Setting Speed Records with the (Fractional) Multibase Non-Adjacent Form Method for Efficient Elliptic Curve Scalar Multiplication

Patrick Longa and Catherine Gebotys

Department of Electrical and Computer Engineering
University of Waterloo, Canada
{plonga,cgebotys}@uwaterloo.ca

Abstract. In this paper, we introduce the Fractional Window-*w* Multibase Non-Adjacent Form (Frac-*wmb*NAF) method to perform the scalar multiplication. This method generalizes the recently developed Window-*w mb*NAF (*wmb*NAF) method by allowing an unrestricted number of precomputed points. We then make a comprehensive analysis of the most recent and relevant methods existent in the literature for the ECC scalar multiplication, including the presented generalization and its original non-window version known as Multibase Non-Adjacent Form (*mb*NAF). Moreover, we present new improvements in the point operation formulae. Specifically, we reduce further the cost of composite operations such as doubling-addition, tripling, quintupling and septupling of a point, which are relevant for the speed up of methods using multiple bases. Following, we also analyze the precomputation stage in scalar multiplications and present efficient schemes for the different studied scenarios. Our analysis includes the standard elliptic curves using Jacobian coordinates, and also Edwards curves, which are gaining growing attention due to their high performance. We demonstrate with extensive tests that *mb*NAF is currently the most efficient method without precomputations not only for the standard curves but also for the faster Edwards form. Similarly, *Frac-wmb*NAF is shown to attain the highest performance among window-based methods for all the studied curve forms.

Keywords: Elliptic curve cryptosystem, scalar multiplication, multibase non-adjacent form, fractional windows, point operation, composite operation, precomputation scheme.

1 Introduction

Since Koblitz and Miller independently proposed the use of elliptic curves in cryptography [Kob87,Mil85], giving birth to a new area of study known as Elliptic Curve Cryptography (ECC), the elegant elliptic curve-based cryptosystems have attracted increasing attention from the research and industry communities. Contrary to what is currently known for other traditional public-key cryptosystems such as RSA, the problem of solving discrete logarithms on the group of points on an elliptic curve over a finite field in subexponential time is intractable. Hence, EC-based cryptosystems can attain equivalent security levels to RSA with significantly smaller cryptographic parameters. For instance, it is widely accepted that 160-bit ECC offers equivalent security to 1024-bit RSA. This significant difference makes ECC especially attractive for applications in constrained environments as shorter key sizes are translated to less storage requirements and reduced computing times.

Scalar multiplication, denoted by kP, where P is a point on the elliptic curve and k is a scalar (working as the secret key in some cryptographic protocols), is the central and most resource demanding operation in many EC-based systems. Hence, its efficient implementation has been the focus of intensive research during the last few years.

Well-known methods [Rei60,Sol00] to efficiently execute kP are Non-Adjacent Form (NAF) and window-w NAF (wNAF), which use short signed radix 2-based representations of the scalar to attempt to minimize the number of point operations, namely doubling of a point ($2P$) and addition of points ($P+Q$). In particular, wNAF offers very high performance at the cost of a few precomputations.

Recently, there have been new methods proposed for scalar multiplication using numeric representations that mix radices 2 and 3 [DIM05,DI06], and 2, 3 and 5 [MD07a]. We will refer to these methods as Double-Base (DB) and Triple-Base (TB) respectively. Since these methods use very compact representations in their number of terms, the number of point additions to perform the scalar multiplication is significantly reduced, which is expected to reduce the computational cost of kP. Very recently, new research expanding and demonstrating the capabilities of DB, especially focusing on window-based variants, have appeared in the literature [BBP07,MD07b,BBL$^+$07].

However, DB and TB (and their variants) have critical shortcomings, questioning the real usefulness of these methods in practice. First, conversion to DB (or TB) representation is based on a "Greedy" algorithm that basically searches for the closest {2,3} (or {2,3,5}) terms in a pre-stored table. In the best case scenario, this approach requires more memory resources. Second, from a theoretical perspective, these schemes are difficult to define in terms of the expected number of nonzero terms. In fact, the performance of most efficient DB schemes is determined empirically. Furthermore, expansions are constructed with a heuristic selection of the upper bounds for exponents of powers of 2 and 3 (and 5).

On the other hand, in a recent effort Longa [Lon07] introduced a new generic multibase representation that also mixes radices to represent the scalar (see also Longa and Miri [LM08]). However, the main difference with previous works is that this representation uses a non-adjacent form (similar to NAF), and consequently, its conversion process becomes efficient, and its theoretical definition, straightforward. We will refer to this method as Multibase Non-Adjacent Form (*mb*NAF). Its window-based version with an extended set of precomputations appears as a natural extension and will be referred to as Window-*w* *mb*NAF (*wmb*NAF).

Nevertheless, *wmb*NAF restricts the allowed number of precomputed points to

$(2^{w-2}-1)$ (without including $\{0,\pm1\}$), where $w>2\in\mathbb{Z}^+$, following the same restriction of its analogous counterpart in the radix-2 domain, namely *w*NAF. In some settings, it is possible that the optimal performance is achieved by precomputing a number of points that do not follow such a standard window size.

To solve that problem, in this paper we propose the Frac-*wmb*NAF method, which generalizes *wmb*NAF to any number of precomputed points by following a similar idea applied by Möller [Möl02] when generalizing *w*NAF to his well-known Fractional *w*NAF method (denoted by Frac-*w*NAF).

We then focus on demonstrating the performance of the *mb*NAF and Frac-*wmb*NAF methods to compute the elliptic curve scalar multiplication, specifically in the case where the point *P* to perform *kP* is not known in advance, as happens in several cryptographic systems such as the ElGamal encryption and the Diffie-Hellman key exchange. We analyze the performance of these methods and compare to that of NAF, DB and TB (and their variants) in the context of the standard form of an elliptic curve over prime fields (see eq. (1)). Then, we extend our analysis to other very efficient elliptic curve forms proposed recently, namely Edwards curves [Edw07].

It should be noted that in the case that *P* is fixed, very efficient methods based on precomputations already exist (see [HMV04, pp. 103] for further details). As a consequence, precomputations will be included in the total cost of the methods (window-based schemes) with the objective of quantifying true performance. Such a comprehensive analysis has not been included in most previous research [DIM05,DI06,MD07a,MD07b,LM07c]).

Unlike previous research, this paper proposes new and/or applies existing precomputation schemes that are the most efficient, to the best of our knowledge, for each scalar multiplication method. Also, the state-of-the-art in point operation formulae is applied to estimate the overall cost of the scalar multiplication. In that sense, the paper includes formulas by Bernstein and Lange for Edwards coordinates [BL07a] and Inverted Edwards coordinates [BL07b], and formulas by Longa and Miri [LM07a,LM07b] for Jacobian coordinates. We also include new formulas introduced here using radix-3 in the context of Edwards curves.

Furthermore, this paper extends the work presented in [LM07b,LM07c] and introduces revised formulas for computing the doubling-addition (2*P*+*Q*), and the tripling (3*P*), quintupling (5*P*) and septupling (7*P*) of a point. The superior performance of these new operations will be shown to further speed up the computation of all methods discussed in this work including the traditional NAF and *w*NAF, in the case of standard elliptic curves using Jacobian coordinates.

Extensive tests in the last section of this work will be shown to demonstrate the superior performance of *mb*NAF over all other schemes without precomputations. On the other hand, Frac-*wmb*NAF will be shown to achieve the lowest costs when some extra memory is available. Our comparisons take into account all the improvements introduced and described throughout this work, including efficient point operations and precomputation schemes.

Our work is organized as follows. In Section 2, we detail some background about ECC over prime fields. Then, in the following section we describe the state-of-the-art point formulae for Jacobian coordinates, giving some extra details about composite operations. Improvements to these operations are discussed in this section. In Section 4, we give a quick overview about Edwards curves, summarizing the state-of-the-art point formulae in this setting. We introduce here a few new formulas using radix-3. In the following section, a detailed analysis of traditional and very recent scalar multiplication methods is carried out. We highlight drawbacks that make some of them potentially impractical. The *mb*NAF and *wmb*NAF methods are also described in this section. In Section 6, we introduce the new Frac-*wmb*NAF method for the scalar multiplication, highlighting its advantages and high performance. In the following section, we discuss efficient precomputation schemes for the different studied methods. These are then used in Section 8 to evaluate the cost performance of various methods for scalar multiplication through extensive tests. Some conclusions summarizing the contributions of this work are presented at the end.

2 Preliminaries

Here we work with an elliptic curve E over a prime field $\mathbb{F}_p$ (denoted by $E(\mathbb{F}_p)$), which is defined by the reduced Weierstrass equation [HMV04]:

$$E: \quad y^2 = x^3 + ax + b , \tag{1}$$

where a , $b \in \mathbb{F}_p$ and $\Delta = 4a^3 + 27b^2 \neq 0$. Δ denotes the discriminant, and its inequality with zero guarantees that a unique tangent line exists for each point on the elliptic curve E. The latter, together with a few other additive rules (see [HMV04], pp. 79, for further details) allows for the conformation of the so called “group law”, which mainly consists of two basics point operations: the doubling of a point (2*P*) and the addition of two points (*P*+*Q*). In the remainder, we will refer to (1) as the standard elliptic curve form.

The set of pairs (x, y) that solves (1), where $x, y \in \mathbb{F}_p$, together with the point at infinity O, which works as the identity for the “group law”, form an *abelian* group, ($E(\mathbb{F}_p)$,+), on top of which the basic ECC point operations are performed.

The representation using (x, y) , known as affine coordinates, introduces field inversions (*I*) during the computation of point operations. In efficient implementations, we would like to get rid of these expensive operations. To that end, new point representations with the form (X, Y, Z) , known as projective coordinates, were introduced to replace inversions by a few cheaper field operations by means of a third coordinate, *Z*.

The foundation of these inversion-free coordinate systems can be explained by the concept of equivalence classes, which are defined in the following in the context of *Jacobian* coordinates, a special case of projective coordinates that has yielded very efficient inversion-free point operations.

Given a prime field $\mathbb{F}_p$, there is an equivalence relation ~ among nonzero triplets over

$\mathbb{F}_p$, such that [HMV04, pp. 86]:

$(X_1,Y_1,Z_1) \sim (X_2,Y_2,Z_2) \Leftrightarrow X_1 = \lambda^2 X_2$, $Y_1 = \lambda^3 Y_2$ and $Z_1 = \lambda Z_2$, for some $\lambda \in \mathbb{F}_p^*$.

Thus, the equivalence class of a (Jacobian) projective point, denoted by $(X : Y : Z)$, is:

$$(X : Y : Z) = \{(\lambda^2 X,\ \lambda^3 Y,\ \lambda Z) : \lambda \in \mathbb{F}_p^*\} \tag{2}$$

It is important to remark that any triplet (X, Y, Z) in the equivalence class (2) can be used as a representative of a given (Jacobian) projective point. This fact has also been exploited to reduce further the cost of point operations by replacing multiplications for squarings (see [LM07a]). In this work, we again make use of such a flexible technique to modify point formulas in Section 3.

3 Point Operations in Jacobian Coordinates

In this section, we summarize the state-of-the-art point formulae for standard ECC curves using Jacobian coordinates, and compare it with previous efforts. Further, we introduce additional speed ups to formulas for doubling-addition (DA), tripling, quintupling and septupling of a point in Jacobian coordinates, and also present new formulas using mixed coordinate systems. For instance, we present formulas to compute $3\mathcal{A} \to \mathcal{J}$ and $5\mathcal{A} \to \mathcal{J}$, where $\mathcal{A}$ and $\mathcal{J}$ denote points in affine and Jacobian coordinates, respectively. This development is critical to boost the performance not only of traditional scalar multiplications methods such as NAF but also, and more critically, of methods using radices 2, 3 and 5. See [EPAF] for an up-to-date compilation of the most efficient formulas to perform point arithmetic on standard EC curves using Jacobian coordinates.

For the remainder of this work, we will follow the general approach of expressing the cost of point operations in terms of field multiplications (*M*) and squarings (*S*), disregarding the cost of field addition/subtractions (*A*) and multiplication/divisions by small constants as negligible for simplification purposes. We however remark that such cheaper operations do have an impact in the total cost depending on their relative ratio with regard to multiplication (which depends on the targeted platform). Hence, we will discuss their influence in the cost of point formulas whenever relevant.

Also, to determine the superiority of a given formula, we will assume that $1S \approx 0.8M$, which is widely assumed in the literature. In some implementations, however, it has been observed that the *S*/*M* ratio can be as low as 0.6. The reader must note that the improvements introduced in this work are actually more advantageous in such scenario.

