$	(X_1^2)	Δ_6	$R_2 = R_2 \times R_2$	(Y_1^2)
	$R_5 = R_4 + R_4$	$(2X_1^2)$		$R_2 = R_2 + R_2$	$(2Y_1^2)$
	$*$			$*$	
	$R_4 = R_4 + R_5$	$(3X_1^2)$		$*$	
Δ_2	$R_5 = R_3 \times R_3$	(Z_1^2)	Δ_7	$R_5 = R_1 \times R_2$	(S)
	$R_1 = R_1 + R_1$	$(2X_1)$		$*$	
	$*$			$R_5 = -R_5$	$(-S)$
	$*$			$*$	
Δ_3	$R_5 = R_5 \times R_5$	(Z_1^4)	Δ_8	$R_1 = R_4 \times R_4$	(M^2)
	$*$			$R_1 = R_1 + R_5$	$(M^2 - S)$
	$*$			$*$	
	$*$			$R_1 = R_1 + R_5$	(X_3)
Δ_4	$R_6 = a \times R_5$	(aZ_1^4)	Δ_9	$R_2 = R_2 \times R_2$	$(4Y_1^4)$
	$R_4 = R_4 + R_6$	(M)		$R_7 = R_2 + R_2$	(T)
	$*$			$*$	
	$R_5 = R_2 + R_2$	$(2Y_1)$		$R_5 = R_1 + R_5$	$(X_3 - S)$
Δ_5	$R_3 = R_3 \times R_5$	(Z_3)	Δ_{10}	$R_4 = R_4 \times R_5$	$(M(X_3 - S))$
	$*$			$R_2 = R_4 + R_7$	$(-Y_3)$
	$*$			$R_2 = -R_2$	(Y_3)
	$*$			$*$	

Δ_2'	$R_5 = R_{10} \times R_{10}$	Δ_3'	$R_5 = R_5 \times R_9$
	$R_1 = R_1 + R_1$		$R_4 = R_4 + R_6$
	$*$		$*$
	$*$		$*$

TABLE 12. The w-DBL$^{\mathcal{J}}$ algorithm in atomic blocks. The 10 blocks (or 9 if executed after w-TPL$^{\mathcal{J}}$) of DBL$^{\mathcal{J}}$ (Table 11) must be executed once, followed by the blocks Δ_{11} to Δ_{18} which have to be executed $w-1$ times. After the execution of DBL$^{\mathcal{J}}$, the point of coordinates $(X_t : Y_t : Z_t)$ correspond to the point $[2]P$. After $w-1$ iterations $[2^w]P = (X_3 : Y_3 : Z_3) = (X_t : Y_t : Z_t)$

w-DBL$^{\mathcal{J}}$ / $\mathbb{F}_p$ / Jacobian

Input: $P = (X_1 : Y_1 : Z_1)$

Output: $[2^w]P = (R_1 : R_2 : R_3)$

Init: $(X_t : Y_t : Z_t)$ is the result of DBL$^{\mathcal{J}}(P)$, $R_6 = aZ_1^4$, $R_7 = 8Y_1^4$

Δ_{11}	$R_4 = R_1 \times R_1$	(X_t^2)	Δ_{15}	$R_5 = R_1 \times R_2$	(S)
	$R_5 = R_4 + R_4$	$(2X_t^2)$		$*$	
	$*$			$R_5 = -R_5$	$(-S)$
	$R_4 = R_4 + R_5$	$(3X_t^2)$		$*$	
Δ_{12}	$R_5 = R_6 \times R_7$	$(aZ_t^4 + 8Y_t^4)$	Δ_{16}	$R_1 = R_4 \times R_4$	(M^2)
	$R_6 = R_5 + R_5$	(aZ_t^4)		$R_1 = R_1 + R_5$	$(M^2 - S)$
	$*$			$*$	
	$R_4 = R_4 + R_6$	(M)		$R_1 = R_1 + R_5$	(X_{t+1})
Δ_{13}	$R_3 = R_2 \times R_3$	$(Y_t Z_t)$	Δ_{17}	$R_2 = R_2 \times R_2$	$(4Y_t^4)$
	$R_3 = R_3 + R_3$	(Z_{t+1})		$R_7 = R_2 + R_2$	(T)
	$*$			$*$	
	$R_1 = R_1 + R_1$	$(2X_t)$		$R_5 = R_1 + R_5$	$(X_{t+1} - S)$
Δ_{14}	$R_2 = R_2 \times R_2$	(Y_t^2)	Δ_{18}	$R_4 = R_4 \times R_5$	$(M(X_{t+1} - S))$
	$R_2 = R_2 + R_2$	$(2Y_t^2)$		$R_2 = R_4 + R_7$	$(-Y_{t+1})$
	$*$			$R_2 = -R_2$	(Y_{t+1})
	$*$			$*$	

TABLE 13. The TPL$^{\mathcal{J}}$ algorithm in atomic blocks

TPL$^{\mathcal{J}}$ / $\mathbb{F}_p$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1)$
Output: $[3]P = (X_3 : Y_3 : Z_3) = (R_1 : R_2 : R_3)$
Init: $R_1 = X_1,\ R_2 = Y_1,\ R_3 = Z_1$

Γ_1	$R_4 = R_3 \times R_3$	(Z_1^2)	Γ_9	$R_8 = R_6 \times R_7$	(T)
	$*$			$R_7 = R_7 + R_7$	$(8Y_1^2)$
	$*$			$*$	
	$*$			$*$	
Γ_2	$R_4 = R_4 \times R_4$	(Z_1^4)	Γ_{10}	$R_6 = R_4 \times R_5$	(ME)
	$*$			$*$	
	$*$			$R_6 = -R_6$	$(-ME)$
	$*$			$R_6 = R_8 + R_6$	$(T - ME)$
Γ_3	$R_5 = R_1 \times R_1$	(X_1^2)	Γ_{11}	$R_{10} = R_5 \times R_5$	(E^2)
	$R_6 = R_5 + R_5$	$(2X_1^2)$		$*$	
	$*$			$*$	
	$R_5 = R_5 + R_6$	$(3X_1^2)$		$*$	
Γ_4	$R_9 = a \times R_4$	(aZ_1^4)	Γ_{12}	$R_1 = R_1 \times R_{10}$	(X_1E^2)
	$R_4 = R_5 + R_9$	(M)		$*$	
	$*$			$*$	
	$*$			$*$	
Γ_5	$R_5 = R_2 \times R_2$	(Y_1^2)	Γ_{13}	$R_5 = R_{10} \times R_5$	(E^3)
	$R_6 = R_5 + R_5$	$(2Y_1^2)$		$R_8 = R_8 + R_6$	$(2T - ME)$
	$*$			$R_5 = -R_5$	$(-E^3)$
	$R_7 = R_6 + R_6$	$(4Y_1^2)$		$*$	
Γ_6	$R_5 = R_1 \times R_7$	$(4X_1Y_1^2)$	Γ_{14}	$R_4 = R_6 \times R_7$	$8Y_1^2(T - ME)$
	$R_8 = R_5 + R_5$	$(8X_1Y_1^2)$		$R_6 = R_6 + R_6$	$(2(T - ME))$
	$*$			$R_6 = -R_6$	$(2(ME - T))$
	$R_5 = R_5 + R_8$	$(12X_1Y_1^2)$		$R_1 = R_1 + R_4$	(X_3)
Γ_7	$R_8 = R_4 \times R_4$	(M^2)	Γ_{15}	$R_6 = R_6 \times R_8$	
	$*$			$R_6 = R_6 + R_6$	
	$R_8 = -R_8$	$(-M^2)$		$*$	
	$R_5 = R_5 + R_8$	(E)		$R_6 = R_6 + R_5$	
Γ_8	$R_3 = R_3 \times R_5$	(Z_3)	Γ_{16}	$R_2 = R_2 \times R_6$	(Y_3)
	$*$			$*$	
	$*$			$*$	
	$*$			$*$	

TABLE 14. The w-TPL$^{\mathcal{J}}$ algorithm in atomic blocks. The 16 blocks of TPL$^{\mathcal{J}}$ must be executed once, followed by the blocks Γ_{17} to Γ_{31} which have to be executed $w-1$ times. After the execution of TPL$^{\mathcal{J}}$, the point of coordinates $(X_t : Y_t : Z_t)$ correspond to the point $[3]P$; at the end of the $w-1$ iterations, $[3^w]P = (X_3 : Y_3 : Z_3) = (X_t : Y_t : Z_t)$

w-TPL$^{\mathcal{J}}$ / $\mathbb{F}_p$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1)$
Output: $[3^w]P = (R_1 : R_2 : R_3)$
Init: $(X_t : Y_t : Z_t)$ is the result of TPL$^{\mathcal{J}}(P)$, $R_9 = aZ_1^4$, $R_{10} = E^2$

Γ_{17}	$R_4 = R_9 \times R_{10}$	$(aZ_t^4E^2)$	Γ_{25}	$R_6 = R_4 \times R_5$	(ME)
	$*$			$*$	
	$*$			$R_6 = -R_6$	$(-ME)$
	$*$			$R_6 = R_8 + R_6$	$(T - ME)$
Γ_{18}	$R_5 = R_1 \times R_1$	(X_t^2)	Γ_{26}	$R_{10} = R_5 \times R_5$	(E^2)
	$R_6 = R_5 + R_5$	$(2X_t^2)$		$*$	
	$*$			$*$	
	$R_5 = R_5 + R_6$	$(3X_t^2)$		$*$	
Γ_{19}	$R_9 = R_4 \times R_{10}$	(aZ_t^4)	Γ_{27}	$R_1 = R_1 \times R_{10}$	(X_tE^2)
	$R_4 = R_5 + R_9$	(M)		$*$	
	$*$			$*$	
	$*$			$*$	
Γ_{20}	$R_5 = R_2 \times R_2$	(Y_t^2)	Γ_{28}	$R_5 = R_{10} \times R_5$	(E^3)
	$R_6 = R_5 + R_5$	$(2Y_t^2)$		$R_8 = R_8 + R_6$	$(2T - ME)$
	$*$			$R_5 = -R_5$	$(-E^3)$
	$R_7 = R_6 + R_6$	$(4Y_t^2)$		$*$	
Γ_{21}	$R_5 = R_1 \times R_7$	$(4X_tY_t^2)$	Γ_{29}	$R_4 = R_6 \times R_7$	$(8Y_t^2(T - ME))$
	$R_8 = R_5 + R_5$	$(8X_tY_t^2)$		$R_6 = R_6 + R_6$	$(2(T - ME))$
	$*$			$R_6 = -R_6$	$(2(ME - T))$
	$R_5 = R_5 + R_8$	$(12X_tY_t^2)$		$R_1 = R_1 + R_4$	(X_{t+1})
Γ_{22}	$R_8 = R_4 \times R_4$	(M^2)	Γ_{30}	$R_6 = R_6 \times R_8$	
	$*$			$R_6 = R_6 + R_6$	
	$R_8 = -R_8$	$(-M^2)$		$*$	
	$R_5 = R_5 + R_8$	(E)		$R_6 = R_6 + R_5$	
Γ_{23}	$R_3 = R_3 \times R_5$	(Z_{t+1})	Γ_{31}	$R_2 = R_2 \times R_6$	(Y_{t+1})
	$*$			$*$	
	$*$			$*$	
	$*$			$*$	
Γ_{24}	$R_8 = R_6 \times R_7$	(T)			
	$R_7 = R_7 + R_7$	$(8Y_t^2)$			
	$*$				
	$*$				

APPENDIX B. CURVES DEFINED OVER $\mathbb{F}_{2^m}$ USING JACOBIAN COORDINATES

In this appendix, we give the algorithms for DBL$^{\mathcal{J}}$, ADD$^{\mathcal{J}+\mathcal{A}}$ and TPL$^{\mathcal{J}}$, expressed in atomic blocks, for curves defined over $\mathbb{F}_{2^m}$, with Jacobian coordinates. We consider (s, s, m, a)-blocks to avoid the use of 6 blocks for the doubling. Note that the mixed addition and the tripling can be expressed in 11 and 16 (s, m, a)-blocks respectively.

