Fast and Regular Algorithms for Scalar Multiplication over Elliptic Curves

Matthieu Rivain

CryptoExperts
matthieu.rivain@cryptoexperts.com

Abstract. Elliptic curve cryptosystems are more and more widespread in everyday-life applications. This trend should still gain momentum in coming years thanks to the exponential security enjoyed by these systems compared to the subexponential security of other systems such as RSA. For this reason, efficient elliptic curve arithmetic is still a hot topic for cryptographers. The core operation of elliptic curve cryptosystems is the scalar multiplication which multiplies some point on an elliptic curve by some (usually secret) scalar. When such an operation is implemented on an embedded system such as a smart card, it is subject to *side channel attacks*. To withstand such attacks, one must constrain the scalar multiplication algorithm to be *regular*, namely to have an operation flow independent of the input scalar. A large amount of work has been published that focus on efficient and regular scalar multiplication and the choice leading to the best performances in practice is not clear. In this paper, we look into this question for general-form elliptic curves over large prime fields and we complete the current state-of-the-art. One of the fastest low-memory algorithms in the current literature is the Montgomery ladder using co-Z Jacobian arithmetic *with X and Y coordinates only*. We detail the regular implementation of this algorithm with various trade-offs and we introduce a new binary algorithm achieving comparable performances. For implementations that are less constrained in memory, windowing techniques and signed exponent recoding enable reaching better timings. We survey regular algorithms based on such techniques and we discuss their security with respect to side-channel attacks. On the whole, our work give a clear view of the currently best time-memory trade-offs for regular implementation of scalar multiplication over prime-field elliptic curves.

1 Preliminaries

Let E be an elliptic curve defined over $\mathbb{F}_q$ with $q > 3$ according to the following *short Weierstrass equation*:

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

where $a, b \in \mathbb{F}_q$ such that $4a^3 + 27b^2 \neq 0$. The set of rational points of E is the set $E(\mathbb{F}_q)$ of points $(x, y) \in \mathbb{F}_q^2$ whose coordinates satisfy (1). The rational points of E, augmented with a neutral element $\boldsymbol{\mathcal{O}}$ called *point at infinity*, have an Abelian group structure. The associated addition law – which is usually called *chord-and-tangent rule* – computes the sum of two points $\boldsymbol{P} = (x_1, y_1)$ and $\boldsymbol{Q} = (x_2, y_2)$ as $\boldsymbol{P} + \boldsymbol{Q} = (x_3, y_3)$ where:

$$x_3 = \lambda^2 - x_1 - x_2 \,, \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1 \,, \quad \text{with} \quad \lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & \text{if } \boldsymbol{P} \neq \boldsymbol{Q}, \\ \frac{3x_1^2 + a}{2y_1} & \text{if } \boldsymbol{P} = \boldsymbol{Q}. \end{cases} \tag{2}$$

The scalar multiplication of a point $\boldsymbol{P}$ by a natural integer k is denoted $[k]\boldsymbol{P}$. The discrete logarithm in basis $\boldsymbol{P}$ of some point $\boldsymbol{Q} = [k]\boldsymbol{P}$ is then the integer k. Several cryptographic protocols have been designed whose security relies on the difficulty of computing the discrete logarithm over the rational points of an elliptic curve (only exponential-time algorithms are known to solve this problem in general). The scalar multiplication is the core operation of most of these protocols. Efficient scalar multiplication arithmetic is hence a central issue for cryptography. The interested reader is referred to [HMV03] for a good overview of the question.

Point addition formulae – such as (2) – are based on different operations over $\mathbb{F}_q$ (*e.g.* multiplication, inversion, addition, and subtraction) which have different computational costs. In this paper, we shall denote by $\mathbf{I}$, $\mathbf{M}$, $\mathbf{S}$, and A, the computational costs of a field inversion, a field multiplication, a field squaring, and a field addition respectively. We shall further consider that a field subtraction and a field doubling have the same cost than a field addition. In the context where q is a large prime, it is often

assumed that (i) the inversion cost satisfies $\mathbf{I} \approx 100\mathbf{M}$, (ii) the squaring cost is $\mathbf{S} = 0.8\mathbf{M}$, and (iii) the addition cost can be neglected. These assumptions are derived from the usual software implementations of the field operations. However when the latter are based on a hardware coprocessor – as it is often the case in embedded systems – their costs become architecture-reliant. In general, the inversion shall always cost a few dozens of multiplications, the squaring cost shall be of the same order than the multiplication one (possibly a bit cheaper), and the addition cost shall be clearly lower, but not always negligible (see for instance [GV10]). In the following, we shall take interest into the computational cost of different point addition formulae in terms of field operations. We shall also focus on their memory usage in terms of *field registers*, namely memory registers of size $\log_2(q)$ bits that can store elements of $\mathbb{F}_q$.

This paper only deals with general-form elliptic curves that may be chosen arbitrarily among the set of curves satisfying (1). Note that elliptic curves with special forms exist (*e.g.* Montgomery curves, Edwards curves) which have performance advantages over general-form elliptic curves (see for instance [BL]). However many applications require the compliance with arbitrarily chosen elliptic curves, which motivates the investigation for efficient algorithms over general-form elliptic curves.

1.1 Jacobian Coordinates and Improvements

When points are represented in so-called *affine coordinates* as in (2), the addition of two points involves an expensive field inversion. Fortunately, it is possible to avoid this cost for the intermediate point additions in a scalar multiplication by representing points in *projective coordinates*. In projective coordinates, a point $\boldsymbol{P} = (x, y)$ is represented by a triplet (X, Y, Z) where $(X/Z^c, Y/Z^d) = (x, y)$ for some given integers c and d. A point has several projective representations (as many as different Z). The equivalence class containing (X, Y, Z) and all the other projective representations of the affine point $(X/Z^c, Y/Z^d)$ is denoted $(X : Y : Z)$. We further denote $\boldsymbol{P} \equiv (X : Y : Z)$ if $(X : Y : Z)$ is the equivalence class of $\boldsymbol{P}$.

The most widely used projective coordinates are the *Jacobian coordinates* for which $c = 2$ and $d = 3$. These coordinates enable fast point doubling which is usually the most frequent operation in a scalar multiplication algorithm. Let $\boldsymbol{P} = (X_1, Y_1, Z_1)$, the Jacobian doubling of $\boldsymbol{P}$ is defined as $\boldsymbol{P} + \boldsymbol{P} = (X_3, Y_3, Z_3)$ where:

$$X_3 = B^2 - 2A\ , \quad Y_3 = B(A - X_3) - 8Y_1^4 \ \text{ and } \ Z_3 = 2Y_1Z_1\ , \tag{3}$$

with $A = 4X_1Y_1^2$, $B = 3X_1^2 + aZ_1^4$. The Jacobian doubling can be computed at the cost of $4\mathbf{M} + 6\mathbf{S} + 11\mathrm{A}$ using 6 field registers. For the particular case where $a = -3$, we have $B = 3(X_1 + Z_1^2)(X_1 - Z_1^2)$. This equality enables trading $1\mathbf{M} + 2\mathbf{S} + 3\mathrm{A}$ for $1\mathbf{M} + 4\mathrm{A}$ in the computation of B. The Jacobian doubling can then be computed at the cost of $4\mathbf{M} + 4\mathbf{S} + 12\mathrm{A}$ using 5 field registers. When $a \neq -3$, an optimization is still possible when several doublings are performed successively [CMO98,ITT+99]. Let (X_i, Y_i, Z_i) denote the input of the ith doubling. The optimization consists in holding the value aZ_i^4 obtained in the $(i-1)$th doubling, and using it in the subsequent computation as $aZ_{i+1}^4 = a(2Y_iZ_i)^4 = (8Y_i^4)(aZ_i^4)$. As the computation of $8Y_i^4$ is already involved in the ith point doubling, one can then trade $1\mathbf{M} + 2\mathbf{S}$ for $1\mathbf{M}$ in the computation of aZ_{i+1}^4. The Jacobian doubling can then be computed at the cost of $4\mathbf{M} + 4\mathbf{S} + 11\mathrm{A}$ using 6 field registers. The quadruplet composed of the usual Jacobian coordinates together with aZ^4 is called *modified Jacobian coordinates* in [CMO98].

Now let $\boldsymbol{Q} = (X_2, Y_2, Z_2)$, the Jacobian addition of $\boldsymbol{P}$ and $\boldsymbol{Q}$ (with $\boldsymbol{P} \neq \boldsymbol{Q}$) is defined as $\boldsymbol{P} + \boldsymbol{Q} = (X_3, Y_3, Z_3)$ where:

$$X_3 = -E^3 - 2AE^2 + F\ , \quad Y_3 = -CE^3 + F(AE^2 - X^3) \ \text{ and } \ Z_3 = Z_1Z_2E\ , \tag{4}$$

with $A = X_1Z_2^2$, $B = X_2Z_1^2$, $C = Y_1Z_2^3$, $D = Y_2Z_1^3$, $E = B - A$, $F = D - C$. The Jacobian addition can be computed at the cost of $12\mathbf{M} + 4\mathbf{S} + 7\mathrm{A}$ using 7 field registers. A strategy to speed up the Jacobian addition is to use *mixed-Jacobian-affine* coordinates such as suggested in [CMO98]. Consider that the point $\boldsymbol{Q}$ is represented in affine coordinates (namely $Z_2 = 1$), it is easy to see that one can save $4\mathbf{M} + 1\mathbf{S}$ in the computation of A, C, and Z_3. The obtained mixed-Jacobian-affine addition can then be computed at the cost of $8\mathbf{M} + 3\mathbf{S} + 7\mathrm{A}$ using 7 field registers. Low level algorithms for point addition and doubling formulae discussed above can be found in [IMT02] with detailed operation flow and memory usage.

A further optimization of the Jacobian addition was put forward by Meloni in [Mel07]. It considers two input points sharing the same Z-coordinate. Let $\boldsymbol{P} = (X_1, Y_1, Z)$ and $\boldsymbol{Q} = (X_2, Y_2, Z)$, the so-called

co-Z addition of $\boldsymbol{P}$ and $\boldsymbol{Q}$ (with $\boldsymbol{P} \neq \boldsymbol{Q}$) is defined as $\boldsymbol{P} + \boldsymbol{Q} = (X_3, Y_3, Z_3)$, where:

$$X_3 = D - (B + C)\ , \quad Y_3 = (Y_2 - Y_1)(B - X_3) - E \ \text{ and } \ Z_3 = Z(X_2 - X_1)\ , \tag{5}$$

with $A = (X_2 - X_1)^2$, $B = X_1 A$, $C = X_2 A$, $D = (Y_2 - Y_1)^2$ and $E = Y_1(C - B)$. This co-Z addition is very efficient: it can be computed at the cost of $5\mathbf{M} + 2\mathbf{S} + 7\mathrm{A}$ using 6 field registers. Moreover, it also makes it possible to update the coordinates of $\boldsymbol{P}$ for free such that it shares the same Z-coordinate than $\boldsymbol{P} + \boldsymbol{Q}$. Indeed, a close look at the expression of B and E reveals that $B = X_1(X_2 - X_1)^2 = x_1 Z_3^2$ and $E = Y_1(X_2 - X_1)^3 = y_1 Z_3^3$ where $(x_1, y_1) = (X_1/Z^2, Y_1/Z^3)$ denotes the affine coordinates of $\boldsymbol{P}$; we hence have $\boldsymbol{P} \equiv (E : B : Z_3)$. Such a free update enables the subsequent use of the co-Z addition between $\boldsymbol{P} + \boldsymbol{Q}$ and $\boldsymbol{P}$. It was also shown in [VD10,GJM10a] that the conjugate $\boldsymbol{P} - \boldsymbol{Q}$ sharing the same Z-coordinate than $\boldsymbol{P} + \boldsymbol{Q}$ can be obtained with a small additional cost. Indeed, $\boldsymbol{P} - \boldsymbol{Q} = (X_3', Y_3', Z_3)$ where:

$$X_3' = F - (B + C) \ \text{ and } \ Y_3' = (Y_1 + Y_2)(X_3' - B) - E\ , \tag{6}$$

with $F = (Y_1 + Y_2)^2$ (and A, B, C, D, and E defined as in (5)). The conjugate addition computing the two co-Z points $(\boldsymbol{P} + \boldsymbol{Q})$ and $(\boldsymbol{P} - \boldsymbol{Q})$ from two co-Z points $\boldsymbol{P}$ and $\boldsymbol{Q}$ can be computed at the cost of $6\mathbf{M} + 3\mathbf{S} + 16\mathrm{A}$ using 7 field registers[1]. Low level algorithms for co-Z (conjugate) addition formulae are given in [GJM10b].