3.1 Basic Point Operations

Researchers in [LM07a] presented a simple but highly flexible technique to trade multiplications for squarings and other few cheaper operations. They combined the algebraic relation over a prime field

$$2ab = (a+b)^2 - a^2 - b^2 \tag{3}$$

with a transformation of the point formulae by means of an equivalent representative of the form

$$(X : Y : Z) = \{(2^2 X, 2^3 Y, 2Z)\}. \tag{4}$$

The algebraic substitution (3) allows trading *one* multiplication for *one* squaring, assuming that the remainder squarings are already included in a given formula. On the other hand, if a field multiplication that can be potentially replaced using (3) does not have the even form $2ab$, where $a, b \in \mathbb{F}_p$, we first transform the formula using the form given by (4) to introduce the necessary multiples of 2.

Table 1 summarize the costs of the new formulae by [LM07a] using the described technique, and compare it with the previous point operation costs. We consider the well-known general (random a) and special ($a = -3$) cases, and also an additional scenario when the parameter a is defined as sparse and any multiplication with it is considered negligible (it includes the case $a = -3$). Note that the latter is convenient for implementations where squarings are very fast (e.g., $1S \approx 0.6M$).

Table 1. Costs of revised and traditional point operations using Jacobian coordinates.

Operation	general	a sparse	$a = -3$
Fast doubling [LM07a]	$2M + 8S$	$1M + 8S$	$3M + 5S$
Traditional doubling	$4M + 6S$	$3M + 6S$	$4M + 4S$
Fast addition [LM07a]	$11M + 5S$	-	-
Traditional addition	$12M + 4S$	-	-
Fast mixed addition [LM07a]	$7M + 4S$	-	-
Traditional mixed addition	$8M + 3S$	-	-

3.2 Composite Operations

Composite operations are point operations of the form dP, where $d > 2 \in \mathbb{Z}$, or the form $dP+Q$, where $d \geq 2 \in \mathbb{Z}$. They receive this name because these more complex operations are usually built on top of the basic point doubling and addition.

We will summarize the best results for cases d = 3, 5 and 7, which have a special importance since the performance of these composite operations allow the efficient realization of multibase methods for the scalar multiplication.

In the case of tripling, [DIM05] introduced a formula in Jacobian coordinates with a cost of $10M + 6S$. A quintupling formula was presented by [MD07a] with costs of $15M + 8S$ and $15M + 10S$ for the special and general cases, respectively.

Later on, in [LM07a] and [LM08] Longa and Miri reduced the cost of the previous formulas by rearranging terms optimally and applying the technique described in Section 3.1 to trade multiplications for squarings. The result yielded tripling formulas with costs of $7M + 7S$ (respect. $6M + 10S$) for the special case (respect. general case), and quintupling formulas with costs of $11M + 11S$ (respect. $10M + 14S$) for the special case (respect. general case).

A different approach, presented in [LM07b,LM07c], is based on a special addition with identical z-coordinate. The idea of this approach is to derive composite operations of the form dP and $dP+Q$, where $d \geq 2$ is prime, by iteratively adding points using the aforementioned special addition as follows:

$$dP = (2P + (\ldots + (2P + (2P + P)) \ldots)). \tag{5}$$

$$dP + Q = (P + (\ldots + (P + (P + (P+Q))) \ldots)). \tag{6}$$

The special addition with identical z-coordinate due to [Mel06] is now described. Let $P = (X_1, Y_1, Z)$ and $Q = (X_2, Y_2, Z)$ be two points with the same Z in Jacobian coordinates on the standard elliptic curve (1). The addition $P + Q = (X_3, Y_3, Z_3)$ can be obtained as follows:

$$\begin{aligned} X_3 &= (Y_2 - Y_1)^2 - (X_2 - X_1)^3 - 2X_1(X_2 - X_1)^2 . \\ Y_3 &= (Y_2 - Y_1)\left(X_1(X_2 - X_1)^2 - X_3\right) - Y_1(X_2 - X_1)^3 . \\ Z_3 &= Z(X_2 - X_1) . \end{aligned} \tag{7}$$

As can be seen, this new addition formula (derived from the original addition formula in Jacobian coordinates) is surprisingly simple and only costs $5M+2S$.

The authors in [LM07b] shown that we first require one doubling followed by $(d-1)/2$ special additions (7) to compute dP. In this case, the cost can be expressed by

$$1\text{D} + \left(\frac{d-1}{2}\right)(5M + 2S), \tag{8}$$

where D = $3M + 5S$ or $1M + 8S$ for the special and general cases, respectively (see Table 1). Remarkably, composite operations of the form $dP+Q$ using the previous strategy do not use conventional doubling, tripling or quintupling operations. Hence, the cost given by (8) applies indistinctly to the general and special cases without caring about the value of the parameter a (eq. (1)).

Similarly, we require one addition ($P+Q$) and then $(d-1)$ special additions (7) with P to perform $dP+Q$ [LM08]. Thus, the cost can be expressed by

$$\mathrm{A} + (d-1)(5M + 2S)\,, \tag{9}$$

where A represents the cost of a point addition. We remark here that such addition is a mixed addition with Jacobian-affine coordinates (with a cost of 7*M* + 4*S*; see Table 1) if the point *Q* is in affine coordinates, or a general addition (with a cost of 11*M* + 5*S*) if otherwise *Q* is in Jacobian coordinates.

In [LM07b] and [LM08], the researchers shown that the new formulas yielded by combining (7) with the schemes (5) and (6) are faster and/or require very low memory resources.

In the following, we describe a procedure to accelerate further these composite operations.

Additional Cost Reductions

The methods described previously respect the structure of each operation, which keeps the internal execution modular and simple. However, we notice here that the number of field additions can be reduced by avoiding intermediate computations. Further, because of the latter, new multiplications are generated in such a way that more substitutions of field multiplications for squarings can be carried out.

Following, we illustrate this procedure with the DA operation (2*P*+*Q*), when points *P* and *Q* are in Jacobian and affine coordinates, respectively. Application of this idea to other cases, such as 2*P*+*Q* when both *P* and *Q* are in Jacobian coordinates (referred to as general DA), and *dP* (i.e., tripling, quintupling and so on), easily follows.

The first addition in (6) is computed with a mixed Jacobian-affine addition [CMO98] $P+Q=(X_1,Y_1,Z_1)+(x_2,y_2)=(X_3,Y_3,Z_3)$ with the following:

$$X_3=\alpha^2-\beta^3-2X_1\beta^2\,,\ \ Y_3=\alpha\left(X_1\beta^2-X_3\right)-Y_1\beta^3\,,\ \ Z_3=Z_1\beta\,, \tag{10}$$

where $\alpha=Z_1^3y_2-Y_1$, $\beta=Z_1^2x_2-X_1$.

A new representative of *P* having the same *z* coordinate as (X_3,Y_3,Z_3) is:

$$\left(X_1^{(1)},Y_1^{(1)},Z_1^{(1)}\right)=\left(X_1\beta^2,Y_1\beta^3,Z_1\beta\right)\equiv(X_1,Y_1,Z_1)\,. \tag{11}$$

Then, by applying the special addition (7) to computation *P* + (*P*+*Q*), we obtain:

$$X_4=\omega^2-\theta^3-2X_1^{(1)}\theta^2\,,\ \ Y_4=\omega\left(X_1^{(1)}\theta^2-X_4\right)-Y_1^{(1)}\theta^3\,,\ \ Z_4=Z_1^{(1)}\theta\,, \tag{12}$$

where $\theta=X_3-X_1^{(1)}$, $\omega=Y_3-Y_1^{(1)}$.

The cost of the previous execution sequence is (8*M* + 3*S*) + (5*M* + 2*S*) = 13*M* + 5*S*. Also note that these formulae require a very low number of additions. If we take these operations into account the total cost is approximately $13M+5S+12A$, which is superior to the traditional execution consisting of a doubling followed by a mixed addition: $12M + 7S + 16A$ if $a=-3$, and $12M + 9S + 15A$ if a is randomly chosen. See [HMV04, Section 3.2.2] for cost details of traditional operations. To simplify comparisons, costs of field multiplication/division by small constants have been considered equivalent to field addition. The reader must note that, if these cheaper operations are relatively expensive in a given implementation, the advantage of the proposed DA is further increased.

If the cost ratio *A*/*M* is very low, then we can reduce costs as follows. The addition in (10) is first transformed using the strategy described in Section 3.1 consisting in trading multiplications for squarings [LM07a]:

$$\begin{aligned} X_3 &= 4\alpha^2 - 4\beta^3 - 8X_1\beta^2 , \\ Y_3 &= 2\alpha\left(4X_1\beta^2 - X_3\right) - 8Y_1\beta^3 , \\ Z_3 &= 2Z_1\beta = \left(Z_1+\beta\right)^2 - Z_1^2 - \beta^2 , \end{aligned} \tag{13}$$

where $\alpha = Z_1^3 y_2 - Y_1$, $\beta = Z_1^2 x_2 - X_1$.

Then, the new representative of *P* having the same *z* coordinate as (X_3, Y_3, Z_3) is:

$$\left(X_1^{(1)}, Y_1^{(1)}, Z_1^{(1)}\right) = \left(4X_1\beta^2, 8Y_1\beta^3, 2Z_1\beta\right) \equiv \left(X_1, Y_1, Z_1\right). \tag{14}$$

And by applying the special addition (7) to computation *P* + (*P*+*Q*), we obtain:

$$X_4 = \omega^2 - \theta^3 - 2X_1^{(1)}\theta^2,\ Y_4 = \omega\left(X_1^{(1)}\theta^2 - X_4\right) - Y_1^{(1)}\theta^3,\ Z_4 = Z_1^{(1)}\theta . \tag{15}$$

where $\theta = X_3 - X_1^{(1)}$, $\omega = Y_3 - Y_1^{(1)}$.

Observe that:

$$\begin{aligned} \theta &= X_3 - X_1^{(1)} = (4\alpha^2 - 4\beta^3 - 8X_1\beta^2) - (4X_1\beta^2) = 4\alpha^2 - 4\beta^3 - 12X_1\beta^2 , \\ \omega &= Y_3 - Y_1^{(1)} = 2\alpha\left(4X_1\beta^2 - X_3\right) - 8Y_1\beta^3 - 8Y_1\beta^3 = 2\alpha\left(4X_1\beta^2 - X_3\right) - 16Y_1\beta^3 , \\ &= 2\alpha\left(X_1^{(1)} - X_3\right) - 16Y_1\beta^3 = -2\alpha\theta - 16Y_1\beta^3 = -(\alpha+\theta)^2 + \alpha^2 + \theta^2 - 16Y_1\beta^3 , \end{aligned}$$

which avoids intermediate computations (X_3, Y_3, Z_3), and fixes the cost of the DA operation at only $11M + 7S$. We remark at this point that this formula achieves better performance than the one described previously (eq. (10), (11) and (12)) whenever field additions are very inexpensive, as the performed transformations introduce a few more of these operations. Also, the reader must observe that this new formula benefits from a small *S*/*M* cost ratio (e.g., implementations where $1S \approx 0.6M$).

Similar results can be obtained for the computation of a general DA by simply replacing the first mixed addition $\mathcal{J}+\mathcal{A}\rightarrow\mathcal{J}$ (13) by a general addition $\mathcal{J}+\mathcal{J}\rightarrow\mathcal{J}$, where $\mathcal{J}$ and $\mathcal{A}$ represent points in Jacobian and affine coordinates, respectively. In this case, the cost is fixed at $14M + 9S$. A further reduction can be achieved if some extra memory is available. By precomputing and storing values Z_2^2 and Z_2^3 (see [EPAF] for complete details), the cost of the general DA can be reduced to only $13M + 8S$.

Similarly, we can apply the described procedure to composite operations derived from the scheme (5). Table 2 contains the costs of the composite operations optimized in such a way (items labeled with (2)), and compares them with the costs of previous formulas (tripling and quintupling) presented in [LM07a,LM07c]. Also, we compare our results with the best previous operation costs in the literature. Since there is no septupling formula in the literature which we could compare with, we estimate its cost by combining doubling, tripling and general addition in the cheapest possible way (a doubling/tripling formula for the general case was presented in [LM07b, Appendix A] with a cost of $9M + 15S$; other operations in the special case are from [LM07a]; see Table 1).

Table 2. Costs of optimized composite operations using Jacobian coordinates in comparison with previous works.

Composite Operation	Our work		
	general	$a = -3$	# reg.
Tripling [DIM05]	(1) $10M + 6S$	-	10
Tripling [LM07a]	$6M + 10S$	$7M + 7S$	8
Tripling [LM08]	(2) $7M + 10S$	$8M + 7S$	6
Tripling (this work)	(2) $6M + 11S$	$7M + 8S$	6
Quintupling [MD07a]	$15M + 10S$	$15M + 8S$	N/A
Quintupling [LM07c, Appendix C]	$10M + 14S$	$11M + 11S$	10
Quintupling [LM08]	(2) $12M + 12S$	$13M + 9S$	6
Quintupling (this work, Appendix A)	(2) $9M + 15S$	$10M + 12S$	N/A
Septupling (traditional)	$20M + 20S$	$21M + 17S$	8
Septupling [LM08]	(2) $17M + 14S$	$18M + 11S$	6
Septupling (this work)	(2) $13M + 18S$	$14M + 15S$	N/A
General DA (traditional)	$16M + 10S$	$16M + 8S$	N/A
General DA (this work)	(3) $14M + 9S$	$14M + 9S$	N/A
General DA with extra storage (this work)	(3) $13M + 8S$	$13M + 8S$	N/A
DA (traditional)	$12M + 9S$	$12M + 7S$	7
DA (this work)	(3) $13M + 5S$	$13M + 5S$	6
DA (this work; also [LM07b])	(3) $11M + 7S$	$11M + 7S$	6

(1) SSCA-protected tripling by [DIM05]; (2) Using scheme (5); (3) Using scheme (6)

As can be seen, composite operations derived using schemes (5) and (6) are significantly faster and, remarkably, more memory-efficient. Further reductions introduced in this section have contributed with this high performance in terms of computing cost.