TABLE 15. The $\mathrm{DBL}^{\mathcal{J}}$ algorithm for curves over $\mathbb{F}_{2^m}$ using Jacobian coordinates

$\mathrm{DBL}^{\mathcal{J}}$ / $\mathbb{F}_{2^m}$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1)$
Output: $[2]P = (X_3 : Y_3 : Z_3) = (R_1 : R_2 : R_3)$
Init: $R_1 = X_1,\ R_2 = Y_1,\ R_3 = Z_1$

Δ_1	$R_4 = R_1^2$	(X_1^2)	Δ_4	$*$	
	$R_5 = R_4^2$	$(X1^4)$		$*$	
	$R_2 = R_2 \times R_3$	(Y_1Z_1)		$R_3 = R_5 \times R_3$	$(X_1^4Z_3)$
	$R_2 = R_4 + R_2$	$(X_1^2 + Y_1Z_1)$		$*$	
Δ_2	$R_3 = R_3^2$	(Z_1^2)	Δ_5	$*$	
	$R_4 = R_3^2$	(Z_1^4)		$*$	
	$R_3 = R_1 \times R_3$	(Z_3)		$R_2 = R_2 \times R_1$	
	$R_2 = R_2 + R_3$			$R_2 = R_3 + R_2$	(Y_3)
Δ_3	$R_1 = R_4^2$	(Z_1^8)			
	$*$				
	$R_1 = a_6 \times R_1$	$(a_6Z_1^8)$			
	$R_1 = R_5 + R_1$	(X_3)			

TABLE 16. The $\mathrm{ADD}^{\mathcal{J}+\mathcal{A}}$ algorithm for curves over $\mathbb{F}_{2^m}$ using Jacobian coordinates

$\mathrm{ADD}^{\mathcal{J}+\mathcal{A}}$ / $\mathbb{F}_{2^m}$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1),\ Q = (X_2 : Y_2 : 1)$
Output: $P + Q = (X_3 : Y_3 : Z_3) = (R_4 : R_5 : R_3)$
Init: $R_1 = X_1,\ R_2 = Y_1,\ R_3 = Z_1,\ R_4 = X_2,\ R_5 = Y_2$

Δ_1	$R_6 = R_3^2$	(Z_1^2)	Δ_7	$*$	
	$*$			$*$	
	$R_7 = R_4 \times R_6$	(B)		$R_4 = a_2 \times R_8$	$(a_2Z_3^2)$
	$R_1 = R_1 + R_7$	(E)		$*$	
Δ_2	$R_7 = R_1^2$	(E^2)	Δ_8	$*$	
	$*$			$*$	
	$R_6 = R_6 \times R_3$	(Z_1^3)		$R_2 = R_2 \times R_6$	(FI)
	$*$			$R_4 = R_4 + R_2$	$(a_2Z_3^2 + FI)$
Δ_3	$*$		Δ_9	$*$	
	$*$			$*$	
	$R_6 = R_6 \times R_5$	$(Y_2Z_1^3)$		$R_2 = R_7 \times R_1$	(E^3)
	$R_2 = R_2 + R_6$	(F)		$R_4 = R_4 + R_2$	(X_3)
Δ_4	$*$		Δ_{10}	$*$	
	$*$			$*$	
	$R_3 = R_1 \times R_3$	(Z_3)		$R_4 = R_6 \times R_4$	(IX_3)
	$R_6 = R_2 + R_3$	(I)		$*$	
Δ_5	$R_8 = R_3^2$	(Z_3^2)	Δ_{11}	$*$	
	$*$			$*$	
	$R_4 = R_2 \times R_4$	(FX_2)		$R_5 = R_8 \times R_5$	(Z_3^2H)
	$*$			$R_5 = R_4 + R_5$	(Y_3)
Δ_6	$*$				
	$*$				
	$R_5 = R_3 \times R_5$	(Z_3Y_2)			
	$R_5 = R_4 + R_5$	(H)			

TABLE 17. The TPL$^{\mathcal{J}}$ algorithm for curves defined over $\mathbb{F}_{2^m}$ using Jacobian coordinates

TPL$^{\mathcal{J}}$ / $\mathbb{F}_{2^m}$ / Jacobian
Input: $P = (X_1 : Y_1 : Z_1)$
Output: $[3]P = (X_3 : Y_3 : Z_3) = (R_6 : R_4 : R_{11})$
Init: $R_1 = X_1,\ R_2 = Y_1,\ R_3 = Z_1$

Γ_1	$R_4 = R_1^2$	$(A = X_1^2)$	Γ_9	$R_6 = R_1 \times R_2$	(F^2)
	$R_5 = R_1^2$	$(B = X_1^4)$		$*$	
	$R_6 = R_2 \times R_3$	(Y_1Z_1)		$R_6 = a_2 \times R_6$	(a_2F^2)
	$*$			$R_7 = R_4 + R_{11}$	$(G + F)$
Γ_2	$R_7 = R_3^2$	$(C = Z_1^2)$	Γ_{10}	$R_{11} = R_{10}^2$	(E^2)
	$R_8 = R_7^2$	(Z_1^4)		$*$	
	$R_7 = R_1 \times R_7$	(Z_2)		$R_{12} = R_4 \times R_7$	$(G(G + F))$
	$R_9 = R_4 + R_6$	$(A + D)$		$R_6 = R_6 + R_{12}$	
Γ_3	$R_8 = R_8^2$	(Z_1^8)	Γ_{11}	$*$	
	$*$			$*$	
	$R_8 = a_6 \times R_8$	$(a_6Z_1^8)$		$R_{12} = R_{10} \times R_{11}$	(E^3)
	$R_8 = R_5 + R_8$	(X_2)		$R_6 = R_6 + R_{12}$	$(H = X_3)$
Γ_4	$*$		Γ_{12}	$*$	
	$*$			$*$	
	$R_5 = R_5 \times R_7$	(BZ_2)		$R_6 = R_6 \times R_7$	$(H(G + F))$
	$R_9 = R_9 + R_7$	$(A + D + Z_2)$		$*$	
Γ_5	$*$		Γ_{13}	$*$	
	$*$			$*$	
	$R_9 = R_9 \times R_8$			$R_4 = R_4 \times R_8$	(GX_2)
	$R_{10} = R_7 + R_8$	(E)		$*$	
Γ_6	$*$		Γ_{14}	$*$	
	$*$			$*$	
	$R_{11} = R_{10} \times R_7$	$(F = Z_3)$		$R_5 = R_{10} \times R_5$	(EY_2)
	$R_5 = R_5 + R_9$	(Y_2)		$R_4 = R_4 + R_5$	$(GX_2 + EY_2)$
Γ_7	$*$		Γ_{15}	$*$	
	$*$			$*$	
	$R_4 = R_4 \times R_7$	(AZ_2)		$R_4 = R_{11} \times R_4$	
	$*$			$R_4 = R_4 + R_6$	$(I = Y_3)$
Γ_8	$*$				
	$*$				
	$R_4 = R_4 \times R_6$	(AZ_2D)			
	$R_4 = R_4 + R_5$	(G)			

References

[1] J.-P. Allouche and J. Shallit, *Automatic sequences*, Cambridge University Press, 2003. MR1997038 (2004k:11028)

[2] R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of elliptic and hyperelliptic curve cryptography*, CRC Press, 2005. MR2162716 (2007f:14020)

[3] R. Avanzi, V. Dimitrov, C. Doche, and F. Sica, *Extending scalar multiplication using double bases*, Advances in Cryptology, ASIACRYPT'06, Lecture Notes in Computer Science, vol. 4284, Springer, 2006, pp. 130–144.

[4] R. Avanzi and F. Sica, *Scalar multiplication on Koblitz curves using double bases*, Cryptology ePrint Archive, Report 2006/067, 2006, http://eprint.iacr.org/2006/067.

[5] J.-C. Bajard, L. Imbert, and T. Plantard, *Modular number systems: Beyond the Mersenne family*, Proceedings of the 11th International Workshop on Selected Areas in Cryptography, SAC'04, Lecture Notes in Computer Science, vol. 3357, Springer, 2005, pp. 159–169. MR2181315 (2006h:94071)

[6] V. Berthé, *Autour du système de numération d'Ostrowski*, Bulletin of the Belgian Mathematical Society **8** (2001), 209–239. MR1838931 (2002k:68147)

[7] V. Berthé and L. Imbert, *On converting numbers to the double-base number system*, Advanced Signal Processing Algorithms, Architecture and Implementations XIV, Proceedings of SPIE, vol. 5559, SPIE, 2004, pp. 70–78.

[8] I. F. Blake, G. Seroussi, and N. P. Smart, *Advances in elliptic curve cryptography*, London Mathematical Society Lecture Note Series, no. 317, Cambridge University Press, 2005. MR2166105

[9] É. Brier and M. Joye, *Fast point multiplication on elliptic curves through isogenies*, Applied Algebra, Algebraic Algorithms and Error-Correcting Codes, AAECC 2003, Lecture Notes in Computer Science, vol. 2643, Springer, 2003, pp. 43–50. MR2042411 (2005a:14029)

[10] B. Chevalier-Mames, M. Ciet, and M. Joye, *Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity*, IEEE Transactions on Computers **53** (2004), no. 6, 760–768.

[11] J. Chung and A. Hasan, *More generalized Mersenne numbers*, Selected Areas in Cryptography, SAC'03, Lecture Notes in Computer Science, vol. 3006, Springer, 2004, pp. 335–347. MR2094740 (2005f:94089)

[12] M. Ciet, M. Joye, K. Lauter, and P. L. Montgomery, *Trading inversions for multiplications in elliptic curve cryptography*, Designs, Codes and Cryptography **39** (2006), no. 2, 189–206. MR2209936 (2006j:94057)

[13] M. Ciet and F. Sica, *An analysis of double base number systems and a sublinear scalar multiplication algorithm*, Progress of Cryptology, Mycrypt 2005, Lecture Notes in Computer Science, vol. 3715, Springer, 2005, pp. 171–182.

[14] H. Cohen, A. Miyaji, and T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates*, Advances in Cryptology, ASIACRYPT'98, Lecture Notes in Computer Science, vol. 1514, Springer, 1998, pp. 51–65. MR1726152

[15] E. De Win, S. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle, *A fast software implementation for arithmetic operations in* $\mathrm{GF}(2^n)$, Advances in Cryptology, ASIACRYPT'96, Lecture Notes in Computer Science, vol. 1163, Springer, 1996, pp. 65–76. MR1486049

[16] V. Dimitrov, L. Imbert, and P. K. Mishra, *Efficient and secure elliptic curve point multiplication using double-base chains*, Advances in Cryptology, ASIACRYPT'05, Lecture Notes in Computer Science, vol. 3788, Springer, 2005, pp. 59–78. MR2236727

[17] V. S Dimitrov, K. Järvinen, M. J. Jacobson, Jr., W. F. Chan, and Z. Huang, *FPGA implementation of point multiplication on Koblitz curves using Kleinian integers*, Cryptographic Hardware and Embedded Systems, CHES'06, Lecture Notes in Computer Science, vol. 4249, Springer, 2006, pp. 445–459.

[18] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, *An algorithm for modular exponentiation*, Information Processing Letters **66** (1998), no. 3, 155–159. MR1627991 (99d:94023)

[19] C. Doche, T. Icart, and D. R. Kohel, *Efficient scalar multiplication by isogeny decompositions*, Public Key Cryptography, PKC'06, Lecture Notes in Computer Science, vol. 3958, Springer, 2006, pp. 191–206.

[20] C. Doche and L. Imbert, *Extended double-base number system with applications to elliptic curve cryptography*, Progress in Cryptology, INDOCRYPT'06, Lecture Notes in Computer Science, vol. 4329, Springer, 2006, pp. 335–348.

[21] K. Eisenträger, K. Lauter, and P. L. Montgomery, *Fast elliptic curve arithmetic and improved Weil pairing evaluation*, Topics in Cryptology – CT-RSA 2003, Lecture Notes in Computer Science, vol. 2612, Springer, 2003, pp. 343–354. MR2080147

[22] K. Fong, D. Hankerson, J. Lòpez, and A. Menezes, *Field inversion and point halving revisited*, IEEE Transactions on Computers **53** (2004), no. 8, 1047–1059.

[23] D. M. Gordon, *A survey of fast exponentiation methods*, Journal of Algorithms **27** (1998), no. 1, 129–146. MR1613189 (99g:94014)

[24] T. Granlund, *GMP, the GNU multiple precision arithmetic library*, see: http://www.swox.com/gmp/.

[25] J. Guajardo and C. Paar, *Efficient algorithms for elliptic curve cryptosystems*, Advances in Cryptology, CRYPTO'97, Lecture Notes in Computer Science, vol. 1294, Springer, 1997, pp. 342–356. MR1630403 (99b:94033)

[26] D. Hankerson, J. Lòpez Hernandez, and A. Menezes, *Software implementation of elliptic curve cryptography over binary fields*, Cryptographic Hardware and Embedded Systems, CHES'00, Lecture Notes in Computer Science, vol. 1965, Springer, 2000, pp. 1–24.

[27] D. Hankerson, A. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*, Springer, 2004. MR2054891 (2005c:94049)

[28] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara, *Fast implementation of public-key cryptography on a DSP TMS320C6201*, Cryptographic Hardware and Embedded Systems, CHES'99, Lecture Notes in Computer Science, vol. 1717, Springer, 1999, pp. 61 – 72.

[29] T. Izu, B. Möller, and T. Takagi, *Improved elliptic curve multiplication methods resistant against side channel attacks*, Progress in Cryptology, INDOCRYPT'02, Lecture Notes in Computer Science, vol. 2551, Springer, 2002, pp. 269–313.

[30] T. Izu and T. Takagi, *A fast parallel elliptic curve multiplication resistant against side channel attacks*, Public Key Cryptography, PKC'02, Lecture Notes in Computer Science, vol. 2274, Springer, 2002, pp. 280–296.

[31] ———, *Fast elliptic curve multiplications resistant against side channel attacks*, IEICE Transactions Fundamentals **E88-A** (2005), no. 1, 161–171.