As noticed in [VD10], it is possible to compute several succesive co-Z (conjugate) additions without computing the Z-coordinate. It can indeed be checked from (5) and (6) that the X and Y coordinates resulting from a co-Z addition do not depend on the input Z-coordinate. This trick enables to save one multiplication per co-Z (conjugate) addition provided that the underlying scalar multiplication algorithm enables the final Z-coordinate recovery. Such algorithms are presented in Section 2.

1.2 Binary Scalar Multiplication Algorithms

The problem of designing a scalar multiplication algorithm from point addition and doubling formulae is similar to that of computing an exponentiation from multiplications and squares. A simple and efficient algorithm is the binary exponentiation – or *binary scalar multiplication* in the context of elliptic curves – also known as the square-and-multiply algorithm – or *double-and-add algorithm*. The binary algorithm processes a loop scanning the bits of the scalar and performing a point doubling, followed by a point addition whenever the current scalar bit equals 1. Two versions of the binary algorithm exist that either scan the scalar bits in a *left-to-right* direction or in a *right-to-left* direction.

Let k be an integer with binary expansion $(k_{n-1}, \ldots, k_1, k_0)_2$, namely $k = \sum_i k_i 2^i$, where $k_i \in \{0, 1\}$ for every $i < n - 1$ and $k_{n-1} = 1$. The binary scalar multiplication of some point $\boldsymbol{P}$ by k can be understood as follows. Defining $T_i = [(k_{n-1}, \ldots, k_i)_2]\boldsymbol{P}$, we get a backward sequence where $T_{n-1} = \boldsymbol{P}$, $T_0 = [k]\boldsymbol{P}$ and $T_i = 2T_{i+1} + k_i\boldsymbol{P}$, which leads to the left-to-right binary algorithm (see Algorithm 1). On the other hand, we have $[k]\boldsymbol{P} = \sum_i k_i[2^i]\boldsymbol{P}$ which directly yields the right-to-left binary algorithm (see Algorithm 2).

Algorithm 1 Left-to-right binary algorithm

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $R_0 \leftarrow \boldsymbol{P}$; $R_1 \leftarrow \boldsymbol{P}$
2. **for** $i = n - 2$ **downto** 0 **do**
3. $\quad R_0 \leftarrow 2R_0$
4. $\quad$ **if** $k_i = 1$ **then** $R_0 \leftarrow R_0 + R_1$
5. **end for**
6. **return** R_0

Algorithm 2 Right-to-left binary algorithm

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $R_0 \leftarrow \boldsymbol{\mathcal{O}}$; $R_1 \leftarrow \boldsymbol{P}$
2. **for** $i = 1$ **to** $n - 1$ **do**
3. $\quad$ **if** $k_i = 1$ **then** $R_0 \leftarrow R_0 + R_1$
4. $\quad R_1 \leftarrow 2R_1$
5. **end for**
6. **return**

Both binary algorithms involve n point doublings and $n/2$ point additions on average. In order to lower the number of point additions it is possible to use a signed representation of the exponent

[1] As we shall argue in Appendix A, it is possible to save $3\mathrm{A}$ using an additional field register.

$k = \sum_i \kappa_i 2^i$ where $\kappa_i \in \{-1, 0, 1\}$. The binary algorithm remains the same but when $\kappa_i = -1$, the point in R_1 is subtracted to the point in R_0 instead of being added. The subtraction over an elliptic curve is essentially equivalent to the addition since subtracting some point $\boldsymbol{P} = (X_1, Y_1, Z_1)$ to another point $\boldsymbol{Q}$, is done by inverting its Y-coordinate to get $-\boldsymbol{P} = (X_1, -Y_1, Z_1)$ and then by adding $-\boldsymbol{P}$ to $\boldsymbol{Q}$. Moreover, there exists a signed representation of the scalar called *non-adjacent form* (NAF) that only has $n/3$ non-zero signed bits on average (see for instance [MO90,HMV03]). The number of point additions performed by the binary algorithm using NAF representation of the scalar hence falls to $n/3$ on average.

The above binary algorithms are simple and efficient, however they are insecure in a context where the scalar is secret and where the implementation is subject to *side-channel analysis* (*e.g.* a smart card performing an ECDSA signature). Side-channel analysis (SCA) exploits the physical information leakage produced by a device during a cryptographic computation such as its power consumption and its electromagnetic radiations [KJJ99,QS02,AARR02]. Scalar multiplication implementation are vulnerable to two main kinds of SCA: *simple power analysis* (SPA) and *differential power analysis* (DPA). The latter uses correlations between the leakage and processed data, and it can usually be efficiently defeated by the use of randomization techniques [Cor99,JT01,CJ03]. On the other hand, an SPA can recover the secret scalar from a single leakage trace of a binary algorithm computation (even in the presence of data randomization). The reason of such a flaw is that point additions and point doublings have different operation flows and hence induce different leakage patterns. The leakage trace is hence composed of several points doubling patterns interleaved by point addition patterns only for the loop iterations where the scalar bit equals 1 (or -1 in a signed representation). Consequently, a single leakage trace of the scalar multiplication can reveal the whole secret scalar (or significant information about it in a signed representation).

In order to withstand SPA, one must render the scalar multiplication *regular*, namely such that it performs a constant operation flow whatever the scalar value. A first possibility is to make addition and doubling patterns indistinguishable. This can be achieved by using unified formulae for point addition and point doubling [BJ02] or by the mean of *side-channel atomicity* whose principle is to build point addition and point doubling algorithms from the same atomic pattern of field operations [CMCJ04]. Another possibility is to render the scalar multiplication algorithm itself regular, independently of the field operation flows in each point operation. Namely, one designs a scalar multiplication with a constant flow of point operations. This approach was first followed by Coron in [Cor99] who proposed to perform a dummy addition in the binary algorithm loop whenever the scalar bit equals 0. The obtained *double-and-add-always* algorithm performs a point doubling and a point addition at every loop iteration and the scalar bits are no more distinguishable from a leakage trace. Using Jacobian coordinates for R_0 (assuming $a = -3$) and affine coordinates for R_1, the left-to-right double-and-add-always algorithm performs a scalar multiplication at the cost of $12\mathbf{M} + 7\mathbf{S} + 19\mathrm{A}$ per loop iteration.

Other regular binary algorithms exist in the literature which enjoy attractive features, such as the Montgomery ladder [Mon87] and the double-and-add algorithm proposed by Joye in [Joy07] (see Algorithms 3 and 4). These algorithms are based on loop invariants the point registers R_0 and R_1. In the Montgomery ladder, the relation $R_1 - R_0 = \boldsymbol{P}$ is satisfied at the end of every loop iteration, while in Joye algorithm the ith loop iteration yields $R_0 + R_1 = [2^i]\boldsymbol{P}$ (see [Mon87,Joy07] for further details).

Algorithm 3 Montgomery ladder

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $R_0 \leftarrow \boldsymbol{\mathcal{O}}$; $R_1 \leftarrow \boldsymbol{P}$
2. **for** $i = n - 1$ **downto** 0 **do**
3. $\quad b \leftarrow k_i$; $R_{1-b} \leftarrow R_{1-b} + R_b$
4. $\quad R_b \leftarrow 2R_b$
5. **end for**
6. **return** R_0

Algorithm 4 Joye double-and-add

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $R_0 \leftarrow \boldsymbol{\mathcal{O}}$; $R_1 \leftarrow \boldsymbol{P}$
2. **for** $i = 0$ **to** $n - 1$ **do**
3. $\quad b \leftarrow k_i$
4. $\quad R_{1-b} \leftarrow 2R_{1-b} + R_b$
5. **end for**
6. **return** R_0

The Montgomery ladder was initially proposed as a scalar multiplication algorithm for a specific kind of elliptic curves with very efficient point arithmetic: the so-called *Montgomery curves* [Mon87]. The Montgomery proposal also reaches additional speed up by only computing (X, Z)-coordinates of inter-

mediate points. This is made possible as the Montgomery ladder involves a so-called *differential addition* which computes the sum of two points whose difference is known. This approach was subsequently generalized to elliptic curves in the Weierstrass form [BJ02,IT02,IMT02]. In particular, it was shown in [IMT02] that a loop iteration of the Montgomery ladder using (X, Z)-coordinates only can be performed in $11\mathbf{M} + 4\mathbf{S} + 2\mathbf{M_a} + 18\mathrm{A}$ where $\mathbf{M_a}$ denotes the cost of the multiplication by the curve parameter a (which is a few additions if a is small *e.g.* $a = -3$).

The Montgomery ladder and the Joye double-and-add also provide efficient regular scalar multiplications based on co-Z addition formulae. In [VD10], Venelli and Dassande proposed several variants of the Montgomery ladder based on co-Z arithmetic. The most efficient variant is based on (X, Y)-only co-Z point additions and it reaches a computational cost of $9\mathbf{M} + 5\mathbf{S}$ (plus several additions) per bit of the scalar. Independently of this work, Goundar *et al.* applied co-Z arithmetic to the Montgomery ladder and the Joye double-and-add in [GJM10a]. The resulting algorithms involve $11\mathbf{M} + 5\mathbf{S} + 23\mathrm{A}$ per scalar bit, which can be reduced to $9\mathbf{M} + 7\mathbf{S} + 27\mathrm{A}$ using standard tricks. Recently, Hutter *et al.* proposed Montgomery ladder algorithms with (X, Z)-only projective co-Z arithmetic in [HJS11]. Their fastest variant involve $10\mathbf{M} + 5\mathbf{S} + 13\mathrm{A}$ per scalar bit.

In the next section we investigate regular binary algorithms based on Jacobian (X, Y)-only co-Z arithmetic. After detailing the point operation algorithms, we propose a new binary algorithm using these operations and based on a *full signed expansion* of the scalar. Then we detail the Montgomery ladder algorithm based on such arithmetic. Compared to [VD10], we propose several tarde-offs and we save a few field additions per scalar bit as well as a few memory registers.

2 Regular Binary Algorithms from Jacobian (X, Y)-only Co-Z Arithmetic

Algorithms 5 and 6 give the (X, Y)-only versions of the co-Z addition with update and the co-Z conjugate addition. The so-called XYCZ-ADD algorithm takes the (X, Y)-coordinates of two co-Z points $\boldsymbol{P}$ and $\boldsymbol{Q}$ and it computes the (X, Y)-coordinates of $\boldsymbol{P} + \boldsymbol{Q}$ and the update (X, Y)-coordinates of $\boldsymbol{P}$ (*i.e.* such that $\boldsymbol{P}$ and $\boldsymbol{P} + \boldsymbol{Q}$ are co-Z). On the other hand, the XYCZ-ADDC algorithm computes the (X, Y)-coordinates of $\boldsymbol{P} + \boldsymbol{Q}$ and of its co-Z conjugate $\boldsymbol{P} - \boldsymbol{Q}$. The low level descriptions of these algorithms are given in Appendix A. They show that the addition with update can be computed at the cost of $4\mathbf{M} + 2\mathbf{S} + 7\mathrm{A}$ using 5 field registers. For the conjugate addition, a time-memory trade-off is possible. A first implementation involves $5\mathbf{M} + 3\mathbf{S} + 11\mathrm{A}$ and requires 7 field registers, while a second one uses only 6 field registers for a cost of $5\mathbf{M} + 3\mathbf{S} + 16\mathrm{A}$ (see Appendix A). In comparison, the implementation proposed in [VD10] (LightAddSub algorithm) involves $5\mathbf{M} + 3\mathbf{S} + 13\mathrm{A}$ and it requires 9 field registers.