In terms of speed, the tripling presented in [LM07a] is still the fastest. However, our introduced improvements have made the quintupling, septupling and doubling-addition derived from the schemes (5) and (6) the current speed leaders. Note that the explicit formula for quintupling is given in Appendix A.

We use these faster operations for our comparisons and cost analyses of scalar multiplication methods in Section 8. As it will turn out, composite operations of the form dP will be highly useful for the efficient realization of methods using multiple radices (e.g., *mb*NAF and Frac-*wmb*NAF). In the case of DA and general DA, besides contributing to speed up further multibase NAF methods, these operations will be shown to significantly reduce the cost of the traditional radix-2 methods such as NAF, *w*NAF and Frac-*w*NAF.

3.3 New Point Operation Formulas in mixed Jacobian-affine coordinates

The use of radices other than 2 opens the possibility of improving the computing time of the scalar multiplication by allowing specialized formulas with other radices. In this context, we contribute in this section with a new set of formulas in radices 3 and 5 for the special cases $3\mathcal{A} \rightarrow \mathcal{J}$ and $5\mathcal{A} \rightarrow \mathcal{J}$, respectively. Thus, they represent the computation of the tripling (or quintupling) of a point that is in affine coordinates and yields the result in Jacobian coordinates.

The costs of the new formulas are summarized in Table 3. They appear on top of the corresponding formulas which they were derived from. For details about these formulas, the reader is referred to Appendices B and C. Also, they can be found in our online database of state-of-the-art formulas in Jacobian coordinates [EPAF].

Table 3. Costs of new formulas in mixed Jacobian-affine coordinates.

Operation	Cost
Tripling ($Z_1 = 1$) (this work)	$5M + 7S$
Tripling [LM07a]	$7M + 7S$
Quintupling ($Z_1 = 1$) (this work)	$8M + 12S$
Quintupling (this work; Appendix A)	$10M + 12S$

As can be seen, these formulas in mixed Jacobian-affine coordinates reduce the cost in $2M$ in comparison with their traditional counterparts. They can be used for instance to perform computations with the points in affine coordinates stored in a precomputed table.

We make use of these formulas to reduce further the computing cost of the scalar multiplication using *mb*NAF and Frac-*wmb*NAF methods in Section 8.

4 Edwards Curves

The standard ECC curve (1) has been widely deployed in most implementations using elliptic curves as it has been sponsored by international standardization bodies such as NIST. However, nowadays they do not present the fastest group law. In fact, in [Edw07] Edwards introduced a new normal form for elliptic curves with more efficient group operations. Such a curve has the following form

$$E: \quad x^2 + y^2 = 1 + dx^2y^2 , \tag{16}$$

where $d \notin \{0,1\}$. The addition law is given by

$$(x_1, y_1), (x_2, y_2) \mapsto \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right), \tag{17}$$

Interestingly enough, if d is not a square in the underlying finite field, say $\mathbb{F}_p$, there are no exceptions in the group law. It works for all points, contrarily to what happens with the standard ECC form (1).

In [BL07a], the authors presented explicit formulas for point operations in Edwards curves using (X,Y,Z) coordinates, which are known as Edwards coordinates. They showed that general addition, mixed addition, doubling and tripling can be performed with 10M + 1S, 9M + 1S, 3M + 4S and 9M + 4S, respectively, when d from eq. (16) is fixed at a small value and, hence, multiplications with it can be considered negligible in comparison with a regular field multiplication.

Later, [BL07b] introduced a new coordinate system, known as inverted Edwards coordinates, where (x_i, y_i) is represented by $(Z_i/X_i, Z_i/Y_i)$ instead of $(X_i/Z_i, Y_i/Z_i)$ (used to derive formulas in Edwards coordinates). With these new coordinates, the costs of most operations remain the same, with exception of general and mixed additions that are reduced in 1M to 9M + 1S and 8M + 1S, respectively.

Other useful operations are doubling and addition in mixed (inverted) Edwards-affine coordinates. In the first case, a doubling with the form $2\mathcal{A} \rightarrow \mathcal{E}$, where $\mathcal{A}$ denotes an initial point in affine and $\mathcal{E}$ denotes the resulting point in (inverted) Edwards coord., costs 3M + 3S. An addition with form $\mathcal{A} + \mathcal{A} \rightarrow \mathcal{E}$ costs 6M + 1S and 7M in Edwards and inverted Edwards coordinates, respectively.

In the following section, we contribute with new formulas for the computation of operations using mixed coordinates on these efficient elliptic curve forms.

4.1 New Point Operation Formulas in mixed Edwards-affine coordinates

Similarly to the case of Jacobian coordinates, the use of multibase methods for the scalar multiplication on an Edwards curve allows the improvement of the computing time by means of specialized formulas with radices different than 2. In this section, we contribute to the state-of-the-art with a new set of formulas in radix 3 for the special case $3\mathcal{A} \rightarrow \mathcal{E}$.

The costs of the new formulas are summarized in Table 4. They appear on top of the corresponding formulas which they were derived from. For details about these formulas, the reader is referred to Appendices D and E.

Table 4. Costs of new formulas in Edwards and inverted Edwards coordinates.

Operation	Coordinate system	Cost
Tripling ($Z_1 = 1$) (this work)	Edw.	$6M + 3S$
Tripling [BBL+07]		$9M + 4S$
Tripling ($Z_1 = 1$) (this work)	InvEdw.	$7M + 3S$
Tripling [BBL+07]		$9M + 4S$

As can be seen, these formulas in mixed Edwards (inverted Edwards)-affine coordinates reduce the cost in $3M + 1S$ ($2M + 1S$) in comparison with their traditional counterparts.

These formulas are used to accelerate the execution of the multibase methods for scalar multiplication on Edwards curves in Section 8.

5 Scalar Multiplication Methods

Much work has been invested to the development of efficient methods for the ECC scalar multiplication. In this section, we explore and present a detailed analysis of some very recent efforts, especially of the methods working with double-base number representation (DB). Also, we discuss the advantages of the *mb*NAF and *wmb*NAF methods. For completeness, the popular radix 2-based approaches, namely NAF, *w*NAF and Frac-*w*NAF, are also defined. The reader must note that we do not deal here with methods protected against side-channel attacks. If, for instance, simple side-channel analysis (SSCA) attacks are a concern, then we could apply a technique based on side-channel atomicity [CCJ04] to protect the methods discussed in this section (see [LM08] for an example of the SSCA-protected implementation of *mb*NAF and *wmb*NAF).

In the following, we assume that $\# E(\mathbb{F}_p) = h.q$, where q is prime and $h << q$. Hence, $p \approx q$. If the scalar k of a scalar multiplication kP is randomly chosen in the range $[1, q - 1]$, and $n = \log_2 p \approx \log_2 q$, then the average length of k is $l \approx n - 1$. We refer indistinctly as density or Hamming weight to the number of nonzero elements of a given scalar representation. In the case of scalar multiplication, the latter directly translates to the number of point additions required to compute kP using such representation.

5.1 Non-Adjacent Form (NAF) and Window-*w* Non-Adjacent Form (*w*NAF)

Radix 2 or binary representations have been traditionally used because the expansion translates directly into a given sequence of point doublings ($2P$) and additions ($P+Q$), which are the basic ECC point operations. For instance, the binary method using elements $\{0,1\}$ is known to achieve a density of 1/2. The density of the binary expansion can be effectively reduced with a signed representation that uses elements in the set $\{-1,0,1\}$, taking advantage of the fact that the cost of computing inverses of points (e.g., $-P$) in additive groups is negligible. Among different signed radix 2-based representations, NAF is a canonical representation with the fewest number of nonzero digits for any scalar k. The NAF representation of k, denoted by NAF(k), contains at most *one* nonzero digit among any two successive digits.

The expected number of doublings and additions using NAF is approximately (n – 1) and n/3, respectively. Thus, the cost of the scalar multiplication using this method is approximately $(n-1)\text{D}+(n/3)\text{A}$, where D and A represent the cost of a point doubling and addition, respectively.

If there is memory available, one can make use of precomputations to reduce the computing time for scalar multiplication. In such case, *w*NAF is the natural expansion of NAF. It basically exploits the availability of precomputed values to “insert” windows of width w, which permits the consecutive execution of several doublings to reduce the density of the expansion. The *w*NAF representation of k, denoted by $\text{NAF}_w(k)$, contains at most *one* nonzero digit among any w successive digits.

It can be observed that *w*NAF is simply a generalization of NAF to any window value, and that NAF is the only variant in such a generalization that does not require precomputations P_i (hereafter we refer as precomputed points to non-trivial points not including $\{O, P\}$).

The average density of nonzero digits in *w*NAF for a window of width w is

$$\mathcal{D}=\frac{1}{w+1}, \tag{18}$$

and the number of required precomputed points is $(2^{w-2}-1)$. Thus, the cost of the *w*NAF method is approximately $(n-1)\text{D}+(n/(w+1))\text{A}$.

Fractional Window-*w* NAF (Frac-*w*NAF)

In [Möl02], the author observed that the required table of precomputed points using *w*NAF is of the form $d_i \in D=\{\pm1,\pm3,\pm5,\ldots,\pm(2^{w-1}-1)\}$, where $w>2\in\mathbb{Z}^+$. The latter limits the number of precomputed points to 1, 3, 7, 15 points, and so on. However, a specific implementation could have memory restrictions that are different to these values. Moreover, because methods involving an unknown scalar force to compute the precomputed table every time a scalar multiplication is performed, it can happen that a table with a different number of points achieves the minimal cost. Thus, [Möl02] proposes to recode the binary representation of an integer by using windows of flexible size in such a way that allows the use of a digit set of the form $d_i \in D = \{\pm 1, \pm 3, \pm 5, \ldots, \pm m\}$, where $m \geq 1$ is an odd integer. In this way, one can flexibly choose any number of precomputed points.

The average nonzero density for Frac-*w*NAF is given by [Möl04]

$$\mathcal{D} = \left(\lfloor \log_2 m \rfloor + \frac{(m+1)}{2^{\lfloor \log_2 m \rfloor}} + 1 \right)^{-1} \qquad (19)$$

Note that if $m=1$, Frac-*w*NAF is actually reduced to the NAF method with a nonzero density of about 1/3. Similarly, eq. (19) attains the same values as (18) for the standard window values of *w*NAF. For instance, Frac-*w*NAF with $m=7$ reduces to *w*NAF with $w = 4$ ($\mathcal{D} \approx 0.2$).

We will apply the same concept of fractional windows in radix 2 to the context of multibase methods to derive the new Frac-*wmb*NAF in Section 6.

5.2 Double-Base (DB) Methods

[DIM05] proposed to represent the scalar k using mixed powers of 2 and 3 as follows:

$$k = \sum_{i=1}^{m} k_i 2^{b_i} 3^{c_i} \;, \qquad (20)$$

where m is the length of the expansion, k_i is the sign (i.e., $k_i \in \{-1, 1\}$), and b_i and c_i form decreasing sequences $b_{\max} \geq b_1 \geq b_2 \geq \ldots \geq b_m \geq 0$ and $c_{\max} \geq c_1 \geq c_2 \geq \ldots \geq c_m \geq 0$, respectively.

This Double-Base (DB) representation is highly sparse and, consequently, permits to reduce the Hamming weight of the expansion for the scalar. With the introduction of efficient tripling formulae [DIM05], these representations using ternary bases are thought to greatly reduce the execution time of scalar multiplication.

Nevertheless, finding the expansion (20) of a given integer in a reasonable amount of time and utilization of resources is probably the main obstacle to apply this method in practice. In fact, several authors have speculated about the difficulty of finding short expansions using $\{2^{b_i} 3^{c_i}\}$-terms, and have defined it as a difficult problem on its own. [DIM05] proposed to solve that problem by establishing "efficient" maximum bounds $b_{\max}$ and $c_{\max}$ for the powers of 2 and 3, respectively, and then executing an exhaustive search for closest terms $2^{b_i} 3^{c_i}$ (method referred to as "Greedy" algorithm). This approach is obviously heuristic and introduces several drawbacks into the DB scheme. We summarize some of them in the following.