[32] M. Joye and C. Tymen, *Protections against differential analysis for elliptic curve cryptography – an algebraic approach*, Cryptographic Hardware and Embedded Systems, CHES'01, Lecture Notes in Computer Science, vol. 2162, Springer, 2001, pp. 377 – 390. MR1946618 (2003k:94031)

[33] N. Koblitz, *Elliptic curve cryptosystems*, Mathematics of Computation **48** (1987), no. 177, 203–209. MR866109 (88b:94017)

[34] P. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, Advances in Cryptology, CRYPTO'99, Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 388–397.

[35] P. C. Kocher, *Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems*, Advances in Cryptology, CRYPTO'96, Lecture Notes in Computer Science, vol. 1109, Springer, 1996, pp. 104–113.

[36] J. Lopez and R. Dahab, *An improvement of the Guajardo-Paar method for multiplication on non-supersingular elliptic curves*, Proceedings of the XVIII International Conference of the Chilean Society of Computer Science, SCCC'98, 1998, pp. 91–95.

[37] K. Mahler, *On a special functional equation*, Journal of the London Mathematical Society **s1-15** (1940), no. 2, 115–123. MR0002921 (2:133e)

[38] V. S. Miller, *Uses of elliptic curves in cryptography*, Advances in Cryptology, CRYPTO'85, Lecture Notes in Computer Science, vol. 218, Springer, 1986, pp. 417–428. MR851432 (88b:68040)

[39] National Institute of Standards and Technology, *FIPS PUB 186-2: Digital signature standard (DSS)*, National Institute of Standards and Technology, January 2000.

[40] W. B. Pennington, *On Mahler's partition problem*, Annals of Mathematics **57** (1953), no. 3, 531–546. MR0053959 (14:846m)

[41] C. M. Skinner, *On the diophantine equation* $ap^x + bq^y = c + dp^z q^w$, Journal of Number Theory **35** (1990), 194–207. MR1057322 (91h:11021)

[42] J. Solinas, *Generalized mersenne numbers*, Research Report CORR-99-39, Center for Applied Cryptographic Research, University of Waterloo, Waterloo, ON, Canada, 1999.
[43] Certicom Research The SECG group, *SEC 2: Recommended elliptic curve domain parameters*, Standard for Efficient Cryptography, September 2000, http://www.secg.org/.
[44] R. Tijdeman, *On the maximal distance between integers composed of small primes*, Compositio Mathematica **28** (1974), 159–162. MR0345917 (49:10646)
[45] H. S. Wilf, *Generatingfunctionology*, 2nd ed., Academic Press Inc., 1994. MR1277813 (95a:05002)

Department of Mathematics, Centre for Information Security and Cryptography, University of Calgary, 2500 University drive N.W., Calgary, AB, T2N 1N4, Canada
E-mail address: dimitrov@vlsi.enel.ucalgary.ca

Department of Mathematics, Centre for Information Security and Cryptography, University of Calgary, 2500 University drive N.W., Calgary, AB, T2N 1N4, Canada
Current address: LIRMM, University Montpellier 2, CNRS, 161 rue Ada, 34392 Montpellier, France
E-mail address: Laurent.Imbert@lirmm.fr

Department of Mathematics, Centre for Information Security and Cryptography, University of Calgary, 2500 University drive N.W., Calgary, AB, T2N 1N4, Canada
E-mail address: pradeep@math.ucalgary.ca

Extended Double-Base Number System with Applications to Elliptic Curve Cryptography

Christophe Doche[1] and Laurent Imbert[2]

[1] Department of Computing
Macquarie University, Australia
doche@ics.mq.edu.au
[2] LIRMM, CNRS, Université Montpellier 2, UMR 5506, France
& ATIPS, CISaC, University of Calgary, Canada
Laurent.Imbert@lirmm.fr

Abstract. We investigate the impact of larger digit sets on the length of Double-Base Number system (DBNS) expansions. We present a new representation system called *extended DBNS* whose expansions can be extremely sparse. When compared with double-base chains, the average length of extended DBNS expansions of integers of size in the range 200–500 bits is approximately reduced by 20% using one precomputed point, 30% using two, and 38% using four. We also discuss a new approach to approximate an integer n by $d2^a3^b$ where d belongs to a given digit set. This method, which requires some precomputations as well, leads to realistic DBNS implementations. Finally, a left-to-right scalar multiplication relying on extended DBNS is given. On an elliptic curve where operations are performed in Jacobian coordinates, improvements of up to 13% overall can be expected with this approach when compared to window NAF methods using the same number of precomputed points. In this context, it is therefore the fastest method known to date to compute a scalar multiplication on a generic elliptic curve.

Keywords: Double-base number system, Elliptic curve cryptography.

1 Introduction

Curve-based cryptography, especially elliptic curve cryptography, has attracted more and more attention since its introduction about twenty years ago [1,2,3], as reflected by the abundant literature on the subject [4,5,6,7]. In curve-based cryptosystems, the core operation that needs to be optimized as much as possible is a scalar multiplication. The standard method, based on ideas well known already more than two thousand years ago, to efficiently compute such a multiplication is the double-and-add method, whose complexity is linear in terms of the size of the input. Several ideas have been introduced to improve this method; see [8] for an overview. In the remainder, we will mainly focus on two approaches:

- Use a representation such that the expansion of the scalar multiple is sparse. For instance, the non-adjacent form (NAF) [9] has a nonzero digit density of

R. Barua and T. Lange (Eds.): INDOCRYPT 2006, LNCS 4329, pp. 335–348, 2006.

1/3 whereas the average density of a binary expansion is 1/2. This improvement is mainly obtained by adding -1 to the set $\{0, 1\}$ of possible coefficients used in binary notation. Another example is the double-base number system (DBNS) [10], in which an integer is represented as a sum of products of powers of 2 and 3. Such expansions can be extremely sparse, *cf.* Section 2.
- Introduce precomputations to enlarge the set of possible coefficients in the expansion and reduce its density. The k-ary and sliding window methods as well as window NAF methods [11,12] fall under this category.

In the present work, we mix these two ideas. Namely, we investigate how precomputations can be used with the DBNS and we evaluate their impact on the overall complexity of a scalar multiplication.

Also, computing a sparse DBNS expansion can be very time-consuming although it is often neglected when compared with other representations. We introduce several improvements that considerably speed up the computation of a DBNS expansion, *cf.* Section 4.

The plan of the paper is as follows. In Section 2, we recall the definition and basic properties of the DBNS. In Section 3, we describe how precomputations can be efficiently used with the DBNS. Section 4 is devoted to implementation aspects and explains how to quickly compute DBNS expansions. In Section 5, we present a series of tests and comparisons with existing methods before concluding in Section 6.

2 Overview of the DBNS

In the *Double-Base Number System*, first considered by Dimitrov *et al.* in a cryptographic context in [13], any positive integer n is represented as

$$n = \sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}, \text{ with } d_i \in \{-1, 1\}. \tag{1}$$

This representation is obviously not unique and is in fact highly redundant. Given an integer n, it is straightforward to find a DBNS expansion using a greedy-type approach. Indeed, starting with $t = n$, the main task at each step is to find the $\{2, 3\}$-integer z that is the closest to t (i.e. the integer z of the form $2^a 3^b$ such that $|t - z|$ is minimal) and then set $t = t - z$. This is repeated until t becomes 0. See Example 2 for an illustration.

Remark 1. *Finding the best $\{2, 3\}$-approximation of an integer t in the most efficient way is an interesting problem on its own. One option is to scan all the points with integer coordinates near the line $y = -x \log_3 2 + \log_3 t$ and keep only the best approximation. A much more sophisticated method involves continued fractions and Ostrowski's number system, cf. [14]. It is to be noted that these methods are quite time-consuming. See Section 4 for a more efficient approach.*

Example 2. *Take the integer* $n = 841232$. *We have the sequence of approximations*

$$\begin{aligned} 841232 &= 2^7 3^8 + 1424, \\ 1424 &= 2^1 3^6 - 34, \\ 34 &= 2^2 3^2 - 2. \end{aligned}$$

As a consequence, $841232 = 2^7 3^8 + 2^1 3^6 - 2^2 3^2 + 2^1$.

It has been shown that every positive integer n can be represented as the sum of at most $O\big(\frac{\log n}{\log \log n}\big)$ signed $\{2,3\}$-integers. For instance, see [13] for a proof. Note that the greedy approach above-mentioned is suitable to find such short expansions.

This initial class of DBNS is therefore very sparse. When one endomorphism is virtually free, like for instance triplings on supersingular curves defined over $\mathbb{F}_3$, the DBNS can be used to efficiently compute $[n]P$ with $\max a_i$ doublings, a very low number of additions, and the necessary number of triplings [15]. Note that this idea has recently been extended to Koblitz curves [16]. Nevertheless, it is not really suitable to compute scalar multiplications in general. For generic curves where both doublings and triplings are expensive, it is essential to minimize the number of applications of these two endomorphisms. Now, one needs at least $\max a_i$ doublings and $\max b_i$ triplings to compute $[n]P$ using (1). However, given the DBNS expansion of n returned by the greedy approach, it seems to be highly non-trivial, if not impossible, to attain these two lower bounds simultaneously.

So, for generic curves the DBNS needs to be adapted to compete with other methods. The concept of *double-base chain*, introduced in [17], is a special type of DBNS. The idea is still to represent n as in (1) but with the extra requirements $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$ and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$. These properties allow to compute $[n]P$ from right-to-left very easily. It is also possible to use a Horner-like scheme that operates from left-to-right. These two methods are illustrated after Example 3.

Note that, it is easy to accommodate these requirements by restraining the search of the best exponents (a_{j+1}, b_{j+1}) to the interval $[0, a_j] \times [0, b_j]$.

Example 3. *A double-base chain of* n *can be derived from the following sequence of equalities:*

$$\begin{aligned} 841232 &= 2^7 3^8 + 1424, \\ 1424 &= 2^1 3^6 - 34, \\ 34 &= 3^3 + 7, \\ 7 &= 3^2 - 2, \\ 2 &= 3^1 - 1. \end{aligned}$$

As a consequence, $841232 = 2^7 3^8 + 2^1 3^6 - 3^3 - 3^2 + 3^1 - 1$.

In that particular case, the length of this double-base chain is strictly bigger than the one of the DBNS expansion in Example 2. This is true in general as

well and the difference can be quite large. It is not known whether the bound $O\big(\frac{\log n}{\log\log n}\big)$ on the number of terms is still valid for double-base chains.

However, computing $[841232]P$ is now a trivial task. From right-to-left, we need two variables. The first one, T being initialized with P and the other one, S set to point at infinity. The successive values of T are then P, $[3]P$, $[3^2]P$, $[3^3]P$, $[2^1 3^6]P$, and $[2^7 3^8]P$, and at each step T is added to S. Doing that, we obtain $[n]P$ with 7 doublings, 8 triplings, and 5 additions. To proceed from left-to-right, we notice that the expansion that we found can be rewritten as

$$841232 = 3\big(3\big(3\big(2^1 3^3(2^6 3^2+1)-1\big)-1\big)+1\big)-1,$$

which implies that

$$[841232]P = [3]\big([3]\big([3]\big([2^1 3^3]([2^6 3^2]P+P)-P\big)-P\big)+P\big)-P.$$

Again, 7 doublings, 8 triplings, and 5 additions are necessary to obtain $[n]P$.

More generally, one needs exactly a_1 doublings and b_1 triplings to compute $[n]P$ using double-base chains. The value of these two parameters can be optimized depending on the size of n and the respective complexities of a doubling and a tripling (see Figure 2).

To further reduce the complexity of a scalar multiplication, one option is to reduce the number of additions, that is to minimize the density of DBNS expansions. A standard approach to achieve this goal is to enlarge the set of possible coefficients, which ultimately means using precomputations.

3 Precomputations for DBNS Scalar Multiplication

We suggest to use precomputations in two ways. The first idea, which applies only to double-base chains, can be viewed as a two-dimensional window method.

3.1 Window Method

Given integers w_1 and w_2, we represent n as in (1) but with coefficients d_i in the set $\{\pm 1, \pm 2^1, \pm 2^2, \ldots, \pm 2^{w_1}, \pm 3^1, \pm 3^2, \ldots, \pm 3^{w_2}\}$. This is an indirect way to relax the conditions $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$ and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$ in order to find better approximations and hopefully sparser expansions. This method, called (w_1, w_2)-*double-base chain*, lies somewhere between normal DBNS and double-base chain methods.

Example 4. *The DBNS expansion of* $841232 = 2^7 3^8 + 2^1 3^6 - 2^2 3^2 + 2^1$, *can be rewritten as* $841232 = 2^7 3^8 + 2^1 3^6 - \mathbf{2} \times 2^1 3^2 + 2^1$, *which is a* $(1,0)$*-window-base chain. The exponent* a_3 *that was bigger than* a_2 *in Example 2 has been replaced by* a_2 *and the coefficient* d_3 *has been multiplied by* 2 *accordingly. As a result, we now have two decreasing sequences of exponents and the expansion is only four terms long.*

It remains to see how to compute $[841232]P$ from this expansion. The right-to-left scalar multiplication does not provide any improvement, but this is not the

case for the left-to-right approach. Writing $841232 = 2\big(3^2\big(3^4(2^6 3^2+1)-2\big)+1\big)$, we see that

$$[841232]P = [2]\big([3^2]\big([3^4]([2^6 3^2]P + P) - [2]P\big) + P\big).$$

If $[2]P$ is stored along the computation of $[2^6 3^2]P$ then 7 doublings, 8 triplings and only 3 additions are necessary to obtain $[841232]P$.