Algorithm 5 (X, Y)-only co-Z addition with update – XYCZ-ADD

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_1', Y_1') s.t. $\boldsymbol{P} \equiv (X_1' : Y_1' : Z_3)$ and $\boldsymbol{P} + \boldsymbol{Q} \equiv (X_3 : Y_3 : Z_3)$ for some $Z_3 \in \mathbb{F}_q$

1. $A \leftarrow (X_2 - X_1)^2$
2. $B \leftarrow X_1 A$
3. $C \leftarrow X_2 A$
4. $D \leftarrow (Y_2 - Y_1)^2$
5. $E \leftarrow Y_1(C - B)$
6. $X_3 \leftarrow D - (B + C)$
7. $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$
8. $X_1' \leftarrow B$
9. $Y_1' \leftarrow E$
10. **return** $\big((X_3, Y_3), (X_1', Y_1')\big)$

Algorithm 6 (X, Y)-only co-Z conjugate addition – XYCZ-ADDC

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_3', Y_3') s.t. $\boldsymbol{P} + \boldsymbol{Q} \equiv (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} - \boldsymbol{Q} \equiv (X_3' : Y_3' : Z_3)$ for some $Z_3 \in \mathbb{F}_q$

1. $A \leftarrow (X_2 - X_1)^2$
2. $B \leftarrow X_1 A$
3. $C \leftarrow X_2 A$
4. $D \leftarrow (Y_2 - Y_1)^2$; $F \leftarrow (Y_1 + Y_2)^2$
5. $E \leftarrow Y_1(C - B)$
6. $X_3 \leftarrow D - (B + C)$
7. $Y_3 \leftarrow (Y_2 - Y_1)(B - X_3) - E$
8. $X_3' \leftarrow F - (B + C)$
9. $Y_3' \leftarrow (Y_1 + Y_2)(X_3' - B) - E$
10. **return** $\big((X_3, Y_3), (X_3', Y_3')\big)$

In [GJM10a] a co-Z double-add algorithm is proposed. This algorithm first performs a co-Z addition (with update) between $\boldsymbol{P}$ and $\boldsymbol{Q}$ leading to a pair of co-Z points $(\boldsymbol{R}, \boldsymbol{P})$ where $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$. Then a

conjugate co-Z addition is processed that yields the pair of co-Z points $(\boldsymbol{R}+\boldsymbol{P}, \boldsymbol{R}-\boldsymbol{P}) = (2\boldsymbol{P}+\boldsymbol{Q}, \boldsymbol{Q})$. Algorithm 7 gives the (X, Y)-only version of the co-Z double-add with update.

Algorithm 7 (X, Y)-only co-Z double-add with update – XYCZ-DA

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_4, Y_4) and (X_4', Y_4') s.t. $2\boldsymbol{P}+\boldsymbol{Q} \equiv (X_4 : Y_4 : Z_4)$ and $\boldsymbol{Q} \equiv (X_4' : Y_4' : Z_4)$ for some $Z_4 \in \mathbb{F}_q$
1. $\big((X_3, Y_3), (X_1', Y_1')\big) \leftarrow \mathsf{XYCZ\text{-}ADD}\big((X_1, Y_1), (X_2, Y_2)\big)$
2. $\big((X_4, Y_4), (X_4', Y_4')\big) \leftarrow \mathsf{XYCZ\text{-}ADDC}\big((X_3, Y_3), (X_1', Y_1')\big)$
3. **return** $\big((X_4, Y_4), (X_4', Y_4')\big)$

Algorithm 7 processes an addition and a conjugate addition sequentially, which involves $9\mathbf{M} + 5\mathbf{S}$. We can also deduce from the low level algorithms of Appendix A that the total cost for this double-add algorithm is either of $9\mathbf{M} + 5\mathbf{S} + 18\mathrm{A}$ with 7 field registers or of $9\mathbf{M} + 5\mathbf{S} + 23\mathrm{A}$ with 6 field registers. In [GJM10a], some tricks are proposed to trade field multiplications against field squarings[2]. Only one of the two proposed tricks applies in our context since the other one targets the Z-coordinate computation. It enables trading one field multiplication for one field squaring and 8 field additions/subtractions (see Appendix A for details). The total cost of the obtained *reduced* double-add then becomes either $8\mathbf{M} + 6\mathbf{S} + 31\mathrm{A}$ with 6 field registers or $8\mathbf{M} + 6\mathbf{S} + 26\mathrm{A}$ with 7 field registers. Using this trick hence leads to a reduced double-add faster than the original one if and only if $1\mathbf{M} > 1\mathbf{S} + 8\mathrm{A}$.

2.1 Signed Binary Algorithm

Let k be a scalar with binary expansion $(k_{n-1}, \ldots, k_1, k_0)_2$, where $k_i \in \{0, 1\}$ for every $i < n-1$ and $k_{n-1} = 1$. Without loss of generality, we will assume that k is odd (otherwise it could be replaced by $k+r$ where r is the group order). There exists a unique *full signed expansion* $(\kappa_{n-1}, \ldots, \kappa_1, \kappa_0)$ of k such that $k = \sum_i \kappa_i 2^i$ with $\kappa_i \in \{-1, 1\}$ for every $i < n-2$ and $\kappa_{n-1} = \kappa_{n-2} = 1$. This expansion is easily obtained from the observation that for every $w > 1$, we have $1 = 2^w - \sum_{i=0}^{w-1} 2^i$. It follows that any group of w bits $00\ldots01$ in the binary expansion of k can be replaced by the group of w signed bits $1\bar{1}\bar{1}\ldots\bar{1}$ (where $\bar{1} = -1$). The full signed expansion of k is then obtained by:

$$\kappa_i = \begin{cases} 1 & \text{if } i = n-1 \text{ or } i = n-2, \\ (-1)^{1+k_{i+1}} & \text{otherwise.} \end{cases} \tag{7}$$

We perform the scalar multiplication $\boldsymbol{Q} \leftarrow [k]\boldsymbol{P}$ with a left-to-right binary algorithm using the full signed representation of k, namely we iterate $\boldsymbol{Q} \leftarrow 2\boldsymbol{Q} + \kappa_i \boldsymbol{P}$ from $i = n-2$ down to $i = 0$ (initializing $\boldsymbol{Q}$ to $\boldsymbol{P}$). In every iteration, we use the (X, Y)-only co-Z double-add with update (see Algorithm 7). Note that when $\kappa_i = -1$, $\boldsymbol{P}$ must be inverted prior to the double-add computation and the resulting update is the inverse of $\boldsymbol{P}$. That is, when $\kappa_i = -1$, we compute $\mathsf{XYCZ\text{-}DA}(\boldsymbol{Q}, -\boldsymbol{P}) = (2\boldsymbol{Q} - \boldsymbol{P}, -\boldsymbol{P})$. Then for the next step, we can keep going with $-\boldsymbol{P}$ if κ_{i-1} also equals -1, otherwise we must invert the update of $-\boldsymbol{P}$ to recover the update of $\boldsymbol{P}$. More generally, the sign of $\boldsymbol{P}$ must be switched at the ith iteration if and only if $\kappa_i \neq \kappa_{i+1}$. Namely, we process $R_0 \leftarrow (-1)^b R_0$ where $b = k_{i+1} \oplus k_{i+2}$ and R_0 denotes the register holding $\pm\boldsymbol{P}$.

At the end of the double-add loop, R_1 holds the (X, Y) coordinates of $\boldsymbol{Q} = [k]\boldsymbol{P}$ and R_0 holds that of $(-1)^{1+k_1}\boldsymbol{P}$. Then, we can recover the affine coordinates of $\boldsymbol{P}$ since $\boldsymbol{P}$ and $\boldsymbol{Q}$ share the same Z-coordinate. Indeed, if $\boldsymbol{P} = (x, y) \equiv (X : Y : Z)$, then we have $Z = xY/(Xy)$. This leads to a simple coordinate recovery algorithm which computes $Z^{-1} = Xy/(xY)$ and then multiplies the X and Y coordinates of $\boldsymbol{Q}$ by $(Z^{-1})^2$ and $(Z^{-1})^3$ respectively. A detailed description of the obtained CoordRec algorithm is given in Appendix B. It has a computational cost of $1\mathbf{I} + 6\mathbf{M} + 1\mathbf{S}$.

Eventually, the initial step $\boldsymbol{Q} \leftarrow 2\boldsymbol{P} + \boldsymbol{P}$ has to be treated in a peculiar way as the co-Z double-add formula requires input points which are not equal. Therefore we must start by tripling $\boldsymbol{P}$ to get a pair of co-Z points $(\boldsymbol{P}, \boldsymbol{Q} = [3]\boldsymbol{P})$. For such a purpose, we use an *initial tripling with co-Z update* as proposed in [GJM10a] but without computing the Z-coordinates of $\boldsymbol{P}$ and $\boldsymbol{Q}$. The corresponding

[2] These tricks are based on the usual approach applying $a \times b = \frac{1}{2}(a+b)^2 - a^2 - b^2$ when a^2 and b^2 are already available.

XYCZ-ITPL algorithm is given in Appendix C. Its computational cost is $6\mathbf{M} + 6\mathbf{S} + 17\mathrm{A}$ and it uses 6 field registers. The overall signed binary algorithm with (X, Y)-only co-Z double-add is depicted in the next algorithm.

Algorithm 8 Signed binary algorithm with (X, Y)-only co-Z double-add

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = k_1 = 1$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $(R_1, R_0) \leftarrow$ XYCZ-ITPL($\boldsymbol{P}$)
2. **for** $i = n - 3$ **downto** 0 **do**
3. $\quad b \leftarrow k_{i+1} \oplus k_{i+2}$
4. $\quad R_0 \leftarrow (-1)^b R_0$
5. $\quad (R_1, R_0) \leftarrow$ XYCZ-DA(R_1, R_0)
6. **end for**
7. $R_0 \leftarrow (-1)^{1+k_1} R_0$
8. **return** CoordRec$(\boldsymbol{P}, R_0, R_1)$

Note that for the actual implementation to be regular one must implement Steps 4 and 7 in a regular way. Possible solutions are provided in Appendix D. We shall consider that the cost of such a regular conditional point inversion is 1A.

Algorithm 8 involves $(n-2)$ XYCZ-DA computations, $(n-1)$ conditional point inversions, the initial XYCZ-ITPL computation and the final coordinate recovery computation. The total computational cost and memory requirement of Algorithm 8 is summarized in the following table, depending on the chosen trade-off for the double-add algorithm (the first row corresponds to the standard double-add while the second row corresponds to the reduced double-add). The memory requirement is obtained from the number of field registers used by the double-add algorithm (which is either 6 or 7) plus two field registers to keep track of the affine coordinates of $\boldsymbol{P}$.