1. From a theoretical point of view, DB expansions cannot (until today) be defined

adequately. The density or number of nonzero terms expected for an n-bit scalar is, thus, estimated only empirically.

2. To find short expansions, some authors have suggested some heuristics to establish the maximum values for exponents of powers 2 and 3. If the level of security (i.e., the bitlength of the scalar) and/or curve form are modified, then different estimates must be applied. This increases the difficulty to analyze optimal parameters, making the implementers' work harder.

3. Looking for closest $\{2^{b_i}3^{c_i}\}$-terms to build a DB expansion implies having a table storing several powers of 2 or 3 or combinations of these. At first sight this would result very impractical in constrained settings. We can trade memory for speed and store only part of the required table. However, this leads to higher conversion times (to DB representation) and/or very expensive precomputation stages (we will explore some recent developments of this kind later).

4. There is no established "distance" between terms. In fact, it can happen for instance that only triplings are required between two additions (i.e., between two nonzero terms). On the standard EC curve (1) this would increase the computing cost of scalar multiplication as DB would not allow the use of the highly efficient DA operation.

These were some of the problems inherent to this method since its proposal in [DIM05]. Nevertheless, the authors showed that DB leaded to lower costs to compute the scalar multiplication, even surpassing NAF$_4$. However, point arithmetic has been greatly improved since then and their findings do not completely apply today. Also, their comparisons against other methods did not take into account the memory/computing time penalty that the DB method requires for precomputation and/or conversion. The latter is also missing in most of the subsequent works about DB.

Window-based DB methods by [DI06]

Later, [DI06] extended the DB approach with two methods intended for applications that can afford precomputations. The first method, which we will refer to as *DB with anomalies*, allows extending the range of search for $\{2^{b_i}3^{c_i}\}$-terms by relaxing the condition of monotonically decreasing powers of 2 and 3 during conversion to DB. Thus, it makes use of a precomputed table of the form $\{\pm 2, \pm 2^2, \ldots, \pm 2^{w_1}, \pm 3, \pm 3^2, \ldots, \pm 3^{w_2}\}$, where w_1 and w_2 represent the maximum exponents expanding the range of search. Note that the authors restrict the precomputation values to individual powers of 2 and 3. Otherwise, we would be forced to precompute and store values of the form $2^a 3^b$, whose cost grows very rapidly (we will show later that recent approaches use in fact these mixed terms for precomputations, which lead to inefficient implementations).

The second method presented in [DI06], referred to as *Extended DB*, makes use of an extended set $\mathcal{S}$ of odd digits k_i (see (20)) coprime to 3. For instance, $k_i \in \mathcal{S} = \{\pm1, \pm5, \pm7, \pm11, \ldots\}$. We will denote this method by $\mathcal{S}$-DB. Comparing it with the traditional table used by *w*NAF (namely $\{1,\ 3,\ 5,\ \ldots\ ,\ (2^{w-1}-1)\}$), we notice that one disadvantage of the Extended DB method is the particular structure of the precomputed table, which would require more computations to have a given number of precomputed points. This fact will be evident in Section 7 when we explore efficient schemes for precomputation.

Also, observe that these two window-based DB methods inherit the problems found in the original DB method without precomputations, i.e., the drawbacks of the conversion to DB and the lack of a theoretical foundation.

An extension of the previous work (referred to as Triple-Base or TB) was presented in [MD07a] by adding radix 5 to the DB method. It is also based on a “Greedy” algorithm to find a scalar representation, and hence, exhibits the same shortcomings as DB.

Window-based DB method by [MD07b]

Mishra and Dimitrov presented very recently a new variant for the window version of DB. In this case, they actually resort to precomputed values of the form 2^a3^b with the objective of reducing the searching time during conversion to DB. If compared with the original approach of looking for best approximations in the set $\{2^{b_i}3^{c_i} : 0 \le b_i \le b_{\max}, 0 \le c_i \le c_{\max}\}$, where $b_{\max}$ and $c_{\max}$ are the maximum exponents allowed for 2 and 3, respectively, at each searching iteration, this new method restricts the search at each round to the set $\{2^{b_i}3^{c_i} : b_{\max} - w_1 \le b_i \le b_{\max},\ c_{\max} - w_2 \le c_i \le c_{\max}\}$, where w_1 and w_2 are the window parameters for the exponents 2 and 3, respectively. The latter reduces considerably the time to convert a number to DB representation. Note that $b_{\max}$ and $c_{\max}$ in both cases are updated at each iteration following the monotonically decreasing condition $b_{\max} \ge b_1 \ge b_2 \ge \ldots \ge b_m \ge 0$ and $c_{\max} \ge c_1 \ge c_2 \ge \ldots \ge c_m \ge 0$.

Nevertheless, the authors of [MD07b] also observed the problem that involves the expensive computation of precomputed values of the form 2^a3^b. The latter does not only grows very rapidly for small values w_1 and w_2, but also presents a very sparse structure that increases considerably the cost of the precomputation stage for a given number of points. It is expected then that the possible savings obtained during the scalar multiplication (and the conversion stage) are overwhelmingly downplayed by the excessive cost of the precomputation.

To try to solve this problem, [MD07b] uses a modified scheme in which only part of the required precomputed points are computed in advance. The remainder points are computed on-the-fly when required by the scalar multiplication. However, we note that this approach introduces a new problem into the method. Since the computation of some points required during the scalar multiplication has to be completed on top of the few points precomputed, these precomputed points must be stored in Jacobian coordinates forcing the use of general additions instead of the very efficient mixed Jacobian-affine addition. Comparisons in Section 8 will show that this fact increases the overall costs.

Besides the problems highlighted, the execution sequence of the point operations in the scalar multiplication as proposed by [MD07b] does not allow the use of the very efficient DA operation discussed in Section 3.2. Moreover, although speeding up and reducing the memory requirements of the conversion step, it still does requires extra memory in comparison to other methods, and its heuristic nature (in the selection of maximum bounds for exponents 2 and 3) does not allow for a theoretical analysis of performance.

Window-based DB method by [BPP07]

An alternative window-based method using double-base number representations was recently presented in [BPP07]. From a theoretical viewpoint, the advantage of this method is that allows for an estimation of the expected number of point operations by fixing the size of the windows for exponents of 2 and 3. This permits to establish the number of expected terms in the DB representation, and also to minimize the search space for $\{2^{b_i}3^{c_i}\}$-terms, which reduces further the conversion times to DB and its memory requirements.

However, through a more detailed analysis, it can be seen that (similarly to the method by [MD07b]) the problem of finding closest $\{2^{b_i}3^{c_i}\}$-terms by means of the "Greedy" algorithm has been shifted to the problem of having a very expensive precomputation stage which, at the end, although reduces the memory requirements (the table used to look for DB terms is significantly smaller), increases the overall cost of the scalar multiplication.

In Section 8, we will compare the performance of the DB methods discussed in this section with the new multibase NAF methods introduced in [Lon07] which, as it will be evident in the following section, avoid the drawbacks of DB.

5.3 Multibase Non-Adjacent Form (*mb*NAF) and window-*w* NAF (*wmb*NAF) methods

Longa and Miri recently solved the problem of finding short and generic multibase representations in a very efficient manner [Lon07,LM07c].

They proposed the use of the following representation for the scalar k:

$$k = \sum_{i=1}^{m} k_i \prod_{j=1}^{J} a_j^{c_i(j)}, \tag{21}$$

where: bases $a_1 \neq a_2 \neq \ldots \neq a_J$ are positive prime integers (a_1: main base).
m is the length of the expansion.

k_i are signed digits from a given set D.
$c_i(j)$ are monotonically decreasing exponents, s.t. $c_1(j) \geq c_2(j) \geq \ldots \geq c_m(j) \geq 0$ for each j from 2 to J, and
$c_i(1)$ are monotonically decreasing exponents for the main base a_1 (i.e., $j = 1$), s.t. $c_1(1) > c_2(1) > \ldots > c_m(1) > 0$.

Note that this multibase representation is quite generic, and does not have restrictions in the number or order of bases (contrarily to what happens in the DB representation). In fact, the only restriction is given by the last two conditions of monotonically decreasing exponents given above, which guarantee that an expansion of the form (21) is efficiently executed by a scalar multiplication scanning the digits from left to right.

The key strategy applied to this multibase representation (21) is the use of the property of non-adjacency given by the last condition above. By fixing the minimal number of consecutive zero terms of the base a_1 (called the main base) to only 2 in (21), we achieve the *mb*NAF representation, which minimizes the number of required precomputed points.

On the other hand, if we relax the previous condition and allow larger window sizes (i.e., allowing 3, 4, 5, or more, consecutive zero terms of the main base before the following nonzero term is expected) we can reduce further the average number of nonzero terms in the representation of the scalar at the expense of a larger precomputed table. The previous technique is known as *wmb*NAF.

Algorithm 5.1 performs the conversion of any positive integer to *mb*NAF (and *wmb*NAF) representation following the description above. Note that we have integrated here the algorithms for *mb*NAF and *wmb*NAF given in [Lon07] and [LM08].

Algorithm 5.1 Computing the *mb*NAF (*wmb*NAF) of a positive integer

INPUT: scalar k, bases $\mathcal{A} = \{a_1, a_2, \ldots, a_J\}$, where $a_j \in \mathbb{Z}^+$ are primes for $1 \leq j \leq J$, window $w = 2$ for *mb*NAF, and window $w > 2$ for *wmb*NAF, where $w \in \mathbb{Z}^+$
OUTPUT: the $(a_1, a_2, ..., a_J)\,\text{NAF}_w(k) = (\ldots, k_2^{(a_2)}, k_1^{(a_1)})$

```
1. $i = 0$
2. While $k > 0$ do
   2.1. If $k \bmod a_1 = 0$ or $k \bmod a_2 = 0$ or … or $k \bmod a_j = 0$, then $k_i = 0$
   2.2. Else:
        2.2.1 $k_i = k \text{ mods } a_1^w$
        2.2.2 $k = k - k_i$
   2.3
        2.3.1 If $k \bmod a_1 = 0$, then $k = k/a_1$, $k_i = k_i^{(a_1)}$
        2.3.2 elseif $k \bmod a_2 = 0$, then $k = k/a_2$, $k_i = k_i^{(a_2)}$
```

```
            ⋮
            2.3.$J$ elseif $k \bmod a_J = 0$, then $k = k/a_J$, $k_i = k_i^{(a_J)}$
        2.4 $i = i + 1$
    3. Return $(\ldots, k_2^{(a_2)}, k_1^{(a_1)})$
```

The function *mods* in Algorithm 5.1 represents the next computation:

$$\begin{cases} \text{If } k \bmod a_1^w \geq a_1^w/2, \text{ then:} \\ \qquad k_i = \left(k \bmod a_1^w\right) - a_1^w \\ \text{Else,} \\ \qquad k_i = k \bmod a_1^w \end{cases} \tag{22}$$

Note that Algorithm 5.1 approximates every computation to the closest number divisible by a power of the main base, namely a_1. Thus, two or more consecutive operations by a_1 are guaranteed before the next addition. In contrast to traditional radix-2 methods, here the nonzero density of the expansion is further reduced as the number of bases is increased since the algorithm looks for extra divisions by the additional bases.

As shown in [LM08], the *mb*NAF and *wmb*NAF methods require a precomputed table of the form

$$d_i \in D = \left\{0, \pm 1, \pm 2, \ldots, \pm \left\lfloor \frac{a_1^w - 1}{2} \right\rfloor \right\} \setminus \left\{ \pm 1 a_1, \pm 2 a_1, \ldots, \pm \left\lfloor \frac{a_1^{w-1} - 1}{2} \right\rfloor a_1 \right\}, \tag{23}$$

where $0 \leq i \leq m$ and $w \geq 2 \in \mathbb{Z}^+$ ($w = 2$ for *mb*NAF). The precomputed table (23) involves

$$\frac{a_1^w - a_1^{w-1} - 2}{2} \tag{24}$$

precomputed points without considering $\{0, \pm 1\}$. Further, preliminary analysis shows that the expected average nonzero density is asymptomatically

$$\mathcal{D} = \frac{a_1 - 1}{w(a_1 - 1) + 1} \tag{25}$$

with regard to the main base. Note that if $\mathcal{A} = \{a_1\}$ (single-base case) with $a_1 = 2$, the multibase representation reduces to the traditional radix-2 methods. Specifically, if $w = 2$

*mb*NAF reduces to NAF (with the same nonzero density around 1/3), and if $w > 2$ *wmb*NAF converts into *w*NAF (eq. (25) takes the form of eq. (18)).

Note that it is expected that having $a_1 = 2$ yields the most efficient scalar multiplication in terms of speed on standard EC curves and Edwards curves, where point doubling is highly efficient in comparison with other operations. In fact, results in Section 8 will show that cases using the set of bases $\mathcal{A} = \{2,3\}$ or $\mathcal{A} = \{2,3,5\}$ yields the highest performance on the aforementioned curve forms. However, in other settings where triplings (or other composite operations) are more efficient, the expectation is that the case with $a_1 \neq 2$ provides a better performance.