It is straightforward to design an algorithm to produce (w_1, w_2)-double-base chains. We present a more general version in the following, *cf.* Algorithm 1. See Remark 6 (v) for specific improvements to (w_1, w_2)-double-base chains.

Also a left-to-right scalar multiplication algorithm can easily be derived from this method, *cf.* Algorithm 2.

The second idea to obtain sparser DBNS expansions is to generalize the window method such that any set of coefficients is allowed.

3.2 Extended DBNS

In a (w_1, w_2)-double-base chain expansion, the coefficients are signed powers of 2 or 3. Considering other sets $\mathcal{S}$ of coefficients, for instance odd integers coprime with 3, should further reduce the average length of DBNS expansions. We call this approach *extended DBNS* and denote it by $\mathcal{S}$-DBNS.

Example 5. *We have* $841232 = 2^7 3^8 + \mathbf{5} \times 2^5 3^2 - 2^4$. *The exponents form two decreasing sequences, but the expansion has only three terms. Assuming that* $[5]P$ *is precomputed, it is possible to obtain* $[841232]P$ *as*

$$[2^4]\big([2^1 3^2]([2^2 3^6]P + [5]P) - P\big)$$

with 7 *doublings,* 8 *triplings, and only* 2 *additions*

This strategy applies to any kind of DBNS expansion. In the following, we present a greedy-type algorithm to compute extended double-base chains.

Algorithm 1. Extended double-base chain greedy algorithm

INPUT: A positive integer n, a parameter a_0 such that $a_0 \leqslant \lceil \log_2 n \rceil$, and a set $\mathcal{S}$ containing 1.

OUTPUT: Three sequences $(d_i, a_i, b_i)_{1 \leqslant i \leqslant \ell}$ such that $n = \sum_{i=1}^{\ell} d_i 2^{a_i} 3^{b_i}$ with $|d_i| \in \mathcal{S}$, $a_1 \geqslant a_2 \geqslant \cdots \geqslant a_\ell$, and $b_1 \geqslant b_2 \geqslant \cdots \geqslant b_\ell$.

1. $b_0 \leftarrow \lceil (\log_2 n - a_0) \log_2 3 \rceil$ [See Remark 6 (ii)]
2. $i \leftarrow 1$ and $t \leftarrow n$
3. $s \leftarrow 1$ [to keep track of the sign]
4. **while** $t > 0$ **do**
5. find the best approximation $z = d_i 2^{a_i} 3^{b_i}$ of t with $d_i \in \mathcal{S}$, $0 \leqslant a_i \leqslant a_{i-1}$, and $0 \leqslant b_i \leqslant b_{i-1}$
6. $d_i \leftarrow s \times d_i$
7. **if** $t < z$ **then** $s \leftarrow -s$

8. $t \leftarrow |t - z|$
9. $i \leftarrow i + 1$
10. **return** (d_i, a_i, b_i)

Remarks 6

(i) Algorithm 1 processes the bits of n from left-to-right. It terminates since the successive values of t form a strictly decreasing sequence.
(ii) The parameters a_0 and b_0 are respectively the biggest powers of 2 and 3 allowed in the expansion. Their values have a great influence on the density of the expansion, *cf.* Section 5 for details.
(iii) To compute normal DBNS sequences instead of double-base chains, replace the two conditions $0 \leqslant a_i \leqslant a_{i-1}$, $0 \leqslant b_i \leqslant b_{i-1}$ in Step 5 by $0 \leqslant a_i \leqslant a_0$ and $0 \leqslant b_i \leqslant b_0$.
(iv) In the following, we explain how to find the best approximation $d_i 2^{a_i} 3^{b_i}$ of t in a very efficient way. In addition, the proposed method has a time-complexity that is mainly independent of the size of $\mathcal{S}$ and not directly proportional to it as with a naïve search. See Section 4 for details.
(v) To obtain (w_1, w_2)-double-base chains, simply ensure that $\mathcal{S}$ contains only powers 2 and 3. However, there is a more efficient way. First, introduce two extra variables $a_{\max}$ and $b_{\max}$, initially set to a_0 and b_0 respectively. Then in Step 5, search for the best approximation z of t of the form $2^{a_i} 3^{b_i}$ with $(a_i, b_i) \in [0, a_{\max} + w_1] \times [0, b_{\max} + w_2] \setminus [a_{\max} + 1, a_{\max} + w_1] \times [b_{\max} + 1, b_{\max} + w_2]$. In other words, we allow one exponent to be slightly bigger than its current maximal bound, but the (exceptional) situation where $a_i > a_{\max}$ and $b_i > b_{\max}$ simultaneously is forbidden. Otherwise, we should be obliged to include in $\mathcal{S}$ products of powers of 2 and 3 and increase dramatically the number of precomputations. Once the best approximation has been found, if a_i is bigger than $a_{\max}$, then a_i is changed to $a_{\max}$ while d_i is set to $2^{a_i - a_{\max}}$. If b_i is bigger than $b_{\max}$, then b_i is changed to $b_{\max}$ while d_i is set to $3^{b_i - b_{\max}}$. Finally, do $a_{\max} \leftarrow \min(a_i, a_{\max})$ and $b_{\max} \leftarrow \min(b_i, b_{\max})$ and the rest of the Algorithm remains unchanged.
(vi) In the remainder, we discuss some results obtained with Algorithm 1 using different sets of coefficients. More precisely, each set $\mathcal{S}_m$ that we consider contains the first $m + 1$ elements of $\{1, 5, 7, 11, 13, 17, 19, 23, 25\}$.

We now give an algorithm to compute a scalar multiplication from the expansion returned by Algorithm 1.

Algorithm 2. Extended double-base chain scalar multiplication

INPUT: A point P on an elliptic curve E, a positive integer n represented by $(d_i, a_i, b_i)_{1 \leqslant i \leqslant \ell}$ as returned by Algorithm 1, and the points $[k]P$ for each $k \in \mathcal{S}$.
OUTPUT: The point $[n]P$ on E.

1. $T \leftarrow O_E$ [O_E is the point at infinity on E]
2. set $a_{\ell+1} \leftarrow 0$ and $b_{\ell+1} \leftarrow 0$
3. **for** $i = 1$ **to** ℓ **do**
4. $\quad T \leftarrow T \oplus [d_i]P$
5. $\quad T \leftarrow [2^{a_i - a_{i+1}} 3^{b_i - b_{i+1}}]T$
6. **return** T

Example 7. *For $n = 841232$, the sequence returned by Algorithm 2 with $a_0 = 8$, $b_0 = 8$, and $\mathcal{S} = \{1, 5\}$ is $(1, 7, 8), (5, 5, 2), (-1, 4, 0)$. In the next Table, we show the intermediate values taken by T in Algorithm 2 when applied to the above-mentioned sequence. The computation is the same as in Example 5.*

i	d_i	$a_i - a_{i+1}$	$b_i - b_{i+1}$	T
1	1	2	6	$[2^2 3^6]P$
2	5	1	2	$[2^1 3^2]([2^2 3^6]P + [5]P)$
3	-1	4	0	$[2^4]\big([2^1 3^2]([2^2 3^6]P + [5]P) - P\big)$

Remark 8. *The length of the chain returned by Algorithm 1 greatly determines the performance of Algorithm 2. However, no precise bound is known so far, even in the case of simple double-base chains. So, at this stage our knowledge is only empirical, cf. Figure 2. More work is therefore necessary to establish the complexity of Algorithm 2.*

4 Implementation Aspects

This part describes how to efficiently compute the best approximation of any integer n in terms of $d_1 2^{a_1} 3^{b_1}$ for some $d_1 \in \mathcal{S}$, $a_1 \leqslant a_0$, and $b_1 \leqslant b_0$. The method works on the binary representation of n denoted by $(n)_2$. It operates on the most significant bits of n and uses the fact that a multiplication by 2 is a simple shift.

To make things clear, let us explain the algorithm when $\mathcal{S} = \{1\}$. First, take a suitable bound B and form a two-dimensional array of size $(B + 1) \times 2$. For each $b \in [0, B]$, the corresponding row vector contains $[(3^b)_2, b]$. Then sort this array with respect to the first component using lexicographic order denoted by $\preccurlyeq$ and store the result.

To compute an approximation of n in terms of $2^{a_1} 3^{b_1}$ with $a_1 \leqslant a_0$ and $b_1 \leqslant b_0$, find the two vectors v_1 and v_2 such that $v_1[1] \preccurlyeq (n)_2 \preccurlyeq v_2[1]$. This can be done with a binary search in $O(\log B)$ operations.

The next step is to find the first vector v_1' that smaller than v_1 in the sorted array and that is suitable for the approximation. More precisely, we require that:

- the difference δ_1 between the binary length of n and the length of $v_1'[1]$ satisfies $0 \leqslant \delta_1 \leqslant a_0$,
- the corresponding power of 3, *i.e.* $v_1'[2]$, is less than b_0.

This operation is repeated to find the first vector v_2' that is greater than v_2 and fulfills the same conditions as above. The last step is to decide which approximation, $2^{\delta_1}3^{v_1'[2]}$ or $2^{\delta_2}3^{v_2'[2]}$, is closer to n.

In case $|\mathcal{S}| > 1$, the only difference is that the array is of size $\big(|\mathcal{S}|(B+1)\big) \times 3$. Each row vector is of the form $[(d3^b)_2, b, d]$ for $d \in \mathcal{S}$ and $b \in [0, B]$. Again the array is sorted with respect to the first component using lexicographic order. Note that multiplying the size of the table by $|\mathcal{S}|$ has only a negligible impact on the time complexity of the binary search. See [18, Appendix A] for a concrete example and some improvements to this approach.

This approximation method ultimately relies on the facts that lexicographic and natural orders are the same for binary sequences of the same length and also that it is easy to adjust the length of a sequence by multiplying it by some power of 2. The efficiency comes from the sorting operation (done once at the beginning) that allows to retrieve which precomputed binary expansions are close to n, by looking only at the most significant bits.

For environments with constrained memory, it may be difficult or even impossible to store the full table. In this case, we suggest to precompute only the first byte or the first two bytes of the binary expansions of $d3^b$ together with their binary length. This information is sufficient to find two approximations A_1, A_2 in the table such that $A_1 \leqslant n \leqslant A_2$, since the algorithm operates only on the most significant bits. However, this technique is more time-consuming since it is necessary to actually *compute* at least one approximation and sometimes more, if the first bits are not enough to decide which approximation is the closest to n.

In Table 1, we give the precise amount of memory (in bytes) that is required to store the vectors used for the approximation for different values of B. Three situations are investigated, *i.e.* when the first byte, the first two bytes, and the full binary expansions $d3^b$, for $d \in \mathcal{S}$ and $b \leqslant B$ are precomputed and stored. See [19] for a PARI/GP implementation of Algorithm 1 using the techniques described in this section.

5 Tests and Results

In this section, we present some tests to help evaluating the relevance of extended double-base chains for scalar multiplications on generic elliptic curves defined over $\mathbb{F}_p$, for p of size between 200 and 500 bits. Comparisons with the best systems known so far, including ℓ-NAF$_w$ and normal double-base chains are given.

In the following, we assume that we have three basic operations on a curve E to perform scalar multiplications, namely addition/subtraction, doubling, and tripling. In turn, each one of these elliptic curve operations can be seen as a sequence of inversions I, multiplications M, and squarings S in the underlying field $\mathbb{F}_p$.

There exist different systems of coordinates with different complexities. For many platforms, projective-like coordinates are quite efficient since they do not

Table 1. Precomputations size (in bytes) for various bounds B and sets $\mathcal{S}$

Bound B	25	50	75	100	125	150	175	200
$\mathcal{S} = \{1\}$								
First byte	33	65	96	127	158	190	221	251
First two bytes	54	111	167	223	279	336	392	446
Full expansion	85	293	626	1,084	1,663	2,367	3,195	4,108
$\mathcal{S} = \{1, 5, 7\}$								
First byte	111	214	317	420	523	627	730	829
First two bytes	178	356	534	712	890	1,069	1,247	1,418
Full expansion	286	939	1,962	3,357	5,122	7,261	9,769	12,527
$\mathcal{S} = \{1, 5, 7, 11, 13\}$								
First byte	185	357	529	701	873	1,045	1,216	1,381
First two bytes	300	597	894	1,191	1,488	1,785	2,081	2,366
Full expansion	491	1,589	3,305	5,642	8,598	12,173	16,364	20,972
$\mathcal{S} = \{1, 5, 7, 11, 13, 17, 19, 23, 25\}$								
First byte	334	643	952	1,262	1,571	1,881	2,190	2,487
First two bytes	545	1,079	1,613	2,148	2,682	3,217	3,751	4,264
Full expansion	906	2,909	6,026	10,255	15,596	22,056	29,630	37,947

require any field inversion for addition and doubling, *cf.* [20] for a comparison. Thus, our tests will not involve any inversion. Also, to ease comparisons between different scalar multiplication methods, we will make the standard assumption that S is equivalent to 0.8M. Thus, the complexity of a scalar multiplication will be expressed in terms of a number of field multiplications only and will be denoted by N_{M}.