Cost per bit ($n-2$ times)	Additional cost	# field registers
$9\mathbf{M} + 5\mathbf{S} + 24\mathrm{A}/19\mathrm{A}$	$1\mathbf{I} + 12\mathbf{M} + 7\mathbf{S} + 18\mathrm{A}$	8/9
$8\mathbf{M} + 6\mathbf{S} + 32\mathrm{A}/27\mathrm{A}$	$1\mathbf{I} + 12\mathbf{M} + 7\mathbf{S} + 18\mathrm{A}$	8/9

2.2 Montgomery Ladder

The Montgomery ladder can also benefit from (X, Y)-only co-Z arithmetic [VD10,GJM10a]. Every loop iteration in the Montgomery ladder (see Algorithm 3) can indeed be rewritten to perform a conjugate addition followed by a regular addition:

$$\begin{cases} R_{1-b} \leftarrow R_{1-b} + R_b \\ R_b \leftarrow 2R_b \end{cases} \quad \Leftrightarrow \quad \begin{cases} (R_{1-b}, R_b) \leftarrow (R_b + R_{1-b}, R_b - R_{1-b}) \\ R_b \leftarrow R_b + R_{1-b} \end{cases}$$

We can then use the (X, Y)-only co-Z addition formulae depicted above to perform such operations. Moreover, we can retrieve the Z-coordinate of the result and hence its affine coordinates [VD10]. Indeed, as the Montgomery ladder satisfies $(R_0, R_1) = ([k]\boldsymbol{P}, [k+1]\boldsymbol{P})$ at the end of the loop, we simply perform a (X, Y)-only co-Z addition of $[k+1]\boldsymbol{P}$ and $-[k]\boldsymbol{P}$ with update of $-[k]\boldsymbol{P}$. This yields a pair of (X, Y)-only co-Z points $(R_0, R_1) = (-[k]\boldsymbol{P}, \boldsymbol{P})$. We can then apply the coordinate recovery algorithm as explained in the previous section to recover the affine coordinates of $[k]\boldsymbol{P}$.

Eventually, the initial step of the Montgomery ladder must be treated in a peculiar way to avoid addition of $\boldsymbol{\mathcal{O}}$ (which is not handled by the co-Z addition formula). Since $k_{n-1} = 1$, the first loop iteration of the Montgomery ladder yields $(R_1, R_0) = (2\boldsymbol{P}, \boldsymbol{P})$. Therefore, we use an *initial doubling with co-Z update* as proposed in [GJM10a], but still without returning the Z-coordinates of $\boldsymbol{P}$ and $2\boldsymbol{P}$. The corresponding XYCZ-IDBL algorithm is given in Appendix C. Its computational cost is $2\mathbf{M} + 4\mathbf{S} + 10\mathrm{A}$ and it uses 6 field registers. The overall (X, Y)-only co-Z Montgomery ladder is depicted in the next algorithm.

Algorithm 9 Montgomery ladder with (X, Y)-only co-Z addition

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $(R_1, R_0) \leftarrow$ XYCZ-IDBL$(\boldsymbol{P})$
2. **for** $i = n - 2$ **downto** 0 **do**
3. $\quad b \leftarrow k_i$
4. $\quad (R_{1-b}, R_b) \leftarrow$ XYCZ-ADDC(R_b, R_{1-b})
5. $\quad (R_b, R_{1-b}) \leftarrow$ XYCZ-ADD(R_{1-b}, R_b)
6. **end for**
7. $(R_1, R_0) \leftarrow$ XYCZ-ADD$(-R_0, R_1)$
8. **return** CoordRec$(\boldsymbol{P}, R_1, -R_0)$

Algorithm 9 involves $(n-1)$ XYCZ-ADDC computations, n XYCZ-ADD computations, the initial XYCZ-IDBL computation and the final coordinate recovery computation. The total computational cost and memory requirement of Algorithm 9 is summarized in the following table depending on the chosen time-memory trade-off for the conjugate addition (see Appendix A). Here again, the memory requirement corresponds to the number of field registers used by the double-add algorithm plus two field registers to keep track of the affine coordinates of $\boldsymbol{P}$.

Cost per bit ($n-1$ times)	Additional cost	# field registers
$9\mathbf{M} + 5\mathbf{S} + 23\mathrm{A}/18\mathrm{A}$	$1\mathbf{I} + 12\mathbf{M} + 7\mathbf{S} + 17\mathrm{A}$	$8/9$

It is suggested in [GJM10a] to merge the regular addition in each iteration with the conjugate addition of the next iteration to obtain a double-add computation whose cost can possibly be reduced depending of the $\mathbf{S}/\mathbf{M}$ and the $\mathrm{A}/\mathbf{M}$ ratios (see discussion above). Namely, one merges the two following steps:

$$R_b \leftarrow R_b + R_{1-b}\ ; \quad (R_{1-d}, R_d) \leftarrow (R_d + R_{1-d}, R_d - R_{1-d})\ ,$$

where $b = k_i$ and $d = k_{i-1}$, which yields:

$$\begin{cases} (R_{1-d}, R_d) \leftarrow (2R_{1-b} + R_b, R_b) & \text{if } d = b, \\ (R_{1-d}, R_d) \leftarrow (2R_{1-b} + R_b, -R_b) & \text{if } d \neq b. \end{cases}$$

This amounts to process $\{(R_{1-d}, R_d) \leftarrow$ XYCZ-DA$(R_{1-b}, R_b)\ ; \quad R_d \leftarrow (-1)^s R_d\}$ where $s = d \oplus b$. The obtained variant of the (X, Y)-only co-Z Montgomery ladder is depicted in the next algorithm.

Algorithm 10 Montgomery ladder with (X, Y)-only co-Z double-add

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. $(R_1, R_0) \leftarrow$ XYCZ-IDBL$(\boldsymbol{P})$
2. $b \leftarrow k_{n-2}$; $(R_{1-b}, R_b) \leftarrow$ XYCZ-ADDC(R_b, R_{1-b})
3. **for** $i = n - 2$ **downto** 1 **do**
4. $\quad b \leftarrow k_i$; $d \leftarrow k_{i-1}$; $s \leftarrow d \oplus b$
5. $\quad (R_{1-d}, R_d) \leftarrow$ XYCZ-DA(R_{1-b}, R_b)
6. $\quad R_d \leftarrow (-1)^s R_d$
7. **end for**
8. $b \leftarrow k_0$; $(R_b, R_{1-b}) \leftarrow$ XYCZ-ADD(R_{1-b}, R_b)
9. $(R_1, R_0) \leftarrow$ XYCZ-ADD$(-R_0, R_1)$
10. **return** CoordRec$(\boldsymbol{P}, R_1, -R_0)$

Algorithm 10 involves $(n-2)$ XYCZ-DA computations, $n-1$ conditional point inversion, 2 XYCZ-ADD computations, 1 XYCZ-ADDC computation, the initial XYCZ-IDBL computation and the final coordinate recovery computation. The following table summarizes the total computational cost and memory requirement of Algorithm 10 depending on the chosen time-memory trade-off for the reduced double-add (see Appendix A).

Cost per bit ($n-2$ times)	Additional cost	# field registers
$8\mathbf{M} + 6\mathbf{S} + 32\mathrm{A}/27\mathrm{A}$	$1\mathbf{I} + 21\mathbf{M} + 12\mathbf{S} + 36\mathrm{A}$	$8/9$

3 Regular Signed Window Algorithms

In the previous section, we have presented fast low-memory binary algorithms for scalar multiplication. We now look at the context where more memory is available allowing the use of window techniques. Window algorithms are similar to binary algorithms but in the former, each loop iteration focuses on a window of w scalar bits rather than on a single bit. Otherwise said, every loop iteration treats a digit of the scalar in radix 2^w.

Let k be some scalar and let $(d_{\ell-1}, \ldots, d_1, d_0)_{2^w}$ denote the expansion of k in radix 2^w, namely $\ell = \lceil n/w \rceil$ where n is the scalar bit-length, and $k = \sum_i d_i 2^{iw}$ with $d_i \in \{0, 1, \ldots 2^w - 1\}$ and $d_{\ell-1} \neq 0$. The principle of the window scalar multiplication is analogous to that of the binary method (see Section 1.2). Defining $T_i = [(d_{\ell-1}, \ldots, d_i)_{2^w}]\boldsymbol{P}$, we get a backward sequence where $T_{\ell-1} = [d_{\ell-1}]\boldsymbol{P}$, $T_0 = [k]\boldsymbol{P}$ and $T_i = [2^w]T_{i+1} + [d_i]\boldsymbol{P}$. The left-to-right window scalar multiplication algorithm consists in evaluating this sequence. One starts by precomputing $[d]\boldsymbol{P}$ for every d, and then compute the T_i sequence from $T_{\ell-1} = [d_{\ell-1}]\boldsymbol{P}$ down to $T_0 = [k]\boldsymbol{P}$. Every iteration then consists in w successive point doublings and one point addition. A right-to-left window algorithm also exists which is based on the equality $[k]\boldsymbol{P} = \sum_i [d_i 2^{iw}]\boldsymbol{P} = \sum_d [d]\big(\sum_{i;d_i=d} [2^{iw}]\boldsymbol{P}\big)$. A loop is processed which applies w successive point doublings in every iteration to compute $[2^{iw}]\boldsymbol{P}$ from $[2^{(i-1)w}]\boldsymbol{P}$, and which adds the result to some accumulator R_{d_i}. At the end of the loop each accumulator R_d contains the sum $\sum_{i;d_i=d}[2^{iw}]\boldsymbol{P}$. The different accumulators are finally aggregated as $\sum_d [d]R_d = [k]\boldsymbol{P}$.

As explained in Section 1.2, subtraction over an elliptic curve is almost as efficient as addition, which allows using a signed representation of the scalar. In the context of window methods, we can use a signed 2^w-radix representation of the scalar $k = (d_{\ell-1}, \ldots, d_1, d_0)_{2^w}$. In such a representation, the digits d_i lie in a basis $\mathcal{B}$ which is different from the simple 2^w-radix basis $\{0, 1, \ldots 2^w - 1\}$ and which includes negative integers. Applying the above principles we can then deduce the signed window scalar multiplication algorithms depicted in Algorithms 11 (left-to-right) and 12 (right-to-left), where $\mathcal{B}^+$ denotes the set $\{|d|; d \in \mathcal{B}\}$.

Algorithm 11 Left-to-right signed window scalar multiplication

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (d_{\ell-1}, \ldots, d_1, d_0)_{2^w} \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. **for all** $d \in \mathcal{B}^+$ **do** $R_d \leftarrow [d]\boldsymbol{P}$
2. $d \leftarrow d_{\ell-1}$; $A \leftarrow R_d$
3. **for** $i = \ell - 2$ **downto** 0 **do**
4. $\quad d \leftarrow |d_i|$
5. $\quad A \leftarrow [2^w]A$
6. $\quad$ **if** $(d_i \geq 0)$ **then** $A \leftarrow A + R_d$
7. $\quad\quad\quad$ **else** $A \leftarrow A - R_d$
8. **end for**
9. **return** A

Algorithm 12 Right-to-left signed window scalar multiplication

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$, $k = (d_{\ell-1}, \ldots, d_1, d_0)_{2^w} \in \mathbb{N}$
Output: $\boldsymbol{Q} = [k]\boldsymbol{P}$
1. **for all** $d \in \mathcal{B}^+$ **do** $R_d \leftarrow \boldsymbol{\mathcal{O}}$
2. $A \leftarrow \boldsymbol{P}$
3. **for** $i = 0$ **to** $\ell - 2$ **do**
4. $\quad d \leftarrow |d_i|$
5. $\quad$ **if** $(d_i \geq 0)$ **then** $R_d \leftarrow R_d + A$
6. $\quad\quad\quad$ **else** $R_d \leftarrow R_d - A$
7. $\quad A \leftarrow [2^w]A$
8. **end for**
9. **return** $\sum_{d \in \mathcal{B}^+} [d]R_d$

Window algorithms with $w > 1$ involve less point operations than their binary counterparts. While a typical regular binary algorithm involves roughly n point doublings and n point additions (see Section 1.2), the utilization of a w-bit window lowers the number of point additions to n/w (neglecting the precomputation and aggregation costs). On the other hand, window algorithms require more memory as they involve $(m + 1)$ point registers where $m = \#\mathcal{B}^+$. We hence see that their memory consumption depends of the choice of $\mathcal{B}$, namely on the recoding of the scalar (this issue is discussed hereafter).