We will illustrate what has been said to this point with the following example. Let k = 618 and bases a_1 = 2, a_2 = 3. Using *mb*NAF, k would be computed (following the form (21)) as

$$618 = 2^6 \times 3^2 + 2^4 \times 3 - 2 \times 3 = 2 \times 3 (2^3 (2^2 \times 3 + 1) - 1), \tag{26}$$

where the inner parentheses are executed first during a scalar multiplication execution.

Note that at least two doublings are executed before the following addition. This restriction adjusts to the description given above regarding the minimal number of zero terms associated to the main base that should happen before any addition. This can be more easily visualized if we present (26) as would be given by Algorithm 5.1:

$$(2,3)\mathrm{NAF}_2(618) = 1^{(2)}\, 0^{(3)}\, 0^{(2)}\, 1^{(2)}\, 0^{(2)}\, 0^{(2)}\, -1^{(2)}\, 0^{(3)}\, 0^{(2)}, \tag{27}$$

where the superscript (a_j) represents the base associated with a given digit. Note that in a scalar multiplication the execution is from left to right, performing a doubling if the digit is $0^{(2)}$, a tripling if it is $0^{(3)}$, and a doubling followed by an addition if it is $\pm 1^{(2)}$.

According to (27) a scalar multiplication only costs 6D+2T+2A, where D, T and A represent doubling, tripling and addition, respectively. Then, if we use operations costs from Table 1 (special case $a = -3$; $1S = 0.8M$), the scalar multiplication $[618]P$ would cost approximately $87.6M$. Further, every doubling followed by an addition can be replaced by a cheaper DA operation. In that case the cost can be reduced to only 4D+2T+2DA = $86.4M$.

Let us now compare the multibase NAF representation with the DB method [DIM05]. Let $b_{\max}$ = 9 and $c_{\max}$ = 5 be the maximum bounds for exponents of bases 2 and 3, respectively. By searching for the closest $\{2^{b_i}3^{c_i}\}$-terms in a table containing the power values, we find that $618 = 2^3 \times 3^4 - 3^3 - 3 = 3(3^2(2^3 \times 3 - 1) - 1)$, which can be represented as a digit string by

$$\mathrm{DB}(618) = 1^{(2)}\, 0^{(3)}\, 0^{(2)}\, 0^{(2)}\, -1^{(2)}\, 0^{(3)}\, -1^{(3)}\, 0^{(3)}. \tag{28}$$

Using operations of Table 1, (28) costs 3D+4T+2A = $91.8M$ (or 2D+4T+1A+1DA = $91.2M$), which is higher than $86.4M$ obtained with the *mb*NAF method.

The main observation from the previous example is that, besides requiring extra memory to store the powers of the bases, the DB method does not necessarily require a minimum number of doublings before each addition. That is one of the reasons why the DB method is quite difficult to define theoretically. It searches for closest terms independently on whether such a term has a well defined "distance" with the precedent one. Consequently, the DB method reduces the number of terms in the scalar representation but insert other operations "randomly". If, for instance, too many triplings relative to doublings are inserted, then the overall cost is increased because tripling is a more expensive operation. Furthermore, DB does not guarantee that a doubling will happen before every addition. Thus, the efficient DA cannot be used in all the cases (as can be seen in the example above). The multibase NAF methods avoid such drawbacks because these allow flexibly "defining" the number of operations of each kind that can appear in the representation of *k*.

To summarize, it is easy to see that both *mb*NAF and *wmb*NAF are natural extensions of NAF and *w*NAF. In that sense, they inherit such valuable properties as non-adjacency, simplicity and efficiency in the conversion of any integer to such representations, and no memory penalty.

In the following section, we generalize these methods using a new multibase recoding that allows any number of points in the precomputed table.

6 Fractional Window-*w* Multibase Non-Adjacent Form (Frac-*wmb*NAF) method

In the previous section, we described how the *(w)mb*NAF method approximates a given integer to the value that maximizes the number of divisions by the main base a_1 in subsequent operations. Such approximation is restricted by the chosen window, whose size is fixed in advance, and which ultimately determines the required number of precomputed points.

Let us illustrate the latter with the integer k = 1105, window w = 4 and bases $\mathcal{A} = \{2,3\}$. From (23), we know that we should have available the precomputed points $d_i P$, where $d_i \in D = \{\pm1,\pm3,\pm5,\pm7\}$. By applying Algorithm 5.1 to get the *wmb*NAF representation of *k*, we obtain the following

$$(2,3)\text{NAF}_4(1105) = 1^{(2)}\,0^{(2)}\,0^{(2)}\,0^{(2)}\,7^{(2)}\,0^{(3)}\,0^{(2)}\,0^{(2)}\,0^{(2)}\,1^{(2)},$$

where again the superscript (a_i) represents the base (i.e., 2 or 3) associated with a digit from the set $\{0,\pm1,\pm3,\pm5,\pm7\}$. Note that digits 0 correspond to doubling or tripling operations (depending on the associated base), and that the remainder digits corresponds to additions with one of the precomputed points $d_i P$. The iterative conversion process can be visualized more easily as follows

$$1105-1 \to \frac{1104}{2} \to \frac{552}{2} \to \frac{276}{2} \to \frac{138}{2} \to \frac{69}{3} \to 23-7 \to \frac{16}{2} \to \frac{8}{2} \to \frac{4}{2} \to \frac{2}{2} \to 1 .$$

We can see in the example that, by shifting the initial odd value to 1104, we guarantee at least *four* subsequent divisions by 2 (or *four* doublings in terms of computations for the scalar multiplication) because $1104 \equiv 0 \,(\mathrm{mod}\, 2^4)$. In other words, we establish a "window" from 1104 to 1120, and approximate our integer to the closest extreme, i.e., 1104, using one of the digits of the precomputed table *D*.

The drawback of the previous approach is that windows have a fixed size and we are forced to precompute 1 point if w = 3, 3 if w = 4, 7 if w = 5, and so on. There is no flexibility to choose tables of 2, 4, 5, 6, 8 points, and so on. As mentioned previously, it can happen that tables with a number of points different to the number provided by fixed windows w achieve the highest performance.

To solve the previous problem, we propose a new multibase recoding scheme that uses "fractional" windows. Instead of fixing the windows to a given value, we flexibly resize the window size in such a way that only a given number of precomputed points is required.

For the remainder, we will assume that the main base a_1 is 2 as this value is expected to achieve the lowest costs with most efficient EC curve forms. A straightforward variation of the following procedure would allow the use of other bases as the main base.

First, let us establish our ideal table with unrestricted number of precomputed points d_iP

$$d_i \in D = \{\pm 1, \pm 3, \pm 5, \ldots, \pm m\} \tag{29}$$

where $m \geq 3 \in \mathbb{Z}^+$ is an odd integer. If we define m in terms of the standard windows w from wNAF and wmbNAF, it would be expressed as

$$m = 2^{w-2} + s \tag{30}$$

where $2^{w-2} < m < 2^{w-1}$ and $s \geq 1 \in \mathbb{Z}^+$ is odd. Notice that expressions are using the absolute values of digits d_i.

We can now define the rules of our recoding scheme for bases $\mathcal{A} = \{a_1, a_2, \ldots, a_J\}$ in the following:

1. If $k \bmod 2 = 0$ or $k \bmod a_2 = 0$ or … or $k \bmod a_J = 0$ then $k_i = 0$,
2. Elseif $0 < r \leq m$ then $k_i = r$,
3. Elseif $m < r < (3m - 4s)$ then $k_i = r - 2^{w-1}$,
4. Elseif $(3m - 4s) \leq r < 2^w$ then $k_i = r - 2^w$,

where $r = k \bmod 2^w$.

Basically, the proposed recoding first detects if k is divisible by one of the bases from $\mathcal{A}$. If not, it establishes a window w and checks if k can be approximated to the closest extreme of the window using any of the digits d_i available. It can be easily verified that the latter will be accomplished if conditions 2 or 4 are satisfied. Thus, k_i gets the corresponding value d_i. Otherwise, the established window is too large and, hence, it is "reduced" to the immediately preceding window size to which k can be approximated (condition 3).

The described procedure is presented in detail in Algorithm 6.1, which converts any positive integer to its Frac-*wmb*NAF representation. Note that t represents the number of precomputed points (without including $\{0,\pm 1\}$) that would be required to perform a scalar multiplication.

We remark that the proposed algorithm is a generalization of the original *mb*NAF and *wmb*NAF representations (see Section 5.3). In fact, if we fix $w \geq 2 \in \mathbb{Z}^+$ with the traditional window values where the last term m in the precomputed table $\in \{1,3,7,15,\ldots,(2^{w-1}-1)\}$, we obtain the mentioned representations. Observe that we have included the extra conditions $m = 1$, $s = 0$, so that Algorithm 6.1 is also able of converting any integer to its *mb*NAF form (for the particular case $a_1 = 2$).

Algorithm 6.1 Computing the *Frac-wmb*NAF of a positive integer

INPUT: scalar k, bases $\mathcal{A} = \{a_1,a_2,\ldots,a_J\}$, where $a_1 = 2$ and $a_j \in \mathbb{Z}^+$ are primes for $1 \leq j \leq J$, table $D = \{\pm 1, \pm 3, \ldots, \pm(m = 2t+1)\}$, $w \geq 2 \in \mathbb{Z}^+$; $m = 2^{w-2} + s$ and $2^{w-2} < m < 2^{w-1}$, where $m \geq 3$ and $s \geq 1$ are odd integers ($m = 1$, $s = 0$ for *mb*NAF)

OUTPUT: the $(2, a_2, \ldots, a_J)\,\mathrm{NAF}_{w,t}(k) = (\ldots, k_2^{(a_2)}, k_1^{(a_1)})$

```
1. $i = 0$
2. While $k > 0$ do
    2.1. If $k \bmod 2 = 0$ or $k \bmod a_2 = 0$ or … or $k \bmod a_J = 0$, then $k_i = 0$
    2.2. Else:
        2.2.1  $r = k \bmod 2^w$
        2.2.2  If $0 < r \leq m$, then $k_i = r$
        2.2.3  Elseif $m < r < (3m - 4s)$, then $k_i = r - 2^{w-1}$
        2.2.4  Elseif $(3m - 4s) \leq r < 2^w$, then $k_i = r - 2^w$
        2.2.5  $k = k - k_i$
    2.3
        2.3.1  If $k \bmod 2 = 0$, then $k = k/2$, $k_i = k_i^{(2)}$
        2.3.2  Elseif $k \bmod a_2 = 0$, then $k = k/a_2$, $k_i = k_i^{(a_2)}$
        ⋮
        2.3.J  Elseif $k \bmod a_J = 0$, then $k = k/a_J$, $k_i = k_i^{(a_J)}$
    2.4 $i = i + 1$
3. Return $(\ldots, k_2^{(a_2)}, k_1^{(a_1)})$
```

Let us illustrate the new method with an example. If k = 1105 (following the example above) and we select m = 5 to have a table of the form $D = \{\pm 1, \pm 3, \pm 5\}$, then w = 4 and s = 1 by means of (30). Then, using the conversion Algorithm 6.1 we obtain

$$(2,3)\text{NAF}_{4,2}(1105) \;=\; 1^{(2)}\,0^{(3)}\,0^{(2)}\,0^{(2)}\,-1^{(2)}\,0^{(3)}\,0^{(2)}\,0^{(2)}\,0^{(2)}\,1^{(2)}\,,$$

and the conversion process can be visualized as follows

$$1105-1=\frac{1104}{2}=\frac{552}{2}=\frac{276}{2}=\frac{138}{2}=\frac{69}{3}=23+1=\frac{24}{2}=\frac{12}{2}=\frac{6}{2}=\frac{3}{3}=1\,.$$

We can see that, this time, when 23 is obtained in the middle of the conversion, the algorithm finds that it requires an addition with −7 to approximate the value to 16 (which is the closest number $\equiv 0 \pmod{2^4}$, as required for a window w = 4). However, the digit −7 is not part of our table D anymore, so the window size is reduced accordingly to w = 3 to establish a new window between 16 and 24. Then, 23 is rightly approximated to the closest value (i.e., 24) using an addition with 1.

As can be seen, our method resizes the window in such a way that allows the use of a flexible number of digits. Furthermore, in contrast to the traditional radix 2-based Frac-*w*NAF [Möl02,Möl04], our algorithm looks for additional divisions by an extended set of bases so that the number of divisions (or zero terms in the representation) is efficiently extended. This is expected to reduce the nonzero density of the scalar representation, and consequently, reduce the total cost of the scalar multiplication.

Again, it is important to remark that this method falls in the same category as *mb*NAF and *wmb*NAF, inheriting the property of non-adjacency using multiple bases. And also, it has a simple and efficient conversion process with no memory penalty.