Given any curve $E/\mathbb{F}_p$ in Weierstraß form, it is possible to directly obtain $[3]P$ more efficiently than computing a doubling followed by an addition. Until now, all these direct formulas involved at least one inversion, *cf.* [21], but recently, an inversion-free tripling formula has been devised for Jacobian projective coordinates [17]. Our comparisons will be made using this system. In Jacobian coordinates, a point represented by $(X_1 : Y_1 : Z_1)$ corresponds to the affine point $(X_1/Z_1^2, Y_1/Z_1^3)$, if $Z_1 \neq 0$, and to the point at infinity O_E otherwise. A doubling can be done with 4M + 6S, a tripling with 10M + 6S and a mixed addition, *i.e.* an addition between a point in Jacobian coordinates and an affine point, using 8M + 3S.

With these settings, we display in Figure 1, the number of multiplications N_{M} required to compute a scalar multiplication on a 200-bit curve with Algorithm 2, for different choices of a_0 and various DBNS methods. Namely, we investigate double-base chains as in [17], window double-base chains with 2 and 8 precomputations, and extended double-base chains with $\mathcal{S}_2 = \{1, 5, 7\}$ and $\mathcal{S}_8 = \{1, 5, 7, 11, 13, 17, 19, 23, 25\}$, as explained in Section 3.2. Comparisons are done on 1,000 random 200-bit scalar multiples. Note that the costs of the precomputations are not included in the results.

Figure 1 indicates that $a_0 = 120$ is close to the optimal choice for every method. This implies that the value of b_0 should be set to 51. Similar computations have been done for sizes between 250 and 500. It appears that a simple and good heuristic to minimize N_{M} is to set $a_0 = \lceil 120 \times \mathrm{size}/200 \rceil$ and b_0 accordingly. These values of a_0 and b_0 are used in the remainder for sizes in $[200, 500]$.

In Figure 2, we display the average length of different extended DBNS expansions in function of the size of the scalar multiple n. Results show that the length of a classic double-base chain is reduced by more than 25% with only 2 precomputations and by 43% with 8 precomputations.

In Table 2, we give the average expansion length ℓ, as well as the maximal power a_1 (resp. b_1) of 2 (resp. 3) in the expansion for different methods and different sizes. The symbol $\#\mathcal{P}$ is equal to the number of precomputed points for a given method and the set $\mathcal{S}_m$ contains the first $m+1$ elements of $\{1, 5, 7, 11, 13, 17, 19, 23, 25\}$. Again, $1,000$ random integers have been considered in each case.

In Table 3, we give the corresponding complexities in terms of the number of multiplications and the gain that we can expect with respect to a window NAF method involving the same number of precomputations.

See [18] for a full version including a similar study for some special curves.

6 Conclusion

In this work, we have introduced a new family of DBNS, called extended DBNS, where the coefficients in the expansion belong to a given digit set $\mathcal{S}$. A scalar multiplication algorithm relying on this representation and involving precomputations was presented. Also, we have described a new method to quickly find the best approximation of an integer by a number of the form $d2^a3^b$ with $d \in \mathcal{S}$. This approach greatly improves the practicality of the DBNS. Extended DBNS sequences give rise to the fastest scalar multiplications known to date for generic elliptic curves. In particular, given a fixed number of precomputations, the extended DBNS is more efficient than any corresponding window NAF method. Gains are especially important for a small number of precomputations, typically up to three points. Improvements larger than 10% over already extremely optimized methods can be expected. Also, this system is more flexible, since it can be used with any given set of coefficients, unlike window NAF methods.

Further research will include an extension of these ideas to Koblitz curves, for which DBNS-based scalar multiplication techniques without precomputations exist already, see [16,22,23]. This will most likely lead to appreciable performance improvements.

Acknowledgements

This work was partly supported by the Canadian NSERC strategic grant #73-2048, *Novel Implementation of Cryptographic Algorithms on Custom Hardware*

Platforms and by a French-Australian collaborative research program from the *Direction des Relations Européennes et Internationales du CNRS*, France.

References

1. Miller, V.S.: Use of elliptic curves in cryptography. In: Advances in Cryptology – Crypto 1985. Volume 218 of Lecture Notes in Comput. Sci. Springer-Verlag, Berlin (1986) 417–426
2. Koblitz, N.: Elliptic curve cryptosystems. Math. Comp. **48** (1987) 203–209
3. Koblitz, N.: Hyperelliptic cryptosystems. J. Cryptology **1** (1989) 139–150
4. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic curves in cryptography. Volume 265 of London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge (1999)
5. Hankerson, D., Menezes, A.J., Vanstone, S.A.: Guide to elliptic curve cryptography. Springer-Verlag, Berlin (2003)
6. Avanzi, R.M., Cohen, H., Doche, C., Frey, G., Nguyen, K., Lange, T., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. Discrete Mathematics and its Applications (Boca Raton). CRC Press, Inc. (2005)
7. Blake, I.F., Seroussi, G., Smart, N.P.: Advances in Elliptic Curve Cryptography. Volume 317 of London Mathematical Society Lecture Note Series. Cambridge University Press, Cambridge (2005)
8. Doche, C.: Exponentiation. In: [6] 145–168
9. Morain, F., Olivos, J.: Speeding up the Computations on an Elliptic Curve using Addition-Subtraction Chains. Inform. Theor. Appl. **24** (1990) 531–543
10. Dimitrov, V.S., Jullien, G.A., Miller, W.C.: Theory and applications of the double-base number system. IEEE Trans. on Computers **48** (1999) 1098–1106
11. Miyaji, A., Ono, T., Cohen, H.: Efficient Elliptic Curve Exponentiation. In: Information and Communication – ICICS'97. Volume 1334 of Lecture Notes in Comput. Sci., Springer (1997) 282–291
12. Takagi, T., Yen, S.M., Wu, B.C.: Radix-r non-adjacent form. In: Information Security Conference – ISC 2004. Volume 3225 of Lecture Notes in Comput. Sci. Springer-Verlag, Berlin (2004) 99–110
13. Dimitrov, V.S., Jullien, G.A., Miller, W.C.: An algorithm for modular exponentiation. Information Processing Letters **66** (1998) 155–159
14. Berthé, V., Imbert, L.: On converting numbers to the double-base number system. In: In F. T. Luk, editor, Advanced Signal Processing Algorithms, Architecture and Implementations XIV, volume 5559 of Proceedings of SPIE. (2004) 70–78
15. Ciet, M., Sica, F.: An Analysis of Double Base Number Systems and a Sublinear Scalar Multiplication Algorithm. In: Progress in Cryptology – Mycrypt 2005. Volume 3715 of Lecture Notes in Comput. Sci., Springer (2005) 171–182
16. Avanzi, R.M., Sica, F.: Scalar Multiplication on Koblitz Curves using Double Bases. In: proceedings of Vietcrypt 2006. Lecture Notes in Comput. Sci. (2006) See also Cryptology ePrint Archive, Report 2006/067, `http://eprint.iacr.org/`
17. Dimitrov, V.S., Imbert, L., Mishra, P.K.: Efficient and secure elliptic curve point multiplication using double-base chains. In: Advances in Cryptology – Asiacrypt 2005. Volume 3788 of Lecture Notes in Comput. Sci., Springer (2005) 59–78
18. Doche, C., Imbert, L.: Extended Double-Base Number System with Applications to Elliptic Curve Cryptography (2006) full version of the present paper, see Cryptology ePrint Archive, `http://eprint.iacr.org/`

19. Doche, C.: A set of PARI/GP functions to compute DBNS expansions `http://www.ics.mq.edu.au/~doche/dbns_basis.gp`
20. Doche, C., Lange, T.: Arithmetic of Elliptic Curves. In: [6] 267–302
21. Ciet, M., Joye, M., Lauter, K., Montgomery, P.L.: Trading inversions for multiplications in elliptic curve cryptography. Des. Codes Cryptogr. **39** (2006) 189–206
22. Dimitrov, V.S., Jarvine, K., Jr, M.J.J., Chan, W.F., Huang, Z.: FPGA Implementation of Point Multiplication on Koblitz Curves Using Kleinian Integers. In: proceedings of CHES 2006. Lecture Notes in Comput. Sci. (2006)
23. Avanzi, R.M., Dimitrov, V.S., Doche, C., Sica, F.: Extending Scalar Multiplication using Double Bases. In: proceedings of Asiacrypt 2006. Lecture Notes in Comput. Sci., Springer (2006)

Appendix: Graphs and Tables

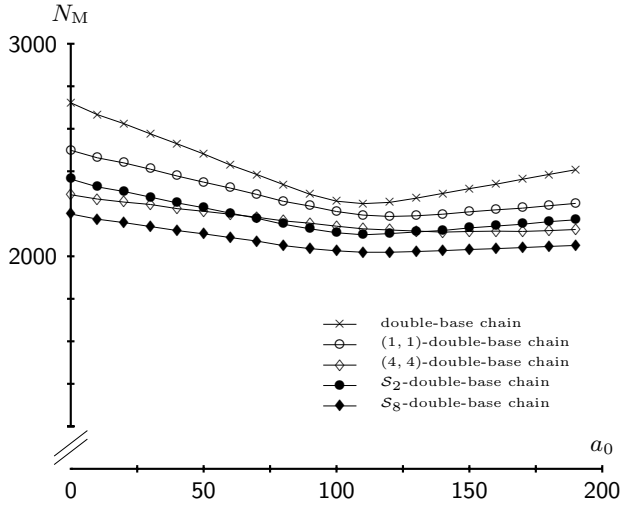

Fig. 1. Average number of multiplications to perform a random scalar multiplication on a generic 200-bit curve with various DBNS methods parameterized by a_0

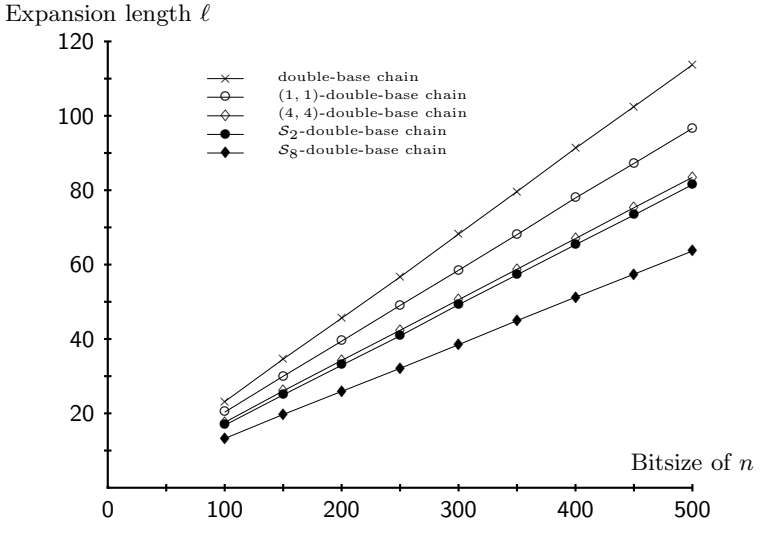

Fig. 2. Average expansion length ℓ of random scalar integers n with various DBNS

Table 2. Parameters for various scalar multiplication methods on generic curves

Size		200 bits			300 bits			400 bits			500 bits		
	$\#\mathcal{P}$	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1	ℓ	a_1	b_1
$2NAF_2$	0	66.7	200	0	100	300	0	133.3	400	0	166.7	500	0
Binary/ternary	0	46.1	90.7	68.1	69.2	136.4	102.2	91.9	182.6	136.3	114.4	228.0	170.7
DB-chain	0	45.6	118.7	50.4	68.2	178.7	75.5	91.3	239.0	100.6	113.7	298.6	126.2
$3NAF_2$	1	50	200	0	75	300	0	100	400	0	125	500	0
$(1,0)$-DB-chain	1	46.8	118.9	50.2	70.5	179.1	75.1	94.5	239.3	100.3	117.7	298.8	125.9
$(0,1)$-DB-chain	1	42.9	118.7	50.4	63.8	178.7	75.5	85.4	239.0	100.6	106.4	298.6	126.2
$\mathcal{S}_1$-DB chain	1	36.8	118.1	49.9	55.0	178.0	75.0	72.9	238.2	100.1	91.0	297.8	125.7
$2NAF_3$	2	50.4	0	126	75.6	0	189	100.8	0	252	126	0	315
$(1,1)$-DB-chain	2	39.4	118.9	50.2	58.5	179.1	75.1	77.9	239.3	100.3	96.6	298.8	125.9
$\mathcal{S}_2$-DB chain	2	32.9	117.8	49.8	49.2	177.8	74.9	65.3	238	100.0	81.5	297.7	125.6
$4NAF_2$	3	40	200	0	60	300	0	80	400	0	100	500	0
$\mathcal{S}_3$-DB chain	3	30.7	117.5	49.7	45.7	177.5	74.8	60.6	237.8	99.8	75.6	297.3	125.4
$(2,2)$-DB-chain	4	36.8	119.2	49.8	54.7	179.3	74.8	72.6	239.4	100.1	90.5	299.0	125.7
$\mathcal{S}_4$-DB chain	4	28.9	117.3	49.6	43.2	177.3	74.7	57.6	237.6	99.8	71.5	297.1	125.4
$(3,3)$-DB-chain	6	35.3	119.3	49.5	52.2	179.4	74.6	69.2	239.5	99.6	86.1	299.2	125.2
$\mathcal{S}_6$-DB chain	6	27.3	117.4	49.4	40.6	177.3	74.5	54.0	237.6	99.6	67.1	297	125.3
$3NAF_3$	8	36	0	126	54	0	189	72	0	252	90	0	315
$(4,4)$-DB-chain	8	34.2	119.3	49.3	50.5	179.5	74.2	67.0	239.6	99.3	83.5	299.3	125
$\mathcal{S}_8$-DB chain	8	25.9	117.2	49.3	38.5	177.1	74.4	51.2	237.4	99.5	63.6	296.9	125.2