Regular Implementation. In order to implement Algorithms 11 and 12 in a regular way, one must avoid the conditional statement in the loop. A simple solution to avoid this statement is to replace Steps 6 and

7 of Algorithm 11 and Steps 5 and 6 of Algorithm 12 by:

$$\begin{cases} s \leftarrow \mathsf{sign}(d_i) \\ R_d \leftarrow (-1)^s R_d \\ A \leftarrow A + R_d \\ R_d \leftarrow (-1)^s R_d \end{cases} \quad \text{and} \quad \begin{cases} s \leftarrow \mathsf{sign}(d_i) \\ A \leftarrow (-1)^s A \\ R_d \leftarrow R_d + A \\ A \leftarrow (-1)^s A \end{cases} \quad \text{where} \quad \mathsf{sign} : x \mapsto \begin{cases} 1 & \text{if } x < 0, \\ 0 & \text{otherwise.} \end{cases}$$

The regular implementation of a conditional point inversion $\boldsymbol{P} \leftarrow (-1)^s \boldsymbol{P}$ is addressed in Appendix D. We expect a computational cost of 1A. Therefore, using the above implementation for the conditional statement shall induce a cost of 2A per loop iteration (in addition to the w point doublings and to the point addition). It is actually possible to do better. One can indeed save 1A by merging the second step $\boldsymbol{P} \leftarrow (-1)^s \boldsymbol{P}$ of an iteration to the first one in the next iteration. For the right-to-left implementation, we simply get $\{s \leftarrow \mathsf{sign}(d_i) \oplus \mathsf{sign}(d_{i-1}); A \leftarrow (-1)^s A\}$, and we can remove the second step $A \leftarrow (-1)^s A$ (a special treatment of the first iteration is also necessary). For the left-to-right implementation, things are slightly more complicated as two consecutive iterations are likely to handle different precomputed points R_d. We shall then use a binary variable s_d to keep track of the current sign of R_d for every d. Each iteration shall then process $\{s \leftarrow \mathsf{sign}(d); R_d \leftarrow (-1)^{s \oplus s_d} R_d; s_d \leftarrow s\}$.

The right-to-left algorithm is still not regular in its current form due to the initialization of the accumulators R_d to $\boldsymbol{\mathcal{O}}$. This initialization implies that the first addition (or subtraction) of A to each R_d must be treated in a peculiar way. Such a special treatment break the regularity of the algorithm, which may leak the indices of the first occurrences of each digit in the scalar to an SPA adversary. To avoid such a weakness, one can initialize each R_d to $\boldsymbol{P}$. Doing so, the aggregation step shall yield the point $[k]\boldsymbol{P} + \sum_{d \in \mathcal{B}^+} [d]\boldsymbol{P}$. A simple trick to correct this value consists in subtracting $\sum_{d \in \mathcal{B}^+} d$ to the scalar prior the loop [Joy09]. This way, the correct result $[k]\boldsymbol{P}$ is well obtained after aggregation.

Another issue arrises for the regular implementation of the left-to-right algorithm when the basis $\mathcal{B}$ includes the zero digit (which is for instance the case with $\mathcal{B} = \{0, 1, \ldots, 2^w - 1\}$). Indeed, if $d = 0$ then $R_d = \boldsymbol{\mathcal{O}}$ and the addition $A \leftarrow A + R_d$ needs a particular treatment. A possibility to avoid this issue is to perform a dummy addition whenever $d = 0$. This is actually what naturally happens with the right-to-left algorithm in which a zero digit implies an addition of A to the accumulator R_0. This addition is actually dummy since the final value of R_0 does not affect the aggregation result. Dummy operations are usually avoided because they render the implementation subject to *safe-error attacks* [YKLM02]. Indeed, by injecting a fault in the addition computation, one could check whether it is dummy or not (depending on the correctness of the result), and hence deduce which digit of the scalar are zero and which are not. Note that such an attack scenario does not apply in the presence of a randomized scalar which is often required for DPA-resistance [Cor99,CJ03]. It is however preferred in general to avoid dummy operations, which constrains to choose a basis $\mathcal{B}$ such that $0 \notin \mathcal{B}$.

Scalar Recoding. The main purpose of signed recoding of the scalar for window methods is to lower the memory requirement. As discussed above, window algorithms require $(m+1)$ point registers where $m = \#\mathcal{B}^+$. It is not hard to see that the straightforward basis $\mathcal{B} = \{0, 1, \ldots, 2^w - 1\}$ implies $m = 2^w$. As suggested by Möller in [Möl01,Möl02], this can be improved by using the signed basis $\mathcal{B} = \{-2^{w-1}, -2^{w-1} + 1, \ldots, 2^{w-1} - 1\}$. The recoding of the scalar into this basis can be simply done by scanning its digits from the right to the left. Indeed if $d_i \geq 2^{w-1}$ one can replace it by $d_i - 2^w$ and add 1 to the next digit d_{i+1} (with possibly a carry propagation to the next digits). To avoid the zero digit, it is also possible to replace it by -2^w in the basis. The scalar recoding then further consists in replacing every $d_i = 0$ by -2^w and incrementing d_{i+1} (still with a possible carry propagation). Note that the new scalar representation may be of length $\ell + 1$ (if $d_{\ell-1} \geq 2^{w-1}$ or because of some carry propagation). This recoding satisfies $m = 2^{w-1} + 1$ and the resulting window algorithm involves $2^{w-1} + 2$ point registers which is almost twice less than the $2^w + 1$ point registers required with the straightforward basis.

A better solution is actually possible which was initially proposed by Okeya and Takagi in [OT03]. It consists in using the basis of odd digits $\mathcal{B} = \{\pm 1, \pm 3, \ldots, \pm 2^w - 1\}$. This recoding assumes an odd scalar, which is not a loss of generality as an even k could be replaced by $k + r$ where r is the (usually prime) group order. The recoding works in a right-to-left direction from d_1 to $d_{\ell-1}$ (d_0 is already odd by assumption) and it consists, at step i, in incrementing d_i (with a possible carry propagation) and replacing d_{i-1} by $d_{i-1} - 2^w$ whenever d_i is even. Note that since the recoding proceeds from the least to the most significant digits, d_{i-1} is always odd and greater than 0 at step i which ensures that $(d_{i-1} - 2^w) \in$

$\mathcal{B}$ in case d_{i-1} has to be updated. A regular version of this recoding has been proposed by Joye and Tunstall in [JT09] which takes some scalar k and computes its 2^w-radix representation $(d_\ell, \ldots, d_1, d_0)_{2^w}$ with digits lying in the odd basis. Their recoding algorithm proceeds as follows:

1. $i \leftarrow 0$
2. **while** $(k > 2^w)$ **do**
3. $\quad d_i \leftarrow (k \bmod 2^{w+1}) - 2^w$
4. $\quad k \leftarrow (k - d_i)/2^w$
5. $\quad i \leftarrow i + 1$
6. **end while**

The odd basis satisfies $m = 2^{w-1}$ and the resulting signed window algorithm requires $2^{w-1}+1$ point registers (which is one register less than in the Möller proposal). It was moreover shown in [OT03] that $m = 2^{w-1}$ is the best that can be reached while ensuring that any (odd) scalar can be represented in the underlying basis. In the following we shall then assume that the odd basis is chosen for signed window algorithms.

Point Arithmetic. Window algorithms mostly involve point doublings so the best choice to get efficient point arithmetic is Jacobian coordinates. Using standard doubling and addition formulae (see (3) and (4)) as proposed in [OT03], one shall then obtain a cost per scalar bit of $4\mathbf{M}+6\mathbf{S}+11\mathrm{A}+\frac{1}{w}(12\mathbf{M}+4\mathbf{S}+7\mathrm{A})$ (without precomputation and aggregation). When $a = -3$, this cost can be reduced to $4\mathbf{M} + 4\mathbf{S} + 12\mathrm{A} + \frac{1}{w}(12\mathbf{M} + 4\mathbf{S} + 7\mathrm{A})$ (see Section 1.1). Concerning the memory usage, each point register requires 3 field registers and the Jacobian addition requires 4 additional field registers[3], which makes a total memory consumption of $3 \cdot 2^{w-1} + 7$ field registers. Nevertheless, it is possible to do better depending on the direction of the scalar scanning.

For the right-to-left algorithm, the case $a \neq -3$ can be optimized by using the repeated doubling formula, *i.e.* by using modified Jacobian coordinates for the point register A (see Section 1.1). The obtained cost then drops to $4\mathbf{M} + 4\mathbf{S} + 11\mathrm{A} + \frac{1}{w}(12\mathbf{M} + 4\mathbf{S} + 7\mathrm{A})$ per scalar bit, but one additional field register is needed. One could actually use this strategy for $a = -3$ also, which would save $1\mathrm{A}$ per scalar bit (and would only involve one software code for both cases).

The case of the left-to-right algorithm is slightly different as the same register (namely A) is doubled and receives the results of point additions. When $a \neq -3$, the accumulator A shall be represented in modified Jacobian coordinates to favor repeated doublings. As the addition of a R_d to A does not yield a point represented in modified Jacobian coordinates, the first doubling is always a standard Jacobian doubling. The obtained cost for the w consecutive doublings is therefore of $4w\mathbf{M}+(4w+2)\mathbf{S}+(11w+1)\mathrm{A}$. When $a = -3$, one shall just apply w times the optimized Jacobian doubling getting a cost of $w(4\mathbf{M}+4\mathbf{S}+12\mathrm{A})$. For the point additions, one can use the standard Jacobian addition (with cost $12\mathbf{M}+4\mathbf{S}+7\mathrm{A}$) or the mixed-Jacobian-affine addition with a reduced cost of $8\mathbf{M}+3\mathbf{S}+7\mathrm{A}$ (see Section 1.1). However, the latter choice requires to translate the precomputed values $R_d = [d]\boldsymbol{P}$ into affine coordinates before the loop, and such translation may induce an important overhead in the precomputation step (see hereafter). Note on the other hand that when affine coordinates are used for the precomputed values the total memory consumption falls to $2^w + 4$ field registers.

Precomputation and Aggregation. For both signed window algorithms some additional computation is required. For the left-to-right algorithm, one must precompute the values $R_d = [d]\boldsymbol{P}$ for every $d \in \mathcal{B}^+$, while for the right-to-left algorithm the aggregation $\sum_{d\in\mathcal{B}^+}[d]R_d$ must be computed after the loop.

The precomputation in the odd basis aims at producing points $[3]\boldsymbol{P}$, $[5]\boldsymbol{P}$, …, $[2^w - 1]\boldsymbol{P}$. Such a computation can be efficiently processed based on co-Z arithmetic [LM08]. One first compute a pair of co-Z points $(\boldsymbol{P}, [2]\boldsymbol{P})$ based on the doubling with co-Z update. A detailed algorithm is given in Appendix C (Algorithm 18), which costs $2\mathbf{M}+4\mathbf{S}+10\mathrm{A}$. The only difference is that the Z-coordinate must be returned but this does not imply any overhead as it is already computed in Algorithm 18. Then $[3]\boldsymbol{P}$ is computed by a co-Z addition between $\boldsymbol{P}$ and $[2]\boldsymbol{P}$ with co-Z update of $[2]\boldsymbol{P}$, afterward $[5]\boldsymbol{P}$ is computed by a co-Z addition between $[3]\boldsymbol{P}$ and $[2]\boldsymbol{P}$ with co-Z update of $[2]\boldsymbol{P}$, and so on until $[2^w - 1]\boldsymbol{P}$. On the whole, the precomputation involves $(2^{w-1}-1)$ co-Z additions that each costs $5\mathbf{M}+2\mathbf{S}+7\mathrm{A}$. The total cost

[3] This is obtained from the memory consumption of the Jacobian addition (7 field registers) by deducting 3 field registers for the result (which is one of the point register already counted).

of the precomputation step is then of $2^{w-1}(5\mathbf{M}+2\mathbf{S}+7\mathrm{A})-3\mathbf{M}+2\mathbf{S}+3\mathrm{A}$. When the precomputed values must be translated to affine coordinates, additional computation must be performed. Let (X_i, Y_i, Z_i) denotes the Jacobian coordinates of the point $[2i-1]\boldsymbol{P}$ obtained from the above computation and let (X'_i, Y'_i, Z_i) denote the corresponding co-Z Jacobian coordinates of $[2]\boldsymbol{P}$. The co-Z addition formula (see 5) implies $Z_{i+1} = (X_i - X'_i)Z_i$ and hence $Z_n = \left(\prod_i (X_i - X'_i)\right) Z_1$ where $n = 2^{w-1}$. Therefore it is possible to design some affine coordinate recovery algorithm which starts by inverting Z_n and then recover the affine coordinates of the different points from $[2^w - 1]\boldsymbol{P}$ down to $[3]\boldsymbol{P}$. Such a scheme has been proposed in [LM08] (Scheme 2) which costs $1\mathbf{I} + 4(2^{w-1} - 2)\mathbf{M} + 3\mathbf{M} + 1\mathbf{S}$ and which requires 5 field registers in addition to those for the R_b. Note that this memory consumption does not overcome the bottleneck considering the 3 field registers necessary for A and the temporary registers for the point additions.