In Section 8, we provide extensive tests proving the high performance of this multibase method. In the following, we present efficient schemes for the precomputation of points as required for methods as the one introduced in this section.

7 Precomputation Schemes

Precomputed points are a useful tool exploited to implement window-based methods for the scalar multiplication. However, when the point to be multiplied is variable, there is no other option that calculating such precomputed points each time a scalar multiplication is executed. Hence, computing the precomputed table is a critical task. In fact, its efficiency determines not only the maximum number of points that yields the optimal performance but also determines whether the scalar multiplication method is actually efficient or not.

In the following, we summarize the precomputation schemes that, to our knowledge, offer the lowest cost in each scenario. Note that some of them are introduced in this work.

7.1 Standard ECC Curve in Jacobian Coordinates

In [LM07b], Longa and Miri introduced a new highly efficient scheme for the computation of the precomputed table of the form $d_i \in D = \{\pm 1, \pm 3, \pm 5, \ldots, \pm m\}$, for $m \geq 1 \in \mathbb{Z}^+$ odd, using Jacobian coordinates. The scheme was based on the following execution sequence combined with the special addition (7)

$$d_iP = 2P + \ldots + 2P + 2P + P \,. \tag{31}$$

Two versions with slightly different memory requirements were derived from this approach. They basically differ in the number of values that are stored from the additions in (31) and reused later during conversion to affine representation. In this work, we will use the fastest scheme (referred to as Scheme $\mathcal{J}$), which requires some extra memory for high window values but achieves the lowest computing cost known in the literature for the standard curve (1). That cost is given by

$$\text{Cost}_{\text{Scheme}\,\mathcal{J}} = 1I + (9L)\,M + (2L+6)\,S \,, \tag{32}$$

where L represents the number of points in the precomputed table without including $\{\pm 1\}$.

In this paper, we introduce a modification to the previous scheme to include the precomputed point $2P$. In (31) it is easy to see that $2P$ is calculated as part of the process, so we can add this point to the table to have an additional precomputed point (i.e., $d_i \in D = \{\pm 1, \pm 2, \pm 3, \pm 5, \ldots, \pm m\}$) at the cost of some extra fields operations required to convert it to affine coordinates in the last stage of the scheme. Since the latter requires $4M$, the new cost of the modified scheme (referred to as Extended Scheme $\mathcal{J}$) is given by

$$\text{Cost}_{\text{Ext. Scheme}\,\mathcal{J}} = 1I + (9L-5)\,M + (2L+4)\,S \,, \tag{33}$$

Note that the number of points has been increased by *one*. Hence, in this case $L = (m+1)/2$.

As can be seen, the cost given by (33) involves one field inversion, which is required when converting points to affine coordinates during the last stage of the scheme. However, we note here that an alternative approach would be to avoid that conversion of coordinates. In fact, for implementations where field inversions are very expensive, it could be advantageous to leave points in Jacobian coordinates once computed as in (31). In such case, the cost of the precomputation is only

$$\text{Cost}_{\text{Scheme } \mathcal{J}_2} = (5L-4)\,M + (2L+3)\,S\ . \tag{34}$$

Observe that the cost of (31) without conversion to affine involves one doubling $2\mathcal{A} \to \mathcal{J}$ $(1M + 5S)$ and $(L-1)$ special additions $(5M + 2S)$.

Also note that, when using this approach (referred to as Scheme $\mathcal{J}_2$) with cost (34), general additions must be used in place of mixed additions since precomputed points are left in Jacobian coordinates. Although this increases the cost of the scalar multiplication, it is expected that the savings introduced by (34) during the precomputation stage are advantageous in the overall cost when using large windows w and the implementation has a high I/M ratio.

The efficient schemes discussed in this section (namely, Extended Scheme $\mathcal{J}$ and Scheme $\mathcal{J}_2$) will be used to estimate the cost of precomputing points for wNAF and Frac-*wmb*NAF methods in Section 8.

In the following, we adapt the schemes presented here and propose improved precomputation schemes in Jacobian coordinates for the Extended DB method.

Improving the Precomputation in the Extended DB method by [DI06]

Precomputations in the Extended DB method have the form $d_i \in D = \{\pm1, \pm5, \pm7, \pm11, \pm13, \pm17, \ldots, \pm m\}$. If we use (31) with the special addition with identical z-coordinate (7), the cost of the precomputation can be fixed at

$$1\text{mD} + \left(\frac{m-1}{2}\right)\text{A}^* + 1I + 3(L-1)M + (3M+1S)L\,, \tag{35}$$

where mD and A^* represent the costs of a doubling of the form $2\mathcal{A} \to \mathcal{J}$ and a special addition, respectively. L and m denote the number of points and the last point in the table, respectively.

The validity of (35) is easy to verify. First, the table requires to compute $2P$. Since P is in affine representation and the result should be in Jacobian coordinates, we require a doubling of the form $2\mathcal{A} \to \mathcal{J}$. To complete the computation with the sequence $1P \to 3P \to 5P \to \ldots \to mP$, it is obvious that we require $(m-1)/2$ additions. If we follow [LM07b], these additions can be in fact special additions (7). Then, to convert to affine representation (we need points in affine coordinates to use mixed additions during the scalar multiplication) we can use the Montgomery' simultaneous inversion method. The latter requires one inversion, $3(L-1)$ multiplications to calculate the denominator Z_i^{-1} for the points, and $(3M+1S)$ per point to compute the coordinates $x_i = X_i/Z_i^2$, $y_i = Y_i/Z_i^3$.

By taking into account point operation costs, (35) can be expressed by

$$1I + \left(\frac{m-1}{2}\right)(5M + 2S) + 2(3L-1)M + (1L+5)S . \tag{36}$$

We remark that our scheme can be easily modified to work with an extended table of the form $d_i \in D = \{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7, \ldots, m\}$, as we did in Section 7.1.

7.2 Edwards Curves in Edwards and inverted Edwards Coordinates

For our analysis in Section 8, we will use *w*NAF and Frac-*wmb*NAF to compute the scalar multiplication in Edwards curves. In this case, it is also recommended to have a precomputed table of the form $d_i \in D = \{\pm 1, \pm 2, \pm 3, \pm 5, \ldots, \pm m\}$, where $m \geq 3 \in \mathbb{Z}^+$ odd.

Following the same pattern as (31), computing precomputations would cost

$$1\text{mD} + 1\text{mA} + (L-2)\text{gA} + 1I + 3(L-1)M + (2L)M , \tag{37}$$

where mD, mA and gA represent the costs of a doubling of the form $2\mathcal{A} \to \mathcal{E}$ (Edwards coordinates), a mixed addition $\mathcal{E} + \mathcal{A} \to \mathcal{E}$ and a general addition $\mathcal{E} + \mathcal{E} \to \mathcal{E}$, respectively.

Note that the cost (37) involves the computation of $2P$ with the form $2\mathcal{A} \to \mathcal{E}$. The first addition to obtain $3P$ uses the mixed form $\mathcal{E} + \mathcal{A} \to \mathcal{E}$. Then, the sequence $3P \to 5P \to \ldots \to mP$ requires $(m-3)/2 = L-2$ general additions with form $\mathcal{E} + \mathcal{E} \to \mathcal{E}$. Finally, to convert to affine representation, we can use the Montgomery' simultaneous inversion method, which requires one inversion, $3(L-1)M$ to calculate the denominator Z_i^{-1} for the points, and $2M$ per point to compute the coordinates $x_i = X_i/Z_i$, $y_i = Y_i/Z_i$.

By taking into account point operation costs (see Section 4), (37) can be expressed by

$$\text{Cost}_{\text{Scheme } \mathcal{E}_1} = 1I + (15L-11)M + (L+2)S , \tag{38}$$

in Edwards coordinates.

In the case of inverted Edwards coordinates the conversion to affine at the end of the precomputation process is slightly different since the transformation involves the computations $x_i = Z_i / X_i$ and $y_i = Z_i / Y_i$ (see Section 4). It can be verified that this step requires $1I + (4L)M + \lceil (L-2)/L \rceil M$ using the simultaneous inversion method.

Thus, the total cost of this scheme in inverted Edwards coordinates is

$$1\text{mD} + 1\text{mA} + (L-2)\text{gA} + 1I + (4L)M + \lceil (L-2)/L \rceil M , \tag{39}$$

If we take cost operations into account, (39) can be expressed by

$$\text{Cost}_{\text{Scheme } \mathcal{IE}_1} = 1I + (\lceil (L-2)/L \rceil + (13L-7)M + (L+2)S\,, \tag{40}$$

in inverted Edwards coordinates.

As previously discussed, inversion can be very expensive in some implementations. For that case, we propose an alternative method without inversions. To achieve the latter, conversion to affine representation should not be included and precomputed points left in Edwards coordinates (X,Y,Z) (or inverse Edwards coord.). In this case, the cost is only $1\text{mD} + 1\text{mA} + (L-2)\text{gA}$. When expressed with operation costs given in Section 4, the costs are

$$\text{Cost}_{\text{Scheme } \mathcal{E}_2} = (10L-8)M + (L+2)S\,, \tag{41}$$

in Edwards coordinates, and

$$\text{Cost}_{\text{Scheme } \mathcal{IE}_2} = (9L-7)M + (L+2)S\,, \tag{42}$$

in inverted Edwards coordinates.

The proposed schemes for Edwards coordinates (Schemes $\mathcal{E}_1$ and $\mathcal{E}_2$) and inverse Edwards coordinates (Schemes $\mathcal{IE}_1$ and $\mathcal{IE}_2$) will be used in the following section for the estimation of costs of the scalar multiplication.

8 Implementation Results

We have carried out extensive tests to determine the performance of the most efficient scalar multiplication methods discussed in this work when applied on both standard (1) and Edwards (16) curves. To have a fair comparison we have used for all methods (whenever possible) the state-of-the-art formulas for the point arithmetic. The costs for Jacobian coordinates, standard ECC curves, were detailed in Section 3. In Section 4, we did the same for Edwards curves for the two known representations: Edwards and inverted Edwards coordinates. Table 5 summarizes these costs. Note that we will consider for our analysis $1S = 0.8M$, so the special case $a = -3$ is assumed for doubling, tripling and quintupling in Jacobian coordinates as this gives the lowest costs for this case. Also, we disregard the cost of cheaper operations such as field additions and multiplications by small constants to simplify comparisons.

We implemented NAF, Frac-*w*NAF, *mb*NAF and Frac-*wmb*NAF and ran the algorithms with 1000 160-bit scalars chosen randomly. We first counted the required number of point operations, averaged the results and then estimated the cost for each method using data from Table 5. Also, for window-based methods we included in the overall cost the cost of calculating the precomputed points. For each case we used the best precomputation scheme as detailed in Section 7. Thus, in the case of Jacobian coordinates we used the Extended Scheme $\mathcal{J}$ (labeled as case 1 in Table 6), whose cost is given in (33), and the Scheme $\mathcal{J}_2$ (labeled as case 2) with cost (34). In the case of Edwards coordinates we also considered two schemes (Section 7.2): using one field inversion (cost given by (38); case 1) and with no inversions (cost given by (41); case 2). Finally, for inverted Edwards coordinates we considered the Schemes $\mathcal{IE}_1$ and $\mathcal{IE}_2$ (using one and nil field inversions, respect.), whose costs are given by (40) and (42), respectively.

Table 5. Costs of point operations in Jacobian, Edwards and inverted Edwards coordinates.

Operation	Cost
Doubling [LM07a]	$3M + 5S$
Doubling $(Z_1 = 1)$ [LM07b]	$1M + 5S$
Tripling [LM07a]	$7M + 7S$
Tripling $(Z_1 = 1)$ (this work, Appendix B)	$5M + 7S$
Quintupling (this work, Appendix A)	$10M + 12S$
Quintupling $(Z_1 = 1)$ (this work, Appendix C)	$8M + 12S$
General addition [LM07a]	$11M + 5S$
Mixed addition [LM07a]	$7M + 4S$
Special addition $(Z_1 = Z_2)$ [Mel06]	$5M + 2S$
Doubling-addition (this work; also [LM07b])	$11M + 7S$
General doubling-addition (this work)	$14M + 9S$
Doubling [BL07a]	$3M + 4S$
Doubling $(Z_1 = 1)$ [BL07c]	$3M + 3S$
Tripling [BL07a]	$9M + 4S$
Tripling $(Z_1 = 1)$ (this work, Appendix D)	$6M + 3S$
General addition [BL07a]	$10M + 1S$
Mixed addition [BL07a]	$9M + 1S$
Doubling [BL07b]	$3M + 4S$
Doubling $(Z_1 = 1)$ [BL07c]	$3M + 3S$
Tripling [BL07b]	$9M + 4S$
Tripling $(Z_1 = 1)$ (this work, Appendix E)	$7M + 3S$
General addition [BL07b]	$9M + 1S$
Mixed addition [BL07b]	$8M + 1S$

The results for 160 bits are shown in Table 6. They are divided by number of precomputed points, curve form and scalar multiplication method.