Table 3. Complexity of various extended DBNS methods for generic curves and gain with respect to window NAF methods having the same number of precomputations

Size		200 bits		300 bits		400 bits		500 bits	
	$\#\mathcal{P}$	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain	N_{M}	Gain
$2NAF_2$	0	2442.9	—	3669.6	—	4896.3	—	6122.9	—
Binary/ternary	0	2275.0	6.87%	3422.4	6.74%	4569.0	6.68%	5712.5	6.70%
DB-chain	0	2253.8	7.74%	3388.5	7.66%	4531.8	7.44%	5666.5	7.45%
$3NAF_2$	1	2269.6	—	3409.6	—	4549.6	—	5689.6	—
$(1,0)$-DB-chain	1	2265.8	0.17%	3410.3	−1.98%	4562.3	−1.72%	5707.4	−1.69%
$(0,1)$-D B-chain	1	2226.5	1.90%	3343.2	1.95%	4471.0	1.73%	5590.4	1.74%
$\mathcal{S}_1$-DB chain	1	2150.4	5.25%	3238.1	5.03%	4326.3	4.91%	5418.1	4.77%
$2NAF_3$	2	2384.8	—	3579.3	—	4773.8	—	5968.2	—
$(1,1)$-DB-chain	2	2188.6	8.23%	3285.5	8.21%	4390.0	8.04%	5487.7	8.05%
$\mathcal{S}_2$-DB chain	2	2106.5	11.67%	3174.1	11.32%	4243.6	11.11%	5314.8	10.95%
$4NAF_2$	3	2165.6	—	3253.6	—	4341.6	—	5429.6	—
$\mathcal{S}_3$- DB chain	3	2078.1	4.04%	3132.8	3.71%	4189.8	3.50%	5248.5	3.34%
$(2,2)$-DB-chain	4	2158.2	—	3242.6	—	4333.1	—	5421.6	—
$\mathcal{S}_4$-DB chain	4	2056.7	—	3105.0	—	4156.1	—	5204.0	—
$(3,3)$-DB-chain	6	2139.4	—	3215.0	—	4291.7	—	5371.9	—
$\mathcal{S}_6$-DB chain	6	2036.3	—	3074.3	—	4115.4	—	5155.1	—
$3NAF_3$	8	2236.2	—	3355.8	—	4475.4	—	5595.0	—
$(4,4)$-DB-chain	8	2125.4	4.95%	3192.2	4.88%	4264.1	4.72%	5340.5	4.55%
$\mathcal{S}_8$-DB chain	8	2019.3	9.70%	3049.8	9.12%	4084.3	8.74%	5116.8	8.55%

Multiplication by a Constant is Sublinear

Vassil Dimitrov
ATIPS Labs, CISaC
University of Calgary
2500 University drive NW
T2N 1N4, Calgary, AB
Canada

Laurent Imbert
LIRMM, Univ. Montpellier 2, CNRS
Montpellier, France
& ATIPS Labs, CISaC
University of Calgary
Canada

Andrew Zakaluzny
ATIPS Labs
University of Calgary
2500 University drive NW
T2N 1N4, Calgary, AB
Canada

***Abstract*— This paper explores the use of the double-base number system (DBNS) for constant integer multiplication. The DBNS recoding scheme represents integers – in this case constants – in a multiple-radix way in the hope of minimizing the number of additions to be performed during constant multiplication. On the theoretical side, we propose a formal proof which shows that our recoding technique diminishes the number of additions in a sublinear way. Therefore, we prove Lefèvre's conjecture that the multiplication by an integer constant is achievable in sublinear time. In a second part, we investigate various strategies and we provide numerical data showcasing the potential interest of our approach.**

I. Introduction

Multiplication by an integer constant has many applications; for example in digital signal processing, image processing, multiple precision arithmetic, cryptography and in the design of compilers. In certain applications, like the discrete cosine transform (DCT), the implementation of integer constant multiplications is the bottleneck as it largely determines the speed of the entire process. Therefore, it is imperative that multiplications by integer constants in these high throughput applications are optimized as much as possible.

The problem of optimizing multiplication by constants is that not all constants behave the same. In other words, a technique for optimizing the multiplication by the specific constant c may not optimize the multiplication by another constant c'. Therefore, finding solutions that optimize multiplication for most constants over a specified range is an important problem which has been sought after by many authors.

Given an integer constant c, the goal is to find a program which computes $c \times x$ with as few operations as possible. A basic complexity model is to count the number of additions only, assuming that multiplications by powers of 2 are free (left shifts). The number of additions is highly dependent upon the number of non-zero digits in the representation of the constant c. For example, if c is a n-bit constant, then one needs on average $n/2$ additions with the double-and-add method, sometimes referred to as the binary method; and $n/3$ additions if c is expressed in the Canonic Signed Digit (CSD) representation: a variant of Booth's recoding technique [1]. Other classes of algorithms based on cost functions or the search for patterns in the binary expansion of c are described in the papers from Bernstein [2], Lefèvre [3], and Boullis and Tisserand [4]. These algorithms give very good results in practice at the cost of quite expensive computations. Moreover, the asymptotic complexities of the generated programs are difficult to analyze.

In this paper, we propose several variants of an algorithm based on integer recoding, where the constant c is represented as a sum of mixed powers of two coprime bases; e.g. 2 and 3 or 2 and 5. This very sparse representation scheme, called Double-Base Number System (DBNS), has been used in digital signal processing [5] and cryptographic [6] applications with great success. By restricting the largest exponent of the second base to some well chosen bound, we obtain a sublinear constant multiplication algorithm; the resulting program requires $O(\log c/ \log \log c)$ additions and/or subtractions. Our numerical experiments confirm the asymptotic sublinear behavior, even for relatively small numbers (32 to 64 bits).

This paper is organized as follows: We define the problem and present some previous works in Section II and III. In Section IV, we introduces the Double-Base Number System. We present our new algorithm and the proof of sublinearity in Section V. In Section VI, we propose different heuristics and several numerical experiments.

II. Problem Definition

In many applications, multiplication of integers is simply done with an all purpose multiplier. (On recent general purpose processors, it is sometimes even faster to use the floating-point multiplier to perform an integer multiplication.) As mentioned before, many applications require a high throughput of constant integer multiplications, and would benefit from a customized integer multiplier suited to the specific constants used. In essence, multiplication is a series of shifts and additions, but in some cases, it might be a good idea to allow subtractions. The central point of the constant multiplication problem is to minimize the total number of additions/subtractions required for each integer multiplication.

In the following, we shall use $x \ll k$ to denote the value obtained when the variable x is shifted to the left by k places (bits); i.e., the value $x \times 2^k$. For simplicity, we consider a complexity model where the cost of shifts is ignored. Note that this widely assumed assumption might not correspond to practical reality, especially in the context of multiple precision arithmetic. For hardware implementations, however, this assumption seems reasonable. Therefore, in order to simplify

18th IEEE Symposium on Computer Arithmetic(ARITH'07)
0-7695-2854-6/07 $20.00 © 2007 IEEE

the presentation, we only take into account the number of additions and/or subtractions. We also assume that addition and subtraction have the same speed and cost. Hence, we will sometimes refer to the number of additions, or even to the number of operations, by which terminology we include subtractions.

Let us start with a very small example. We want compute the product of the unknown integer x by the integer constant $c = 151 = 10010111_2$. Using a naive approach, we can shift each non-zero bits of c to the left to its corresponding position and sum them all together. If $c = \sum_{i=0}^{n-1} c_i 2^i$, this is equivalent to

$$c \times x = \sum_{i=0}^{n-1} c_i 2^i \times x = \sum_{i=0}^{n-1} c_i x 2^i. \quad (1)$$

For example, using the $\ll$ notation, we have

$$151x = (x \ll 7) + (x \ll 4) + (x \ll 2) + (x \ll 1) + x.$$

Such a constant multiplier by $c = 151$ would require 4 additions. In the general case, the number of additions is equal to the Hamming weight (i.e., the number of non-zero digits) of c minus 1. In the next section, we present some more sophisticated methods to perform a constant multiplication.

III. Previous Works

A widely used approach to reduce the number of non-zero digits, and therefore the number of additions, is to consider variants of Booth's recoding [1] technique, where long strings of ones are replaced with equivalent strings with many zeros. An improvement to the multiplication example presented above can be achieved if we represent our constant using signed digits. In the so-called Signed Digit (SD) binary representation, the constant c is expressed in radix 2, with digits in the set $\{\overline{1} = -1, 0, 1\}$. This recoding scheme is clearly redundant. A number is said to be in the Canonical Signed Digit (CSD) format if there are no consecutive non-zero digits in its SD representation. In this case, it can be proved that the number of non-zero digits is minimal among all SD representations [7]. For a n-bit[1] constant, it is bounded by $(n+1)/2$ in the worst case, and is roughly equal to $n/3$ on average (the exact value is $n/3 + 1/9$; see [8]). For example, since $151 = (10010111)_2 = (1010\overline{1}00\overline{1})_2$, the product $c \times x$ reduces to 3 additions:

$$151x = (x \ll 7) + (x \ll 5) - (x \ll 3) - x.$$

Representing the constant in a different format is known as a direct recoding method. The double-base encoding scheme we present in Section IV also falls into this type. Several other constant multiplication methods have been proposed in the literature. Solutions based on genetic algorithms such as evolutionary graph generation seem to provide very poor results. A drawback of typical recoding methods is the impossibility to reuse intermediate values. The first proposed method which takes advantage of intermediate computations is due to Bernstein [2], which is implemented in the GNU Compiler Collection (GCC) [9]. Methods based on pattern search in the binary or SD representation of the constant have also been widely studied. For example, in 2001, Lefèvre proposed an algorithm [3] to efficiently multiply a variable integer x by a given set of integer constants. This algorithm can also be used to multiply x by a single constant. Using similar techniques, Boullis and Tisserand [4] recently proposed improvements in the case of multiplication by constant matrices; a detailed presentation of all the methods mentioned above can be found in their respective papers with the corresponding references. Methods based on cost functions or pattern search generates optimized results at the expense of large computational time. In addition, lower bounds on the maximum left shifts must be considered carefully to minimize overflow – this to the detriment of the optimization. Another interesting method was proposed by MacLeod and Dempster [10] in 1999. It relies on graph generation, and again requires immense computational time as well as large memory requirements with the benefit of greatly optimized results.

IV. The Double-Base Number System

In this section, we present the main properties of the double-base number system, along with some numerical results to provide the reader with some intuitive ideas about this representation scheme. We have intentionally omitted the proofs of previously published results. The reader is encouraged to check the references for further details.

We will need the following definitions.

Definition 1 (S-integer): Given a set of primes S, an S-integer is a positive integer whose prime factors all belong to S.

Definition 2 (Double-Base Number System): Given p, q, two distinct prime integers, the double-base number system (DBNS) is a representation scheme into which every positive integer n is represented as the sum or difference of distinct $\{p, q\}$-integers, i.e., numbers of the form $p^a q^b$.

$$n = \sum_{i=1}^{\ell} s_i \, p^{a_i} q^{b_i}, \quad (2)$$

with $s_i \in \{-1, 1\}$, $a_i, b_i \geq 0$ and $(a_i, b_i) \neq (a_j, b_j)$ for $i \neq j$.

The size, or length, of a DBNS expansion is equal to the number ℓ of terms in (2). In the following, we will only consider bases $p = 2$ and $q \in \{3, 5, 7\}$.

Whether one considers signed ($s_i = \pm 1$) or unsigned ($s_i = 1$) expansions, this representation scheme is highly redundant. Indeed, if one considers unsigned double-base representations (DBNR) only, with bases 2 and 3, then one can prove that 10 has exactly 5 different DBNR; 100 has exactly 402 different DBNR; and 1000 has exactly 1295579 different DBNR. The following theorem holds.