The aggregation step of the right-to-left window algorithm in the odd basis can be efficiently computed using a left-to-right exponentiation principle. More precisely, one performs the following steps:

1. $d \leftarrow 2^w - 1$; $A \leftarrow R_d$
2. **while** $(d > 0)$ **do**
3. $\quad d \leftarrow d - 2$
4. $\quad A \leftarrow [2]A + R_d$
5. **end while**

At the end of the above process, the accumulator A holds the aggregated value $\sum_{d \in \mathcal{B}^+} [d]R_d$. This computation involves $(2^{w-1} - 1)$ point doublings and the same number of point additions. In Jacobian coordinates, the total cost is then of $(2^{w-1} - 1)(16\mathbf{M} + 10\mathbf{S} + 18\mathrm{A})$ in the general case, and $(2^{w-1} - 1)(16\mathbf{M} + 8\mathbf{S} + 19\mathrm{A})$ when $a = -3$.

Eventually, for both algorithms the affine coordinates (x, y) of the result must be recovered from the Jacobian coordinates (X, Y, Z). Such a recovery simply consists in computing $(x, y) = (X/Z^2, Y/Z^3)$ which costs $1\mathbf{I} + 3\mathbf{M} + 1\mathbf{S}$.

4 Security against Zero-Value Attacks

In the previous sections, we have described regular algorithms for elliptic curve scalar multiplication. As explained in Section 1.2, regularity is required to thwart SPA attacks distinguishing scalar bits from different operation patterns in a leakage trace. However, regularity may not suffice to prevent some refine SPA attacks called *zero-value attacks*.

In [Gou03], Goubin explained how to chose a point $\boldsymbol{P}$ that makes appear an intermediate point with a zero-value coordinate in the computation of $[k]\boldsymbol{P}$ for certain values of k. The occurrence of such a zero-coordinate can usually be detected from a leakage trace, which allows an SPA observer to discriminate the value of k. This attack was extended in [AT03] where the authors also exploit zero-value occurrence inside the point addition computations. A simple countermeasure to thwart this kind of attacks is to randomize the scalar [Cor99,CJ03] (whereas other kinds of randomization may fail [Gou03]).

Another kind of zero-value SPA attacks was recently introduced by Courrège *et al.* in [CFR10]. This attack targets modular exponentiation but it straightforwardly transposes to an attack against certain scalar multiplication algorithms. The attack assumes that a side-channel adversary can identify whenever a multiplication by zero (or by some low Hamming weight digit) is performed by the hardware multiplier of the target device. A multiplication over $\mathbb{F}_q$ is usually composed of many $(n \times m)$-bit multiplications performed by some hardware multiplier. An adversary such as considered in [CFR10] could then learn when an operand of some field multiplication has a n-bit (resp. m-bit) digit to zero (or with low Hamming weight). Such an observation can reveal secret information. For instance the left-to-right window algorithm presented in the previous section handles the value $[d]\boldsymbol{P}$ in the ith iteration if and only if $d_i = d$. An adversary could ask for the scalar multiplication of some point $\boldsymbol{P}$ such that $[d]\boldsymbol{P}$ has one or several zero digit(s) in some of its coordinates. Looking at the resulting leakage trace, he could then identify all the digits of the scalar which equal d. Moreover, such an attack may also work when the adversary cannot choose the input point, and even in the presence of randomization (of either the point coordinates or the scalar or both). If the bit-size of the hardware multiplier is small (*e.g.* between 8 and 32) which is usual, the occurrence probability of a zero digit in some (possibly randomized) coordinate of some precomputed point $[d]\boldsymbol{P}$ is not weak. Then the partial information leaked about the scalar (even randomized) over several computations could probably enable a complete break of the system based on a lattice attack (see for instance [NNTW05]). In order to prevent such a vulnerability we recommend

to avoid scalar multiplication algorithms using fixed point with fixed coordinates such as left-to-right (window) algorithms. Note that the binary algorithms presented in Section 2 do not suffer this weakness. Even the signed left-to-right algorithm is secure as the coordinate of the fixed point $\boldsymbol{P}$ change in each iteration thanks to co-Z updates.

5 Comparison

In this section we compare the performances of different scalar multiplication algorithms. We consider previous regular binary algorithms with fast point arithmetic (see Section 1.2), including the double-and-add-always algorithms using mixed-Jacobian-affine coordinates [Cor99], the Montgomery ladder using (X, Z)-only projective coordinates [IMT02], the binary ladders based on co-Z Jacobian coordinates[4] [GJM10a], and the Montgomery ladder using (X, Z)-only co-Z projective coordinates [HJS11]. We also consider the binary algorithms based on (X, Y)-only co-Z coordinates presented in Section 2 with their different trade-offs, as well as the signed window algorithms detailed in Section 3. The performances of the different algorithms are provided in Table 1. For every algorithm, we give the cost per bit (which is to count $n-1$ times where n is the scalar bit-length), the additional cost, and the number of required field registers[5].

The additional cost results from the different steps outside the main loop including precomputation and postcomputation (*e.g.* affine coordinates recovery, aggregation). Some negative values sometimes appear for algorithms in which the main loop is iterated only $n-2$ times. For a fair comparison, we then subtract the cost of one loop iteration to the additional cost.

The results in Table 1 give rise to the following remarks:

- Left-to-right window algorithms using mixed coordinates should provide the best performances in general, especially when a equals -3 ($\frac{2}{w}\mathbf{S}$ are saved) and when the $\mathbf{I}/\mathbf{M}$ ratio is low (left-to-right algorithms with mixed coordinates require two inversions whereas other algorithms only one). However, as explained in Section 4, left-to-right algorithms are vulnerable to zero-value attacks. If the latter are identified as a potential threat, one shall avoid such algorithms and favour other algorithms discussed in this paper.
- For binary algorithms, those based on co-Z (X, Y)-coordinates are faster than any other algorithm for common $\mathbf{S}/\mathbf{M}$ and $\mathrm{A}/\mathbf{M}$ ratios. Assuming $1\mathbf{S} \approx 1\mathbf{M}$, neglecting the field additions, and assuming a to be small (*i.e.* $\mathrm{A} \approx 0$ and $\mathbf{M_a} \approx 0$), these algorithms are roughly 12.5% faster than the co-Z binary ladders from [GJM10a], 7.7% faster than the (X, Z)-only Montgomery ladders from [IMT02] and [HJS11] (without counting the final inversion). Note however that when addition are especially expensive (and more precisely when $6\mathrm{A} > 1\mathbf{M}$), the co-$Z$ (X, Y)-coordinates Montgomery ladder proposed in [HJS11] may offer better performances.
- Among the binary algorithms based on co-Z (X, Y)-coordinates (see Section 2), the best choice between the signed binary algorithm and the Montgomery ladder (already proposed in [VD10]) shall depend of several criteria. When the $\mathbf{S}/\mathbf{M}$ ratio is low (*e.g.* lower than or equal to 0.8) and when additions are cheap, the reduced algorithms shall be preferred, and the signed binary algorithm shall always be faster than the Montgomery ladder (same cost per scalar bit, lower additional cost). On the other hand if $\mathbf{S} \approx \mathbf{M}$ or if additions are not cheap, the non-reduced algorithms shall be prefer. In that case, the Montgomery ladder requires one addition less per scalar bit, but it has a more important additional cost ($8\mathbf{M} + 5\mathbf{S} + 23\mathrm{A}/18\mathrm{A}$ more than the signed binary algorithm). Therefore the best trade-off depends on the $\mathbf{S}/\mathbf{M}$ and $\mathrm{A}/\mathbf{M}$ ratios as well as on the scalar size.
- Right-to-left window algorithms require significantly more memory than the binary algorithms based co-Z (X, Y)-coordinates to reach equivalent speed. For $w \geq 3$, right-to-left window algorithms indeed require more than twice the memory of binary algorithms. Moreover, for a 160-bit scalar, assuming $1\mathbf{S} \approx 1\mathbf{M}$ and neglecting the field additions, our signed binary algorithm is 13.3% faster, 1.6% slower, and 6.2% slower than the right-to-left window algorithm with $w = 2$, $w = 3$, and $w = 4$ respectively (without counting the final inversion). Higher security levels slightly favor window algorithms as the precomputation cost becomes less important in the overall computation.

[4] For binary ladders with co-Z arithmetic from [GJM10a], we only give the best performances which are obtained for the Joye double-and-add algorithm (the Montgomery ladder requiring slightly more precomputation).

[5] For [IMT02] and [HJS11], the $(+1)$ in the number of field registers means $+0$ when a is small and $+1$ (to store a) when a is large.

Table 1. Performances of regular scalar multiplication algorithms.