As we can see, the *mb*NAF method using bases (2,3) and (2,3,5) represents a significant improvement in comparison with NAF. The latter holds for the standard and Edwards curves. For instance, *mb*NAF using bases (2,3,5) is about 8.1% faster than our highly optimized NAF using Jacobian coordinates. This highlights the relevance of the multibase method for implementation on constrained devices where storing precomputed points is not possible or too expensive.

In the case of window-based methods, it can be seen that Frac-*wmb*NAF allows a flexible number of precomputed points. In Table 6, we show the performance of this new method against *w*NAF and Frac-*w*NAF when 2, 4, 6 and 7 points are precomputed (not including $\{\pm 1\}$). In all the cases, for both standard and Edwards curves we observe that Frac-*wmb*NAF surpasses the performance of previous methods. However, in this case the differences between our method and *w*NAF (or Frac-*w*NAF) reduce as the window grows. In particular, Frac-*wmb*NAF achieves the highest performance using bases (2,3,5) in the case of standard curves, and bases (2,3) in the case of Edwards curves. Also, it is important to note that for all the cases the highest speed up is achieved when using the digit set $\{\pm 1, \pm 2, \pm 3, \ldots, \pm 13\}$ (the lowest costs per curve are highlighted in bold), which corresponds to a "fractional" window and, thus, highlights the importance of the new recoding method introduced in Section 6.

For the record, in Table 7 we present a comparison of the lowest costs found in the literature for the case of Jacobian coordinates. We can see that Frac-*wmb*NAF is the speed leader for the cases when precomputations include one or nil field inversions, surpassing the optimized implementations of DB and Frac-*w*NAF due to [BBL$^+$07] and [BL07c]. Moreover, our highly optimized implementations of *w*NAF and Frac-*w*NAF are also superior in performance. In this case, the high efficiency of our implementations is due to improved precomputation schemes and more efficient point formulas.

The reader should note that *mb*NAF is also superior to the best numbers by [BBL$^+$07] and [BL07c] even though our method does not require precomputations.

In Table 7 we also include costs of DB methods by [BPP07] and [MD07b], which were analyzed in Section 5.2. We present there the cases for which those DB methods achieve the lowest costs. Nevertheless, we see that our methods (and also NAF) largely surpass them not only in computing costs but also in memory requirements. As stated before, although the conversion time and the table used to convert numbers to DB (see the column "Table") have been significantly reduced, the precomputation stage has become expensive, which ultimately increases the overall cost.

Finally, in Table 8 we present the lowest costs achieved by the Frac-*wmb*NAF method in comparison with previous efforts for the case of the highly efficient Edwards curves. As we can see, our method is the fastest for both Edwards and inverted Edwards coordinates, achieving slightly lower costs than the results by [BBL$^+$07] and [BL07c]. This is, to our knowledge, the lowest costs in terms of field operations reported in the literature for any elliptic curve over prime fields.

Table 6. Costs of different scalar multiplication methods (n = 160 bits)

Method	Curve	Precomputed table	Cost of precomputations	Cost of scalar multiplication	Total cost
(2,3)NAF	InvEdw.	$\{\pm 1\}$	-	1377.5M	1377.5M
NAF	InvEdw.	$\{\pm 1\}$	-	1447.2M	1447.2M
(2,3)NAF	Edw.	$\{\pm 1\}$	-	1414.3M	1414.3M
NAF	Edw.	$\{\pm 1\}$	-	1500.1M	1500.1M
(2,3,5)NAF	Jac.	$\{\pm 1\}$	-	1484M	1484M
(2,3)NAF	Jac.	$\{\pm 1\}$	-	1509.3M	1509.3M
NAF	Jac.	$\{\pm 1\}$	-	1615.2M	1615.2M
(2,3)NAF, case 2	InvEdw.	$\{\pm 1, \pm 2, \pm 3\}$	14.2M	1322.1M	1336.3M
(2,3)NAF, case 1	InvEdw.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 22.2M	1303.7M	1I + 1325.9M
wNAF, case 2	InvEdw.	$\{\pm 1, \pm 2, \pm 3\}$	14.2M	1339.6M	1353.8M
wNAF, case 1	InvEdw.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 22.2M	1318.7M	1I + 1340.8M
(2,3)NAF, case 2	Edw.	$\{\pm 1, \pm 2, \pm 3\}$	15.2M	1356.4M	1371.6M
(2,3)NAF, case 1	Edw.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 22.2M	1337.6M	1I + 1359.8M
wNAF, case 2	Edw.	$\{\pm 1, \pm 2, \pm 3\}$	15.2M	1379M	1394.2M
wNAF, case 1	Edw.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 22.2M	1357.7M	1I + 1379.9M
(2,3,5)NAF, case 2	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	11.6M	1496.4M	1508M
(2,3,5)NAF, case 1	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 19.4M	1426.1M	1I + 1445.5M
(2,3)NAF, case 2	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	11.6M	1525M	1536.6M
(2,3)NAF, case 1	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 19.4M	1444.6M	1I + 1464M
wNAF, case 2	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	11.6M	1567.1M	1578.7M
wNAF, case 1	Jac.	$\{\pm 1, \pm 2, \pm 3\}$	1I + 19.4M	1474.8M	1I + 1494.2M
(2,3)NAF, case 2	InvEdw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	33.8M	1257M	1290.8M
(2,3)NAF, case 1	InvEdw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	1I + 50.8M	1235.3M	1I + 1286.1M
wNAF, case 2	InvEdw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	33.8M	1267.2M	1301M
wNAF, case 1	InvEdw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	1I + 50.8M	1242.1M	1I + 1292.9M
(2,3)NAF, case 2	Edw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	36.8M	1283.8M	1320.6M
(2,3)NAF, case 1	Edw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	1I + 53.8M	1261.7M	1I + 1315.5M
wNAF, case 2	Edw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	36.8M	1298.8M	1335.6M
wNAF, case 1	Edw.	$\{\pm 1, \pm 2, \pm 3, \pm 5, \pm 7\}$	1I + 53.8M	1273.2M	1I + 1327M

(2,3,5)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$24.8M$	$1448.4M$	$1473.2M$
(2,3,5)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$1I + 40.6M$	$1362.1M$	$1I + 1402.7M$
(2,3)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$24.8M$	$1462.3M$	$1487.1M$
(2,3)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$1I + 40.6M$	$1367.5M$	$1I + 1408.1M$
*w*NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$24.8M$	$1501.9M$	$1526.7M$
*w*NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\pm5,\pm7\}$	$1I + 40.6M$	$1390.6M$	$1I + 1431.2M$
(2,3)NAF, case 2	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$53.4M$	$1231.2M$	$1288.7M$
(2,3)NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 78.4M$	$1210.4M$	$1I + 1284.6M$
Frac-*w*NAF, case 2	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$53.4M$	$1236M$	$1289.4M$
Frac-*w*NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 78.4M$	$1212.8M$	$1I + 1291.2M$
(2,3)NAF, case 2	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$58.4M$	$1256.5M$	$1314.9M$
(2,3)NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 85.4M$	$1235.2M$	$1I + 1320.6M$
Frac-*w*NAF, case 2	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$58.4M$	$1264.8M$	$1323.2M$
Frac-*w*NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 85.4M$	$1241M$	$1I + 1326.4M$
(2,3,5)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$38M$	$1419.8M$	$1457.8M$
(2,3,5)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 61.8M$	$1338.3M$	$1I + 1400.1M$
(2,3)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$38M$	$1434.1M$	$1472.1M$
(2,3)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 61.8M$	$1343.4M$	$1I + 1405.2M$
Frac-*w*NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$38M$	$1460.6M$	$1498.6M$
Frac-*w*NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm11\}$	$1I + 61.8M$	$1358.6M$	$1I + 1420.4M$
(2,3)NAF, case 2	**InvEdw.**	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	**$63.2M$**	**$1218.6M$**	**$1281.8M$**
(2,3)NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 92.2M$	$1199M$	$1I + 1291.2M$
Frac-*w*NAF, case 2	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$63.2M$	$1222.1M$	$1285.3M$
Frac-*w*NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 92.2M$	$1200.1M$	$1I + 1292.3M$
(2,3)NAF, case 2	**Edw.**	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	**$69.2M$**	**$1242.4M$**	**$1311.6M$**
(2,3)NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 101.2M$	$1222.2M$	$1I + 1323.4M$
Frac-*w*NAF, case 2	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$69.2M$	$1249.6M$	$1318.8M$
Frac-*w*NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 101.2M$	$1227M$	$1I + 1328.2M$
(2,3,5)NAF, case 2	**Jac.**	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	**$44.6M$**	**$1403.8M$**	**$1448.4M$**
(2,3,5)NAF, case 1	**Jac.**	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	**$1I + 72.4M$**	**$1326.5M$**	**$1I + 1398.9M$**
(2,3)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$44.6M$	$1414.7M$	$1459.3M$
(2,3)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 72.4M$	$1329.6M$	$1I + 1402M$
Frac-*w*NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$44.6M$	$1441.9M$	$1486.5M$
Frac-*w*NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm13\}$	$1I + 72.4M$	$1344.7M$	$1I + 1417.1M$

(2,3)NAF, case 2	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	73*M*	1211.6*M*	1284.6*M*
(2,3)NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 106*M*	1189.5*M*	1*I* + 1295.5*M*
*w*NAF, case 2	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	73*M*	1213.2*M*	1286.2*M*
*w*NAF, case 1	InvEdw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 106*M*	1188.9*M*	1*I* + 1294.9*M*
(2,3)NAF, case 2	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	80*M*	1235.1*M*	1315.1*M*
(2,3)NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 117*M*	1212.5*M*	1*I* + 1329.5*M*
*w*NAF, case 2	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	80*M*	1239.5*M*	1319.5*M*
*w*NAF, case 1	Edw.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 117*M*	1214.7*M*	1*I* + 1331.7*M*
(2,3,5)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	51.2*M*	1406.7*M*	1457.9*M*
(2,3,5)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 83*M*	1318.4*M*	1*I* + 1401.4*M*
(2,3)NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	51.2*M*	1418.4*M*	1469.6*M*
(2,3)NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 83*M*	1321.7*M*	1*I* + 1404.7*M*
*w*NAF, case 2	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	51.2*M*	1440.2*M*	1491.4*M*
*w*NAF, case 1	Jac.	$\{\pm1,\pm2,\pm3,\ldots,\pm15\}$	1*I* + 83*M*	1232.5*M*	1*I* + 1415.5*M*

Table 7. Comparison with previous methods in Jacobian coordinates (n = 160 bits).

Method	Table	# points	Cost of precomputations	Cost of scalar multiplication	Total cost
(2,3,5)NAF (this work)	-	7	44.6*M*	1403.8*M*	1448.4*M*
(2,3,5)NAF (this work)	-	7	1*I* + 72.4*M*	1326.5*M*	1*I* + 1398.9*M*
Frac-*w*NAF (this work)	-	7	44.6*M*	1441.9*M*	1486.5*M*
*w*NAF (this work)	-	8	1*I* + 83*M*	1232.5*M*	1*I* + 1415.5*M*
DB [BBL⁺07] (2)	3800	7	N/A	N/A	1504.3*M*
Frac-*w*NAF [BL07c]	-	7	N/A	N/A	1511.9*M*
*w*NAF [BL07c]	-	8	N/A	N/A	1*I* + 1434.1*M*
(2,3,5)NAF (this work)	-	0	-	1484*M*	1484*M*
(2,3)NAF (this work)	-	0	-	1509.3*M*	1509.3*M*
NAF (this work)	-	0	-	1615.2*M*	1615.2*M*
(3,1)DB, T_{all}^{P}, [BPP07]	8	11	1*I* + 143*M*	1478*M*	1*I* + 1621*M*
(4,3)DB, [MD07b] (1)(2)	4332	26	290.4*M*	1699.2*M*	1989.6*M*

(1) Values are not updated with state-of-the-art point operation formulae.

(2) Table stores all the possible $\{2^b \times 3^c\}$ values, where $b \le b_{\max}$ and $c \le c_{\max}$.

Table 8. Comparison with previous methods in Edwards and inverted Edwards coordinates (n = 160 bits).

Method	Curve	Table	# points	Total cost
(2,3)NAF (this work)	InvEdw.	-	7	1281.8M
Frac-wNAF [BL07c]	InvEdw.	-	7	1287.8M
DB [BBL$^+$07] (1)	InvEdw.	468	7	1290.3M
(2,3)NAF (this work)	Edw.	-	7	1311.6M
Frac-wNAF [BL07c]	Edw.	-	7	1321.6M
DB [BBL$^+$07] (1)	Edw.	468	7	1322.9M

(1) Table stores all the possible $\{2^b \times 3^c\}$ values, where $b \le b_{\max}$ and $c \le c_{\max}$.