[1] In the standard binary representation.

Theorem 1: Let n be a positive integer and let q be a prime > 2. The number of unsigned DBNR of n with bases 2 and q is given by $f(1) = 1$, and for $n \geq 1$

$$f(n) = \begin{cases} f(n-1) + f(n/q) & \text{if} \quad n \equiv 0 \pmod{q}, \\ f(n-1) & \text{otherwise.} \end{cases} \quad (3)$$

Remark: The proof consists of counting the number of solutions of the diophantine equation $n = h_0 + qh_1 + q^2h_2 + \cdots + q^th_t$, where $t = \lfloor \log_q(n) \rfloor$ and $h_i \geq 0$.

Not only this system is highly redundant, but it is also very sparse. Probably, the most important theoretical result about the double-base number system is the following theorem from [11], which gives an asymptotic estimate for the number of terms one can expect to represent a positive integer.

Theorem 2: Every positive integer n can be represented as the sum (or difference) of at most $O\left(\log n / \log\log n\right)$ $\{p, q\}$-integers.

The proof is based on Baker's theory of linear forms of logarithms and more specifically on the following result by R. Tijdeman [12].

Theorem 3: There exists an absolute constant C such that there is always a number of the form p^aq^b in the interval $\left[n - n/(\log n)^C, n\right]$.

Theorem 1 tells us that there exists very many ways to represent a given integer in DBNS. Some of these representations are of special interest, most notably the ones that require the minimal number of $\{p, q\}$-integers; that is, an integer can be represented as the sum of m terms, but cannot be represented with $(m-1)$ or fewer terms. These so-called canonic representations are extremely sparse. For example, with bases 2 and 3, Theorem 1 tells us that 127 has 783 different unsigned representations, among which 6 are canonic requiring only three $\{2, 3\}$-integers. An easy way to visualize DBNS numbers is to use a two-dimensional array (the columns represent the powers of 2 and the rows represent the powers of 3) into which each non-zero cell contains the sign of the corresponding term. For example, the six canonic representations of 127 are given in Table I.

Finding one of the canonic DBNS representations in a reasonable amount of time, especially for large integers, seems to be a very difficult task. Fortunately, one can use a greedy approach to find a fairly sparse representation very quickly. Given $n > 0$, Algorithm 1 returns a signed DBNR for n.

Although Algorithm 1 sometimes fails in finding a canonic representation (the smallest example is 41; the canonic representation is $32 + 9$, whereas the algorithm returns $41 = 36 + 4 + 1$) it is very easy to implement and it guarantees a representation of length $O\left(\log n / \log\log n\right)$.

The complexity of the greedy algorithm mainly depends on the complexity of step 3: finding the best approximation of n of the form p^aq^b. An algorithm based on Ostrowski's number system was proposed in [13]. It is possible to prove that its complexity is $O(\log\log n)$. (The algorithm proposed in [13] focuses on base 2 and 3 but the results extend to bases p, q.) Since Algorithm 1 finishes in $O(\log n / \log\log n)$ iterations, its overall complexity is thus optimal in $O(\log n)$. Another solution for Step 3 was recently proposed by Doche and Imbert in [14]; it uses lookup tables containing the binary representations of some powers of q and can be implemented very quickly, even for large numbers.

Algorithm 1 Greedy algorithm

INPUT: A positive integer n
OUTPUT: The sequences $(s_i, a_i, b_i)_{i\geq 0}$ s.t. $n = \sum_i s_i p^{a_i} q^{b_i}$ with $s_i \in \{-1, 1\}$, $a_i, b_i \geq 0$ and $(a_i, b_i) \neq (a_j, b_j)$ for $i \neq j$

1: $s \leftarrow 1$ {To keep track of the sign}
2: while $n \neq 0$ do
3: Find the best approximation of n of the form $z = p^aq^b$
4: print (s, a, b)
5: if $n < z$ then
6: $s \leftarrow -s$
7: $n \leftarrow |n - z|$

V. SUBLINEAR CONSTANT MULTIPLICATION

In this core section, we propose a generic algorithm for constant multiplication that takes advantage of the sparseness of the double-base encoding scheme. Our algorithm computes a special DBNS representation of the constants, where the largest exponent of the second base q is restricted to an arbitrary (small) value B. It uses a divide and conquer strategy to operate on separate blocks of small sizes. For each block, it is possible to generate those specific DBNS representations using a modified version of the greedy algorithm, or to precompute and store them in a lookup table in a canonical form; i.e., a DBNS expansion with a minimal number of terms. We show that both approaches lead to sublinear constant multiplication algorithms.

Let us illustrate the algorithm on a small example. We express $c = 10599 = (10100101100111)_2$ in radix 2^7; that is, we split c in two blocks of 7 bits each. We obtain $c = 82 \times 2^7 + 103$ and we represent the "digits" 82 and 103 in DBNS with bases 2 and 3 using as few terms as possible, where the exponents of the second base $q = 3$ are at most equal to 2. We find that 82 can be written using two terms as $64 + 18$ and 103 using only three terms as $96 + 8 - 1$. (We have results which prove that these values are optimal). By sticking the two parts together, we obtain the representation given in Table II.

Using this representation, the product $c \times x$ is decomposed as follows:

$$x_0 = (x \ll 8)$$
$$x_1 = 3x_0 + (x \ll 5)$$
$$x_2 = 3x_1 + (x \ll 13) + (x \ll 3) - x$$

Since multiplications by 3 can be performed by a shift

TABLE I

THE SIX CANONIC UNSIGNED DBNR OF 127

$2^2 3^3 + 2^1 3^2 + 2^0 3^0 = 108 + 18 + 1$

	1	2	4
1	1		
3			
9		1	
27			1

$2^2 3^3 + 2^4 3^0 + 2^0 3^1 = 108 + 16 + 3$

	1	2	4	8	16
1					1
3	1				
9					
27			1		

$2^5 3^1 + 2^0 3^3 + 2^2 3^0 = 96 + 27 + 4$

	1	2	4	8	16	32
1			1			
3						1
9						
27	1					

$2^3 3^2 + 2^1 3^3 + 2^0 3^0 = 72 + 54 + 1$

	1	2	4	8
1	1			
3				
9				1
27		1		

$2^6 3^0 + 2^1 3^3 + 2^0 3^2 = 64 + 54 + 9$

	1	2	4	8	16	32	64
1							1
3							
9	1						
27		1					

$2^6 3^0 + 2^2 3^2 + 2^0 3^3 = 64 + 36 + 27$

	1	2	4	8	16	32	64
1							1
3							
9			1				
27	1						

TABLE II

A DBNS REPRESENTATION OF $c = 10599$ OBTAINED USING TWO BLOCKS OF 7 BITS EACH

	2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}
3^0	-1			1										1
3^1						1								
3^2									1					
	103							82						

followed by an addition, the resulting sequence of shift-and-add becomes:

$$x_0 = (x \ll 8)$$
$$x_1 = ((x_0 \ll 1) + x_0) + (x \ll 5)$$
$$x_2 = ((x_1 \ll 1) + x_1) + (x \ll 13) + (x \ll 3) - x$$

Let us give a formal description of the algorithm outlined in the previous example. We express c in DBNS as

$$c = \sum_{j=0}^{b_{max}} \sum_{i=0}^{a_{max}} c_{i,j} 2^i q^j, \tag{4}$$

with digits $c_{i,j} = \{\overline{1}, 0, 1\}$. Algorithm 2 below can be used to compute $c \times x$. We remark that each step of the algorithm requires a multiplication by q. It is therefore important to select the second base q such that the multiplication by q only requires a single addition; i.e., with $q = 3$, we have $3x = (x \ll 1) + x$. At the end, the result is given by $x_{b_{max}} = c \times x$. If ℓ is the length of the double-base expansion; i.e, the number of non-zero digits $c_{i,j}$ in (4), and if b_{max} is the largest exponent of q, then the overall number of additions is equal to

$$\ell + b_{max} - 1. \tag{5}$$

The goal is to set B, the predefined upper bound for the

Algorithm 2 Double-base constant multiplication

INPUT: A constant $c = \sum_{i,j} c_{i,j} 2^i 3^j$, with $c_{i,j} = \{\overline{1}, 0, 1\}$; and an integer x

OUTPUT: $c \times x$

1: $x_{-1} \leftarrow 0$
2: **for** $j = 0$ to b_{max} do
3: $\quad x_j \leftarrow q \times x_{j-1}$
4: $\quad x_j \leftarrow x_j + \sum_i c_{i,b_{max}-j}\ (x \ll i)$
5: Return $x_{b_{max}}$

exponents of q, such that the overall number of addition is minimal. (Note that b_{max} might be different from B, but $b_{max} \leq B$ holds.)

The following theorem shows that the number of additions required to evaluate the product $c \times x$ using our algorithm is sublinear in the size of the constant c.

Theorem 4: Let c be a positive integer constant of size n (in bits). Then, the multiplication by c can be computed in $O(n/\log n)$ additions.

Proof: We split c in blocks of size $n/\log n$ bits each. Clearly, one needs $\log n$ such blocks. Each block corresponds to an integer of size $n/\log n$ bits and can thus be represented in DBNS with exponents all less than $n/\log n$. In particular, we have $b_{max} \in O(n/\log n)$. From Theorem 2, we know that the number of non-zero digits in the DBNS representations of each block belongs to $O\left(n/(\log n)^2\right)$. Note that this is true whether one uses the greedy algorithm or considers a canonic double-base representation for each block. Therefore, since we have $\log n$ blocks, the number of non-zero digits in the DBNS representation of c belongs to $O\left(n/\log n\right)$. From (5), since $b_{max} \in O(n/\log n)$, we obtain that the overall complexity of the algorithm is in $O(n/\log n)$. □

VI. HEURISTICS

The algorithm presented in the previous section must be seen as a generic method; it must be adapted for each specific application. In particular, there are several parameters that need to be defined carefully: the second base q, the upper bound B on the exponents of q, and the size of the blocks.

As mentioned previously, when the block size is not too large, it is possible to store the canonic representations of each possible number in the range in a lookup table. The values given in Table III have been computed using exhaustive search. For integers of size up to 21 bits, we report the average number of non-zero digits in canonic double base representation with bases 2 and $q = 3$, maximum binary exponent $a_{max} = 19$ and ternary exponent $b_{max} \in \{3, 4, 5\}$. We also give the maximum number of terms ever needed in the corresponding range and the first integer x for which it occurs[2].

Let us analyze some options offered by our algorithm for a multiplication by a 64-bit constant c. For this example, we only consider bases 2 and 3 with maximum ternary exponent $b_{max} = 3$. If we split our 64-bit constant in 7 blocks of 10 bits, we know from Table III that, in the worst case, the DBNS decomposition of c will require $7 \times 3 = 21$ non-zero digits. Therefore, we know that the number of additions will be ≤ 23. If, instead, we consider four blocks of 16 bits each, we obtain 22 additions in the worst case. We remark that our worst case is similar to the average complexity if one uses the CSD representation ($64/3 \simeq 21.3333$). The average number of operations in our case is roughly equal $3.64 \times 4 + 2 \simeq 16.56$, which represents a speedup of about 22% compared to the CSD approach. This is the kind of analysis a programmer should do in order to define an appropriate set of parameters for his specific problem.

This approach is encouraging but is is possible to do better. In the algorithm presented so far, the blocks are all of the same size. This is to the detriment of efficiency since there might exist better way to split the constant than these regular splitting. In the next two sections, we present to families of heuristics that operates from right-to-left or from left-to-right.

A. Right-to-left splitting

The regular splitting does not exploit the nature of the binary representation of the constant c. The idea here is to try to avoid blocks with long strings of zeros and rather use these strings to split the constant c. For a predefined integer $m > 1$, we define a separating string as a string of m consecutive zeros. The heuristic works as follows: starting from the least significant bit, we look for the first separating string. If such a string is found at position j, the first block corresponds to the bits of c of weight less than j. We then look for the next separating string starting from the first non-zero digit of weight $> j$. Therefore, every block is an odd number and there is no need to store the canonic representations of even numbers in our lookup table. The separating size m must be carefully chosen in function of the size of c. If it is too small, there will be too many blocks and the overall number of additions will increase accordingly. Reversely, if it is too large, there might not be any such strings and we might end up with the entire constant c, for which we do not know a canonic DBNS representation. In the second case, the solution is to fix the largest block size (according to the amount of memory available for the lookup table) and to split the constant c either when we find a separating string or when we reach this largest block size. In Figures 1 and 2, we have plotted the average number of additions as a function of the largest block size for $m = 2, 3, 4, 5$, for $a_{max} = 19$ and $b_{max} = 3$, for 100000 random 32-bit and 64-bit constants. We also have similar plots for $b_{max} = 2, 4, 5$ but $b_{max} = 3$ seems to give the best results.

B. Left-to-right splitting

Another possible strategy is to start from the most significant bit of c and to look for the largest odd number of size less than the predefined largest block size. As previously, we impose that each block starts and finishes with a non-zero digit in order to store odd numbers only. This strategy might

[2]We also have similar data for $q = 5$ and $q = 7$.