Method	Cost per scalar bit	Additional cost	# field regs.
Binary algorithms			
DA-always with mixed coord. ($a=-3$)	$12\mathbf{M}+7\mathbf{S}+19\mathrm{A}$	$1\mathbf{I}+3\mathbf{M}+1\mathbf{S}$	12
Mont. ladder in (X,Z)-coord. [IMT02]	$11\mathbf{M}+4\mathbf{S}+2\mathbf{M_a}+18\mathrm{A}$	$1\mathbf{I}+21\mathbf{M}+7\mathbf{S}+25\mathrm{A}$	$9(+1)$
Mont. lad. in co-Z (X,Z)-coord. [HJS11]	$10\mathbf{M}+5\mathbf{S}+13\mathrm{A}$	$1\mathbf{I}+23\mathbf{M}+1\mathbf{M_a}+8\mathbf{S}+21\mathrm{A}$	$10(+1)$
Binary ladder in co-Z coord. [GJM10a]	$11\mathbf{M}+5\mathbf{S}+23\mathrm{A}$	$1\mathbf{I}-1\mathbf{M}+2\mathbf{S}-6\mathrm{A}$	7
$\Rightarrow$ with reduced double-add	$9\mathbf{M}+7\mathbf{S}+27\mathrm{A}$	$1\mathbf{I}+1\mathbf{M}-10\mathrm{A}$	8
Signed binary algorithm (Algo. 8)	$9\mathbf{M}+5\mathbf{S}+24\mathrm{A}/19\mathrm{A}$	$1\mathbf{I}+3\mathbf{M}+2\mathbf{S}-6\mathrm{A}/1\mathrm{A}$	$8/9$
$\Rightarrow$ with reduced double-add	$8\mathbf{M}+6\mathbf{S}+32\mathrm{A}/27\mathrm{A}$	$1\mathbf{I}+4\mathbf{M}+1\mathbf{S}-14\mathrm{A}/9\mathrm{A}$	$8/9$
Montgomery ladder v1 (Algo. 9)	$9\mathbf{M}+5\mathbf{S}+23\mathrm{A}/18\mathrm{A}$	$1\mathbf{I}+12\mathbf{M}+7\mathbf{S}+17\mathrm{A}$	$8/9$
Montgomery ladder v2 (Algo. 10)	$8\mathbf{M}+6\mathbf{S}+32\mathrm{A}/27\mathrm{A}$	$1\mathbf{I}+13\mathbf{M}+6\mathbf{S}+4\mathrm{A}/9\mathrm{A}$	$8/9$
Right-to-left window algorithms			
Jacobian coordinates ($a=-3$)	$4\mathbf{M}+4\mathbf{S}+12\mathrm{A}$ $+\frac{1}{w}(12\mathbf{M}+4\mathbf{S}+8\mathrm{A})$	$1\mathbf{I}+3\mathbf{M}+1\mathbf{S}$ $(2^{w-1}-1)(16\mathbf{M}+8\mathbf{S}+19\mathrm{A})$	$3\cdot 2^{w-1}+7$
$w=2$	$10\mathbf{M}+6\mathbf{S}+16\mathrm{A}$	$1\mathbf{I}+19\mathbf{M}+9\mathbf{S}+19\mathrm{A}$	13
$w=3$	$8\mathbf{M}+5.3\mathbf{S}+14.7\mathrm{A}$	$1\mathbf{I}+51\mathbf{M}+25\mathbf{S}+57\mathrm{A}$	19
$w=4$	$7\mathbf{M}+5\mathbf{S}+14\mathrm{A}$	$1\mathbf{I}+131\mathbf{M}+65\mathbf{S}+162\mathrm{A}$	28
Jacobian coordinates ($a\neq -3$)	$4\mathbf{M}+4\mathbf{S}+11\mathrm{A}$ $+\frac{1}{w}(12\mathbf{M}+4\mathbf{S}+8\mathrm{A})$	$1\mathbf{I}+3\mathbf{M}+1\mathbf{S}$ $(2^{w-1}-1)(16\mathbf{M}+10\mathbf{S}+18\mathrm{A})$	$3\cdot 2^{w-1}+8$
$w=2$	$10\mathbf{M}+6\mathbf{S}+17\mathrm{A}$	$1\mathbf{I}+19\mathbf{M}+11\mathbf{S}+18\mathrm{A}$	14
$w=3$	$8\mathbf{M}+5.3\mathbf{S}+15.7\mathrm{A}$	$1\mathbf{I}+51\mathbf{M}+31\mathbf{S}+54\mathrm{A}$	20
$w=4$	$7\mathbf{M}+5\mathbf{S}+15\mathrm{A}$	$1\mathbf{I}+131\mathbf{M}+81\mathbf{S}+154\mathrm{A}$	29
Left-to-right window algorithms			
Jacobian coordinates ($a=-3$)	$4\mathbf{M}+4\mathbf{S}+12\mathrm{A}$ $+\frac{1}{w}(12\mathbf{M}+4\mathbf{S}+8\mathrm{A})$	$1\mathbf{I}+3\mathbf{S}+3\mathrm{A}$ $+2^{w-1}(5\mathbf{M}+2\mathbf{S}+7\mathrm{A})$	$3\cdot 2^{w-1}+7$
$w=2$	$10\mathbf{M}+6\mathbf{S}+16\mathrm{A}$	$1\mathbf{I}+10\mathbf{M}+7\mathbf{S}+17\mathrm{A}$	13
$w=3$	$8\mathbf{M}+5.3\mathbf{S}+14.7\mathrm{A}$	$1\mathbf{I}+20\mathbf{M}+11\mathbf{S}+31\mathrm{A}$	19
$w=4$	$7\mathbf{M}+5\mathbf{S}+14\mathrm{A}$	$1\mathbf{I}+40\mathbf{M}+19\mathbf{S}+59\mathrm{A}$	28
Jacobian coordinates ($a\neq -3$)	$4\mathbf{M}+4\mathbf{S}+11\mathrm{A}$ $+\frac{1}{w}(12\mathbf{M}+6\mathbf{S}+9\mathrm{A})$	$1\mathbf{I}+3\mathbf{S}+3\mathrm{A}$ $+2^{w-1}(5\mathbf{M}+2\mathbf{S}+7\mathrm{A})$	$3\cdot 2^{w-1}+7$
$w=2$	$10\mathbf{M}+7\mathbf{S}+15.5\mathrm{A}$	$1\mathbf{I}+10\mathbf{M}+7\mathbf{S}+17\mathrm{A}$	13
$w=3$	$8\mathbf{M}+6\mathbf{S}+14\mathrm{A}$	$1\mathbf{I}+20\mathbf{M}+11\mathbf{S}+31\mathrm{A}$	19
$w=4$	$7\mathbf{M}+7.5\mathbf{S}+13.3\mathrm{A}$	$1\mathbf{I}+40\mathbf{M}+19\mathbf{S}+59\mathrm{A}$	28
Mixed coordinates ($a=-3$)	$4\mathbf{M}+4\mathbf{S}+12\mathrm{A}$ $+\frac{1}{w}(8\mathbf{M}+3\mathbf{S}+8\mathrm{A})$	$2\mathbf{I}+1\mathbf{M}+4\mathbf{S}+3\mathrm{A}$ $+2^{w-1}(9\mathbf{M}+2\mathbf{S}+7\mathrm{A})$	2^w+4
$w=2$	$8\mathbf{M}+5.5\mathbf{S}+16\mathrm{A}$	$2\mathbf{I}+19\mathbf{M}+8\mathbf{S}+17\mathrm{A}$	8
$w=3$	$6.7\mathbf{M}+5\mathbf{S}+14.7\mathrm{A}$	$2\mathbf{I}+37\mathbf{M}+12\mathbf{S}+31\mathrm{A}$	12
$w=4$	$6\mathbf{M}+4.8\mathbf{S}+14\mathrm{A}$	$2\mathbf{I}+73\mathbf{M}+22\mathbf{S}+66\mathrm{A}$	20
Mixed coordinates ($a\neq -3$)	$4\mathbf{M}+4\mathbf{S}+11\mathrm{A}$ $+\frac{1}{w}(8\mathbf{M}+5\mathbf{S}+9\mathrm{A})$	$2\mathbf{I}+1\mathbf{M}+4\mathbf{S}+3\mathrm{A}$ $+2^{w-1}(9\mathbf{M}+2\mathbf{S}+7\mathrm{A})$	2^w+4
$w=2$	$8\mathbf{M}+6.5\mathbf{S}+15.5\mathrm{A}$	$2\mathbf{I}+19\mathbf{M}+8\mathbf{S}+17\mathrm{A}$	8
$w=3$	$6.7\mathbf{M}+5.7\mathbf{S}+14\mathrm{A}$	$2\mathbf{I}+37\mathbf{M}+12\mathbf{S}+31\mathrm{A}$	12
$w=4$	$6\mathbf{M}+5.3\mathbf{S}+13.3\mathrm{A}$	$2\mathbf{I}+73\mathbf{M}+22\mathbf{S}+66\mathrm{A}$	20

For a 512-bit scalar, we get that our algorithm is 13% faster, 2.9% slower, and 9.8% slower than the right-to-left window algorithm with $w=2$, $w=3$, and $w=4$ respectively.

– Eventually, let us mention that when the target application is not constrained to use general elliptic curves (*i.e.* all curves satisfying (1)), better algorithms exist over elliptic curves with special forms. For instance, the Montgomery ladder over Montgomery curves [Mon87] only requires $6\mathbf{M}+4\mathbf{S}$ (and a few additions) per scalar bit. Another example is the Edwards form over which the point doubling only requires $3\mathbf{M}+4\mathbf{S}$, the point addition $10\mathbf{M}+1\mathbf{S}$, and the mixed point addition $9\mathbf{M}+1\mathbf{S}$, yielding faster window algorithms [BL07]. See [BL] for further examples.

References

[AARR02] D. Agrawal, B. Archambeault, J.R. Rao, and P. Rohatgi. The EM Side-Channel(s). In Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, 2nd International Workshop – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 29–45. Springer, 2002.

[AT03] Toru Akishita and Tsuyoshi Takagi. Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In Colin Boyd and Wenbo Mao, editors, *Information Security, 6th International Conference – ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2003.

[BJ02] Eric Brier and Marc Joye. Weierstraß elliptic curves and side-channel attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography – PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer, 2002.

[BL] Daniel J. Bernstein and Tanja Lange. Explicit-Formulas Database. Available at http://hyperelliptic.org/EFD/index.html.

[BL07] Daniel J. Bernstein and Tanja Lange. Faster Addition and Doubling on Elliptic Curves. In Kaoru Kurosawa, editor, *Advances in Cryptology, 13th International Conference on the Theory and Application of Cryptology and Information Security – ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.

[CFR10] Jean-Christophe Courrège, Benoit Feix, and Mylène Roussellet. Simple power analysis on exponentiation revisited. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th International Conference – CARDIS 2010*, volume 6035 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2010.

[CJ03] Mathieu Ciet and Marc Joye. (Virtually) Free Randomization Techniques for Elliptic Curve Cryptography. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Information and Communications Security, 5th International Conference – ICICS 2003*, volume 2836 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2003.

[CMCJ04] Benoît Chevallier-Mames, Mathieu Ciet, and Marc Joye. Low-Cost Solutions for Preventing Simple Side-Channel Analysis: Side-Channel Atomicity. *IEEE Transactions on Computers*, 53(6):760–768, 2004.

[CMO98] Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient Elliptic Curve Exponentiation Using Mixed Coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Advances in Cryptology, International Conference on the Theory and Application of Cryptology and Information Security – ASIACRYPT'98*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 1998.

[Cor99] Jean-Sébastien Coron. Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop – CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.

[GJM10a] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves - (Extended Abstract). In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, 12th International Workshop – CHES 2010*, volume 6225 of *Lecture Notes in Computer Science*, pages 65–79. Springer, 2010.

[GJM10b] Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z Addition Formulae and Binary Ladders on Elliptic Curves. Cryptology ePrint Archive, Report 2010/309, 2010. http://eprint.iacr.org/.

[Gou03] Louis Goubin. A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In Yvo Desmedt, editor, *Public Key Cryptography, 6th International Workshop – PKC 2003*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003.

[GV10] Christophe Giraud and Vincent Verneuil. Atomicity improvement for elliptic curve scalar multiplication. In Dieter Gollmann, Jean-Louis Lanet, and Julien Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application, 9th International Conference – CARDIS 2010*, volume 6035 of *Lecture Notes in Computer Science*, pages 80–101. Springer, 2010.

[HJS11] Michael Hutter, Marc Joye, and Yannick Sierra. Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation. To appear in *AFRICACRYPT 2011*, 2011.

[HMV03] Darrel Hankerson, Alfred J. Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer-Verlag, 2003.

[IMT02] Tetsuya Izu, Bodo Möller, and Tsuyoshi Takagi. Improved elliptic curve multiplication methods resistant against side channel attacks. In Alfred Menezes and Palash Sarkar, editors, *Progress in Cryptology, Third International Conference on Cryptology in India – INDOCRYPT 2002*, volume 2551 of *Lecture Notes in Computer Science*, pages 296–313. Springer, 2002.

[IT02] Tetsuya Izu and Tsuyoshi Takagi. A Fast Parallel Elliptic Curve Multiplication Resistant against Side Channel Attacks. In David Naccache and Pascal Paillier, editors, *Public Key Cryptography – PKC 2002*, volume 2274 of *Lecture Notes in Computer Science*, pages 280–296. Springer, 2002.

[ITT+99] Kouichi Itoh, Masahiko Takenaka, Naoya Torii, Syouji Temma, and Yasushi Kurihara. Fast Implementation of Public-Key Cryptography ona DSP TMS320C6201. In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop – CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 61–72. Springer, 1999.

[Joy07] Marc Joye. Highly Regular Right-to-Left Algorithms for Scalar Multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems, 9th International Workshop – CHES 2007*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2007.

[Joy09] Marc Joye. Highly Regular -Ary Powering Ladders. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop – SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2009.

[JT01] Marc Joye and Christophe Tymen. Protections against Differential Analysis for Elliptic Curve Cryptography. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, Third International Workshop - CHES 2001*, volume 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2001.

[JT09] Marc Joye and Michael Tunstall. Exponent Recoding and Regular Exponentiation Algorithms. In Bart Preneel, editor, *Progress in Cryptology, Second International Conference on Cryptology in Africa – AFRICACRYPT 2009*, volume 5580 of *Lecture Notes in Computer Science*, pages 334–349. Springer, 2009.