7 Conclusions

We have introduced a new multibase method that uses “fractional” windows, generalizing previous *wmb*NAF and *mb*NAF methods to any number of precomputations. We have presented a more comprehensive analysis of scalar multiplications methods and tested their performance in comparison with the multibase NAF methods in the context of standard and Edwards curves. The conclusion is that *mb*NAF is currently the fastest in the literature among methods without precomputations, independently of the curve selected. Using bases (2,3,5) we can perform a scalar multiplication with a cost of only 1484M (field multiplications) in Jacobian coordinates. With inverted Edwards coordinates, that cost can be as low as 1377M using bases (2,3). Similar results are attained by the new Frac-*wmb*NAF when there is available memory to store precomputations. In this case, we present the lowest costs reported in the literature: 1448M or 1I+1398M in Jacobian coordinates and 1281M in inverted Edwards coordinates, both achieved with a fractional window using a precomputed table of 7 points.

We also conclude that in general double-base (DB) methods (and all its known variants) become impractical in comparison to other methods due to their higher memory requirements and / or costly precomputation stage. (w)NAF, (w)*mb*NAF and Frac-*wmb*NAF should be largely preferred in realistic scenarios.

Finally, we have contributed with improved composite operation formulas and precomputation schemes that have been beneficially used to accelerate significantly all the studied methods, including the efficient multibase NAF methods and the traditional NAF, wNAF and Frac-wNAF.

References

[BPP07] R. Barua, S.K. Pandey and R. Pankaj, "Efficient Window-Based Scalar Multiplication on Elliptic Curves using Double Base Number System," in *Progress in Cryptology - Indocrypt 2007*, LNCS Vol. 4859, pp. 351-360, Springer, 2007.

[BBL^{+}07] D. Bernstein, P. Birkner, T. Lange, and C. Peters, "Optimizing Double-Base Elliptic-Curve Single-Scalar Multiplication," in *Progress in Cryptology - Indocrypt 2007*, LNCS Vol. 4859, pp. 167-182, Springer, 2007.

[BL07a] D. Bernstein and T. Lange, "Faster Addition and Doubling on Elliptic Curves," in *Advances of Cryptology - Asiacrypt 2007*, LNCS Vol. 4833, pp. 29–50, Springer, 2007.

[BL07b] D. Bernstein and T. Lange, "Inverted Edwards Coordinates," in *Applied Algebra, Algebraic Algorithms, and Error Correcting Codes Symposium (AAECC 2007)*, 2007.

[BL07c] D. Bernstein and T. Lange, "Analysis and Optimization of Elliptic-Curve Single-Scalar Multiplication", in *Cryptology ePrint Archive*, Report 2007/455, 2007.

[CCJ04] B. Chevallier-Mames, M. Ciet and M. Joye, "Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity," in *IEEE Transactions on Computers*, Vol. 53, No 6, pp. 760-768, 2004.

[CMO98] H. Cohen, A. Miyaji, and T. Ono, "Efficient Elliptic Curve Exponentiation using Mixed Coordinates," in *Advances in Cryptology – Asiacrypt 1998*, LNCS Vol. 1514, pp. 51–65, Springer, 1998.

[DI06] C. Doche and L. Imbert, "Extended Double-Base Number System with Applications to Elliptic Curve Cryptography," in *Progress in Cryptology - Indocrypt 2006*, LNCS Vol. 4329, pp. 335-348, Springer, 2006.

[DIM05] V. Dimitrov, L. Imbert and P.K. Mishra, "Efficient and Secure Elliptic Curve Point Multiplication using Double-Base Chains," in *Advances in Cryptology – Asiacrypt 2005*, LNCS Vol. 3788, pp. 59–78, Springer, 2005.

[DOS07] E. Dahmen, K. Okeya and D. Schepers, "Affine Precomputation with Sole Inversion in Elliptic Curve Cryptography," in *Australasian Conference on Information Security and Privacy (ACISP'07)*, LNCS Vol. 4586, pp. 245-258, Springer, 2007.

[Edw07] H.M. Edwards, "A Normal Form for Elliptic Curves," in *Bulletin of the American Mathematical Society*, Vol. 44, pp. 393–422, 2007.

[EPAF] P. Longa, "ECC Point Arithmetic Formulae," available at http://patricklonga.bravehost.com/jacobian.html.

[HMV04] D. Hankerson, A. Menezes and S. Vanstone, "Guide to Elliptic Curve Cryptography," Springer-Verlag, 2004.

[Kob87] N. Koblitz, “Elliptic Curve Cryptosystems,” in *Mathematics of Computation*, Vol. 48, pp. 203–209, 1987.

[LM07a] P. Longa and A. Miri, “Fast and Flexible Elliptic Curve Point Arithmetic over Prime Fields,” in *IEEE Transactions on Computers*, Vol. 57, No 3, pp. 289-302, 2008. Also available at:
http://doi.ieeecomputersociety.org/10.1109/TC.2007.70815.

[LM07b] P. Longa and A. Miri, “New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields,” in *Public Key Cryptography (PKC'08)*, LNCS Vol. 4939, pp. 229-247, Springer, 2008.

[LM08] P. Longa and A. Miri, “New Multibase Non-Adjacent Form Scalar Multiplication and its Application to Elliptic Curve Cryptosystems (Extended Version),” in *Cryptology ePrint Archive*, Report 2008/052, 2008.

[Lon07] P. Longa, “Accelerating the Scalar Multiplication on Elliptic Curve Cryptosystems over Prime Fields,” *Master's Thesis*, University of Ottawa, June 2007. Also available at http://patricklonga.bravehost.com/publications.html.

[MD07a] Mishra and V. Dimitrov, “Efficient Quintuple Formulas for Elliptic Curves and Efficient Scalar Multiplication using Multibase Number Representation,” in *Cryptology ePrint Archive*, Report 2007/040, 2007.

[MD07b] P.K. Mishra and V. Dimitrov, “Window-Based Elliptic Curve Scalar Multiplication using Double Base Number Representation,” in *Inscrypt 2007*, Short Papers, 2007.

[Mel06] N. Meloni, “Fast and Secure Elliptic Curve Scalar Multiplication over Prime Fields using Special Addition Chains,” in *Cryptology ePrint Archive*, Report 2006/216, 2006.

[Mil85] V. Miller, “Use of Elliptic Curves in Cryptography,” in *Advances in Cryptology - Crypto'85*, LNCS Vol. 218, pp. 417-426, Springer, 1986.

[Möl02] B. Möller, “Improved Techniques for Fast Exponentiation,” in *ICISC'02*, LNCS Vol. 2587, pp. 298–312, Springer, 2003.

[Möl04] B. Möller, “Fractional Windows Revisited: Improved Signed - Digit Representations for Efficient Exponentiation,” in *ICISC'04*, LNCS Vol. 3506, pp. 137–153, Springer, 2005.

[Rei60] G.W. Reitweisner, “Binary Arithmetic,” in *Adv. Comput.*, Vol. 1, pp. 232–308, 1960.

[Sol00] J.A. Solinas, “Efficient Arithmetic on Koblitz Curves,” in *Design, Codes and Cryptography*, Vol. 19, pp. 195–249, 2000.

Appendix A: Point Quintupling

Given $P = (X_1, Y_1, Z_1)$ on the elliptic curve *E*, the quintupling $5P = (X_5, Y_5, Z_5)$ in Jacobian coordinates (special case *a* = –3) can be computed by:

$$X_5 = \gamma^2 - 4\phi^3 - 8X_2^{(1)}\phi^2\ ,\ Y_5 = \gamma[4X_2^{(1)}\phi^2 - X_5] - 8Y_2^{(1)}\phi^3\ ,\ Z_5 = 2Z_2[(\theta+\phi)^2 - \theta^2 - \phi^2]\ ,$$

where $\alpha = 3(X_1 + Z_1^2)(X_1 - Z_1^2)$, $X_2 = \alpha^2 - 2X_1^{(1)}$, $2Y_2 = (\alpha+\theta)^2 - \alpha^2 - \theta^2 - 2Y_1^{(1)}$,

$\theta = X_1^{(1)} - X_2$, $\omega = 2Y_1^{(1)} - 2Y_2$, $\gamma = \omega^2 + \phi^2 - (\omega+\phi)^2 - 4Y_2^{(1)}$, $X_1^{(1)} = 4X_1Y_1^2$,

$2Y_1^{(1)} = 16Y_1^4$, $\phi = \omega^2 - 4\theta^3 - 3X_2^{(1)}$, $X_2^{(1)} = 4X_2\theta^2$, $Y_2^{(1)} = 8Y_2\theta^3$,

$Z_2 = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2$.

This quintupling formula derived from the Scheme (5) (see Section 3.2) costs 10*M* + 12*S*. The general case (parameter *a* random) can be easily derived from the formula above. In this case, the cost is fixed at 9*M* + 15*S* with the following change of parameters:

$$\alpha = 3X_1^2 + aZ_1^4\ ,\ \ X_1^{(1)} = 2[(X_1 + Y_1^2)^2 - X_1^2 - Y_1^4]\ .$$

Appendix B: Tripling with mixed Jacobian-affine coordinates

Given $P = (x_1, y_1)$ on the elliptic curve *E*, the tripling $3P = (X_3, Y_3, Z_3)$ in Jacobian coordinates can be computed by:

$$X_3 = 4y_1^2(2\beta - 2\alpha) + x_1\omega^2\ ,$$
$$Y_3 = y_1[(2\alpha - 2\beta)(4\beta - 2\alpha) - \omega^3]\ ,$$
$$Z_3 = 6[(x_1 + y_1^2)^2 - x_1^2 - y_1^4] - \theta^2\ ,$$

where $2\alpha = (\theta+\omega)^2 - \theta^2 - \omega^2$, $\beta = 8Y_1^4$, $\theta = 3(x_1^2 - 1)$. This tripling formula has a cost of 5*M* + 7*S*.

Appendix C: Quintupling with mixed Jacobian-affine coordinates

If we set $Z_1 = 1$ in the quintupling formula given in Appendix A ($a = -3$), we obtain the following formula to compute $5P = 5(x_1, y_1) = (X_5, Y_5, Z_5)$ in Jacobian coordinates:

$$X_5 = \gamma^2 - 4\phi^3 - 8X_2^{(1)}\phi^2 \,,\ Y_5 = \gamma[4X_2^{(1)}\phi^2 - X_5] - 8Y_2^{(1)}\phi^3 \,,\ Z_5 = 4y_1[(\theta+\phi)^2 - \theta^2 - \phi^2] \,,$$

where $\alpha = 3(x_1^2 - 1)$, $X_2 = \alpha^2 - 2X_1^{(1)}$, $2Y_2 = (\alpha+\theta)^2 - \alpha^2 - \theta^2 - 2Y_1^{(1)}$,
$\theta = X_1^{(1)} - X_2$, $\omega = 2Y_1^{(1)} - 2Y_2$, $\gamma = \omega^2 + \phi^2 - (\omega+\phi)^2 - 4Y_2^{(1)}$, $2Y_1^{(1)} = 16y_1^4$,
$X_1^{(1)} = 2[(x_1 + y_1^2)^2 - x_1^2 - y_1^4]$, $\phi = \omega^2 - 4\theta^3 - 3X_2^{(1)}$, $X_2^{(1)} = 4X_2\theta^2$,
$Y_2^{(1)} = 8Y_2\theta^3$.

This quintupling with mixed Jacobian-affine coordinates costs $8M + 12S$.

Appendix D: Tripling with mixed Edwards-affine coordinates

Given $P = (x_1, y_1)$, the tripling $3P = (X_3, Y_3, Z_3)$ in Edwards coordinates can be obtained with the following:

$$X_3 = x_1[\beta - (\alpha - \theta)](\alpha - \theta) \,,$$
$$Y_3 = y_1[\beta + (\alpha - \omega)](\alpha - \omega) \,,$$
$$Z_3 = [\beta - (\alpha - \theta)][\beta + (\alpha - \omega)] \,,$$

where $\alpha = (x_1^2 + y_1^2)^2$, $\beta = 2(x_1^2 + y_1^2)(x_1^2 - y_1^2)$, $\omega = 4c^2x_1^2$, $\theta = 4c^2y_1^2$. This tripling formula has a cost of only $6M + 3S$ if we fix the parameter c to a small value.

Appendix E: Tripling with mixed inverted Edwards-affine coord.

Given $P = (x_1, y_1)$, the tripling $3P = (X_3, Y_3, Z_3)$ in inverted Edwards coordinates can be obtained with the following:

$$X_3 = x_1\phi\alpha \,,\ Y_3 = y_1\omega\beta \,,\ Z_3 = \omega\phi \,,$$

where $\theta = x_1^2 + y_1^2$, $\alpha = -\theta^2 + 4y_1^2dc^2$, $\beta = \theta^2 - 4x_1^2dc^2$,
$\omega = \theta(\theta - 4x_1^2) + 4x_1^2dc^2$, $\phi = \theta(\theta - 4y_1^2) + 4y_1^2dc^2$.

This tripling formula has a cost of only $7M + 3S$ if we fix the parameter c and d to small values.