TABLE III

NUMERICAL RESULTS FOR PARAMETERS $q = 3$, $a_{max} = 19$, $b_{max} = 3, 4, 5$

size	$b_{max} = 3$			$b_{max} = 4$			$b_{max} = 5$		
(in bits)	Avg	Max	at $x =$?	Avg	Max	at $x =$?	Avg	Max	at $x =$?
1	0.5	1	1	0.5	1	1	0.5	1	1
2	0.75	1	1	0.75	1	1	0.75	1	1
3	1.125	2	5	1.125	2	5	1.125	2	5
4	1.375	2	5	1.375	2	5	1.375	2	5
5	1.5625	2	5	1.5625	2	5	1.5625	2	5
6	1.71875	2	5	1.71875	2	5	1.71875	2	5
7	1.89844	3	77	1.84375	3	103	1.83594	3	103
8	2.10547	3	77	2.02734	3	103	1.96875	3	103
9	2.31836	3	77	2.23047	3	103	2.15234	3	103
10	2.51074	3	77	2.42773	3	103	2.34961	3	103
11	2.68408	4	1229	2.59863	4	1843	2.52881	3	103
12	2.86743	4	1229	2.75391	4	1843	2.67871	4	2407
13	3.06897	4	1229	2.92224	4	1843	2.81982	4	2407
14	3.27203	4	1229	3.10913	4	1843	2.97766	4	2407
15	3.46136	5	19661	3.29990	5	29491	3.15594	4	2407
16	3.64391	5	19661	3.47905	5	29491	3.33882	5	52889
17	3.83374	5	19661	3.64627	5	29491	3.50820	5	52889
18	4.03194	5	19661	3.81557	5	29491	3.66204	5	52889
19	4.22856	6	314573	3.99545	6	471859	3.81476	5	52889
20	4.44634	6	314573	4.20838	6	471859	3.99837	5	52889
21	4.67745	6	314573	4.41817	6	471859	4.19770	6	1103161

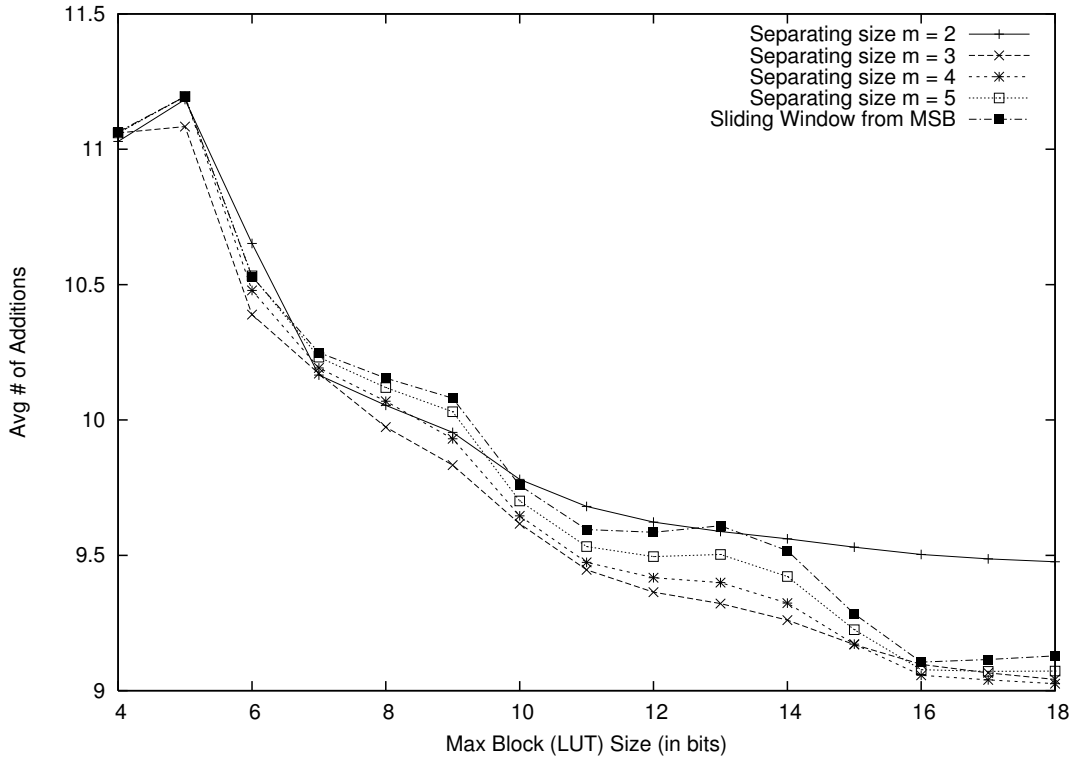

Fig. 1. Average number of additions for 100000 randomly chosen 32-bit constants, using bases 2 and 3, with $a_{max} = 19$ and $b_{max} = 3$

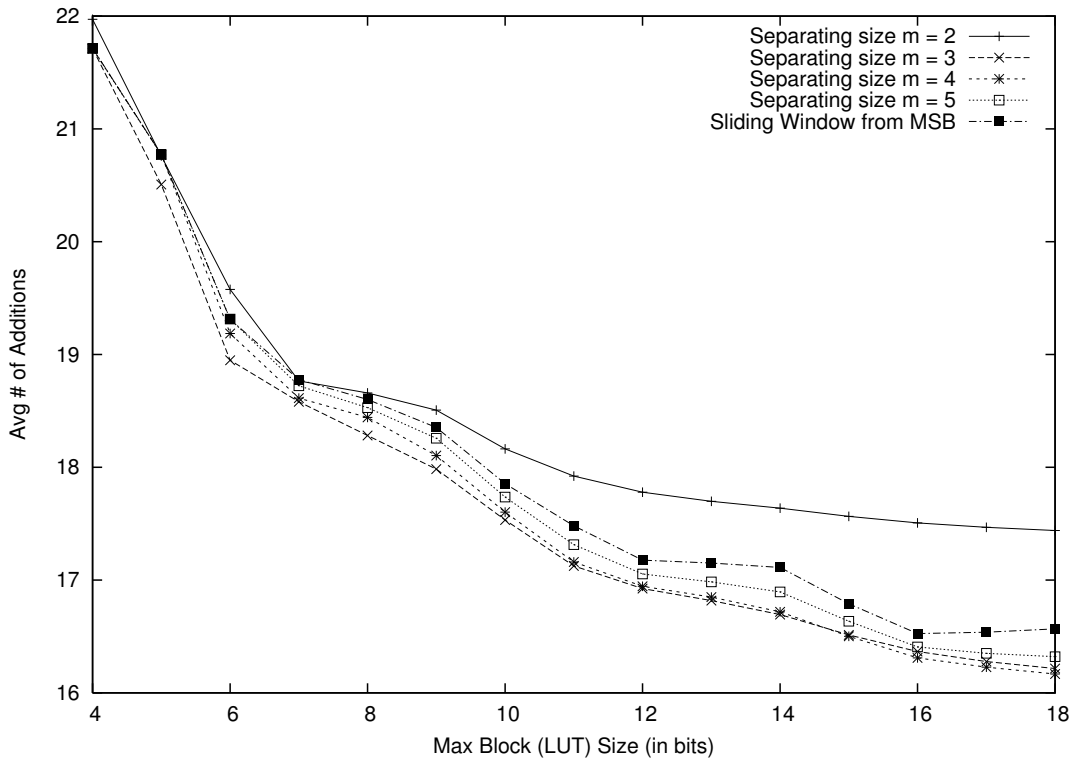

Fig. 2. Average number of additions for 100000 randomly chosen 64-bit constants, using bases 2 and 3, with $a_{max} = 19$ and $b_{max} = 3$

look optimal as it best exploit the precomputed values, but Figures 1 and 2 show that this is not the case.

C. Remarks

1) In Figures 1 and 2, we remark that for 32-bit and 64-bit constants, with lookup tables of reasonable size (10 to 12 input bits), the best results seems to be given for separating string of size $m = 3$.
2) In Table IV, we give the average number of additions and the worst case for 100000 randomly chosen 64-bit constants (with separating size $m = 3$). We remark that Lookup tables of 10 to 12 input bits lead to roughly 17 additions on average and 22 in the worst case. Using much larger lookup tables only provides small improvements. For 64-bit constants, lookup tables of 10 to 12 input bits seems to be a good choice. For 32-bit numbers, tables of size 8 to 10 input bits lead to < 10 additions on average and 13 in the worst case.
3) We have performed the same kind of experiments with second base $q = 5$ and $q = 7$. Bases 2 and 3 seem to provide the best results.
4) In terms of comparisons, our recoding algorithm requires more additions, both on average and in the worst case, than Boullis and Tisserand's algorithm [4] (using the graph heuristic strategy); which is the best known algorithm so far for multiplication by constant matrices. Using their approach for a single constant, one get about 13.5 additions on average and 19 in the worst case. This is not surprising since their approach based on pattern search generates very optimized results. However, the computational cost of our DBNS recoding algorithm, both in time and memory, is smaller, which might allow its use in compilers.

TABLE IV
AVERAGE AND WORST CASE NUMBER OF ADDITIONS FOR 64-BIT CONSTANTS

Max block size	Avg # add	Worst case
4	21.7133	31
5	20.5069	28
6	18.9489	24
7	18.5809	26
8	18.2813	25
9	17.9844	24
10	17.5323	22
11	17.1257	22
12	16.9249	23
13	16.818	22
14	16.694	21
15	16.5134	21
16	16.366	22
17	16.277	21
18	16.2151	21

5) Note that it is possible to reduce the average and worst case number of additions. Indeed, the canonic representations stored in the lookup tables we used for our experiments are not the "best" possible ones; i.e., among all the canonic representations for a given number, we do not necessarily store the representation with the smallest second exponent. By doing so, we can probably save some additions.

VII. Conclusions

In this paper, we proposed a new recoding algorithm for the constant multiplication problem. Our approach uses a divide and conquer strategy combined with the double-base number system. We proved that our approach leads to a sublinear algorithm. To our knowledge, this is the first sublinear algorithm for the constant multiplication problem. We illustrate the potential interest of our approach with several numerical experiments. The sequence of shifts-and-adds obtained with our algorithm is not as "good" as the sequences obtained with the best known methods based on pattern search or cost funcions. However, our DBNS-based generation algorithm requires much less computational effort than these optimal methods and it gives better results than the other direct recoding methods. A natural extension to the problem is the multiplication by constant vectors and matrices, where the high redundancy of the DBNS can certainly be exploited.

Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was partly funded by an NSERC strategic grant on the Canadian side and by an ACI grant on the French side.

References

[1] A. D. Booth, "A signed binary multiplication technique," *Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951, reprinted in E. E. Swartzlander, *Computer Arithmetic*, Vol. 1, IEEE Computer Society Press Tutorial, Los Alamitos, CA, 1990.

[2] R. Bernstein, "Multiplication by integer constants," *Software – Practice and Experience*, vol. 16, no. 7, pp. 641–652, jul 1986.

[3] V. Lefèvre, "Multiplication by an integer constant," INRIA, Research Report 4192, May 2001.

[4] N. Boullis and A. Tisserand, "Some optimizations of hardware multiplication by constant matrices," *IEEE Transactions on Computers*, vol. 54, no. 10, pp. 1271–1282, Oct. 2005.

[5] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "Theory and applications of the double-base number system," *IEEE Transactions on Computers*, vol. 48, no. 10, pp. 1098–1106, Oct. 1999.

[6] V. Dimitrov, L. Imbert, and P. K. Mishra, "Efficient and secure elliptic curve point multiplication using double-base chains," in *Advances in Cryptology, ASIACRYPT'05*, ser. Lecture Notes in Computer Science, vol. 3788. Springer, 2005, pp. 59–78.

[7] G. W. Reitwiesner, "Binary arithmetic," *Advances in Computers*, vol. 1, pp. 231–308, 1960.

[8] R. I. Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677–688, Oct. 1996.

[9] "GCC, the GNU compiler collection," http://www.gnu.org.

[10] A. G. Dempster and M. D. Macleod, "Constant integer multiplication using minimum adders," *IEE Proc. Circuits Devices Syst.*, vol. 141, no. 5, pp. 407–413, 1994.

[11] V. S. Dimitrov, G. A. Jullien, and W. C. Miller, "An algorithm for modular exponentiation," *Information Processing Letters*, vol. 66, no. 3, pp. 155–159, May 1998.

[12] R. Tijdeman, "On the maximal distance between integers composed of small primes," *Compositio Mathematica*, vol. 28, pp. 159–162, 1974.

[13] V. Berthé and L. Imbert, "On converting numbers to the double-base number system," in *Advanced Signal Processing Algorithms, Architecture and Implementations XIV*, ser. Proceedings of SPIE, vol. 5559. SPIE, 2004, pp. 70–78.

[14] C. Doche and L. Imbert, "Extended double-base number system with applications to elliptic curve cryptography," in *Progress in Cryptology, INDOCRYPT'06*, ser. Lecture Notes in Computer Science, vol. 4329. Springer, 2006, pp. 335–348.