[KJJ99] P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In Michael J. Wiener, editor, *Advances in Cryptology, 19th Annual International Cryptology Conference – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.

[LM08] Patrick Longa and Ali Miri. New Composite Operations and Precomputation Scheme for Elliptic Curve Cryptosystems over Prime Fields. In Ronald Cramer, editor, *Public Key Cryptography, 11th International Workshop – PKC 2008*, volume 4939 of *Lecture Notes in Computer Science*, pages 229–247. Springer, 2008.

[Mel07] Nicolas Meloni. New Point Addition Formulae for ECC Applications. In Claude Carlet and Berk Sunar, editors, *Arithmetic of Finite Fields, First International Workshop – WAIFI 2007*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2007.

[MO90] François Morain and Jorger Olivos. Speeding up the computation on an elliptic curve using addition-subtraction chains. *Information Theory Appl.*, 24:531–543, 1990.

[Möl01] Bodo Möller. Securing Elliptic Curve Point Multiplication against Side-Channel Attacks. In George I. Davida and Yair Frankel, editors, *Information Security, 4th International Conference – ISC 2001*, volume 2200 of *Lecture Notes in Computer Science*, pages 324–334. Springer, 2001.

[Möl02] Bodo Möller. Parallelizable Elliptic Curve Point Multiplication Method with Resistance against Side-Channel Attacks. In Agnes Hui Chan and Virgil D. Gligor, editors, *Information Security Conference – ISC 2002*, volume 2433 of *Lecture Notes in Computer Science*, pages 402–413. Springer, 2002.

[Mon87] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, 1987.

[NNTW05] David Naccache, Phong Q. Nguyen, Michael Tunstall, and Claire Whelan. Experimenting with Faults, Lattices and the DSA. In Serge Vaudenay, editor, *Public Key Cryptography, 8th International Workshop – PKC 2005*, volume 3386 of *Lecture Notes in Computer Science*, pages 16–28. Springer, 2005.

[OT03] Katsuyuki Okeya and Tsuyoshi Takagi. The Width-w NAF Method Provides Small Memory and Fast Elliptic Scalar Multiplications Secure against Side Channel Attacks. In Marc Joye, editor, *Topics in Cryptology, The Cryptographers' Track at the RSA Conference – CT-RSA 2003*, volume 2612 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2003.

[QS02] Jean-Jacques Quisquater and David Samyde. Eddy Current for Magnetic Analysis with Active Sensor. Presented at e-Smart 2002, 2002.

[VD10] Alexandre Venelli and François Dassance. Faster Side-Channel Resistant Elliptic Curve Scalar Multiplication. In David Kohel and Robert Rolland, editors, *Arithmetic, Geometry, Cryptography and Coding Theory 2009*, volume 521 of *Contemporary Mathematics*, pages 29–40. American Mathematical Society, 2010.

[YKLM02] Sung-Ming Yen, Seungjoo Kim, Seongan Lim, and Sang-Jae Moon. A Countermeasure against One Physical Cryptanalysis May Benefit Another Attack. In Kwangjo Kim, editor, *Information Security and Cryptology, 4th International Conference – ICISC 2001*, volume 2288 of *Lecture Notes in Computer Science*, pages 414–427. Springer, 2002.

A Low Level Algorithms

This section provides low-level descriptions of the (X, Y)-only co-Z addition formulae with detailed field operation flow and memory usage. The addition with update is depicted in Algorithm 13. This algorithm has a total cost of $4\mathbf{M} + 2\mathbf{S} + 7\mathrm{A}$ and it involves 5 field registers.

Algorithm 13 (X, Y)-only co-Z addition with update – XYCZ-ADD

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_1', Y_1') s.t. $\boldsymbol{P} \equiv (X_1' : Y_1' : Z_3)$ and $\boldsymbol{P} + \boldsymbol{Q} \equiv (X_3 : Y_3 : Z_3)$ for some $Z_3 \in \mathbb{F}_q$
$(T_1 = X_1, T_2 = Y_1, T_3 = X_2, T_4 = Y_2)$

1. $T_5 \leftarrow T_3 - T_1$	$[X_2 - X_1]$
2. $T_5 \leftarrow T_5^2$	$[(X_2 - X_1)^2 = A]$
3. $T_1 \leftarrow T_1 \times T_5$	$[X_1 A = B]$
4. $T_3 \leftarrow T_3 \times T_5$	$[X_2 A = C]$
5. $T_4 \leftarrow T_4 - T_2$	$[Y_2 - Y_1]$
6. $T_5 \leftarrow T_4^2$	$[(Y_2 - Y_1)^2 = D]$
7. $T_5 \leftarrow T_5 - T_1$	$[D - B]$
8. $T_5 \leftarrow T_5 - T_3$	$[X_3]$
9. $T_3 \leftarrow T_3 - T_1$	$[C - B]$
10. $T_2 \leftarrow T_2 \times T_3$	$[Y_1(C - B)]$
11. $T_3 \leftarrow T_1 - T_5$	$[B - X_3]$
12. $T_4 \leftarrow T_4 \times T_3$	$[(Y_2 - Y_1)(B - X_3)]$
13. $T_4 \leftarrow T_4 - T_2$	$[Y_3]$

return $\big((T_5, T_4), (T_1, T_2)\big)$

For the conjugate addition, a time-memory trade-off is possible. The straightforward implementation leads to Algorithm 14 which costs $5\mathbf{M} + 3\mathbf{S} + 11\mathrm{A}$ and requires 7 field registers. A more tricky implementation was suggested in [GJM10b, App. A] which enables to save one field register for an additional cost of $5\mathrm{A}$. It is depicted in Algorithm 15 (but without the Z-coordinate computation). This implementation costs $5\mathbf{M} + 3\mathbf{S} + 16\mathrm{A}$ and it involves 6 field registers. As the conjugate addition is the most memory consuming part of the scalar multiplication algorithms presented in Section 2, saving one field register for the conjugate addition implies saving one field register for the whole scalar multiplication.

Algorithm 14 (X, Y)-only co-Z conjugate addition – XYCZ-ADDC

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_3', Y_3') s.t. $\boldsymbol{P} + \boldsymbol{Q} \equiv (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} - \boldsymbol{Q} \equiv (X_3' : Y_3' : Z_3)$ for some $Z_3 \in \mathbb{F}_q$
$(T_1 = X_1, T_2 = Y_1, T_3 = X_2, T_4 = Y_2)$

1. $T_5 \leftarrow T_3 - T_1$	$[X_2 - X_1]$
2. $T_5 \leftarrow T_5^2$	$[(X_2 - X_1)^2 = A]$
3. $T_1 \leftarrow T_1 \times T_5$	$[X_1 A = B]$
4. $T_3 \leftarrow T_3 \times T_5$	$[X_2 A = C]$
5. $T_5 \leftarrow T_4 + T_2$	$[Y_1 + Y_2]$
6. $T_4 \leftarrow T_4 - T_2$	$[Y_1 - Y_2]$
7. $T_6 \leftarrow T_3 - T_1$	$[C - B]$
8. $T_2 \leftarrow T_2 \times T_6$	$[Y_1(C - B)]$
9. $T_6 \leftarrow T_3 + T_1$	$[B + C]$
10. $T_3 \leftarrow T_4^2$	$[(Y_2 - Y_1)^2]$
11. $T_3 \leftarrow T_3 - T_6$	$[X_3]$
12. $T_7 \leftarrow T_1 - T_3$	$[B - X_3]$
13. $T_4 \leftarrow T_4 \times T_7$	$[(Y_1 - Y_2)(B - X_3)]$
14. $T_4 \leftarrow T_4 - T_2$	$[Y_3]$
15. $T_7 \leftarrow T_5^2$	$[(Y_2 + Y_1)^2 = F]$
16. $T_7 \leftarrow T_7 - T_6$	$[X_3']$
17. $T_6 \leftarrow T_7 - T_1$	$[X_3' - B]$
18. $T_6 \leftarrow T_6 \times T_5$	$[(Y_1 + Y_2)(X_3' - B)]$
19. $T_6 \leftarrow T_6 - T_2$	$[Y_3']$

return $\big((T_3, T_4), (T_7, T_6)\big)$

Algorithm 15 (X, Y)-only co-Z conjugate addition – XYCZ-ADDC

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_3, Y_3) and (X_3', Y_3') s.t. $\boldsymbol{P} + \boldsymbol{Q} \equiv (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} - \boldsymbol{Q} \equiv (X_3' : Y_3' : Z_3)$ for some $Z_3 \in \mathbb{F}_q$
$(T_1 = X_1, T_2 = Y_1, T_3 = X_2, T_4 = Y_2)$

1. $T_5 \leftarrow T_3 - T_1$	$[X_2 - X_1]$	13. $T_3 \leftarrow T_3 - T_1$	$[C - B]$
2. $T_5 \leftarrow T_5^2$	$[(X_2 - X_1)^2 = A]$	14. $T_3 \leftarrow T_3 \times T_2$	$[Y_1(C - B) = E]$
3. $T_1 \leftarrow T_1 \times T_5$	$[X_1 A = B]$	15. $T_4 \leftarrow T_4 - T_2$	$[Y_2 - Y_1]$
4. $T_3 \leftarrow T_3 \times T_5$	$[X_2 A = C]$	16. $T_2 \leftarrow 2T_2$	$[2Y_1]$
5. $T_5 \leftarrow T_4 - T_2$	$[Y_2 - Y_1]$	17. $T_2 \leftarrow T_2 + T_4$	$[Y_1 + Y_2]$
6. $T_5 \leftarrow T_5^2$	$[(Y_1 - Y_2)^2 = D]$	18. $T_6 \leftarrow T_6 - T_1$	$[X_3' - B]$
7. $T_5 \leftarrow T_5 - T_1$	$[D - B]$	19. $T_2 \leftarrow T_2 \times T_6$	$[(Y_1 + Y_2)(X_3' - B)]$
8. $T_5 \leftarrow T_5 - T_3$	$[X_3]$	20. $T_2 \leftarrow T_2 - T_3$	$[Y_3']$
9. $T_6 \leftarrow T_2 + T_4$	$[Y_1 + Y_2]$	21. $T_6 \leftarrow T_6 + T_1$	$[X_3']$
10. $T_6 \leftarrow T_6^2$	$[(Y_1 + Y_2)^2 = F]$	22. $T_1 \leftarrow T_1 - T_5$	$[B - X_3]$
11. $T_6 \leftarrow T_6 - T_1$	$[F - B]$	23. $T_4 \leftarrow T_4 \times T_1$	$[(Y_2 - Y_1)(B - X_3)]$
12. $T_6 \leftarrow T_6 - T_3$	$[X_3']$	24. $T_4 \leftarrow T_4 - T_3$	$[Y_3]$

return $\big((T_5, T_4), (T_6, T_2)\big)$

As explained in Section 2, a double-add algorithm can be designed by performing an addition with update followed by a conjugate addition. By doing so in the (X, Y)-only setting, it is possible to trade one field multiplication for one field squaring and a few field additions. Let $\boldsymbol{P} = (X_1, Y_1)$ and $\boldsymbol{Q} = (X_2, Y_2)$ where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are co-Z. The addition of $\boldsymbol{P}$ and $\boldsymbol{Q}$ yields a point $\boldsymbol{P} + \boldsymbol{Q} = (X_3, Y_3)$ satisfying (5) and a co-Z update $\boldsymbol{P} = (X_1', Y_1')$. The trick proposed in [GJM10a] is based on the following equality:

$$\begin{aligned} 2Y_3 &= (Y_2 - Y_1 + B - X_3)^2 - (Y_2 - Y_1)^2 - (B - X_3)^2 - 2E \\ &= (Y_2 - Y_1 + B - X_3)^2 - D - (X_1' - X_3)^2 - 2Y_1' \,. \end{aligned}$$

As the computation of $(X_1' - X_3)^2$ is already involved in the subsequent addition of $\boldsymbol{P} + \boldsymbol{Q}$ and $\boldsymbol{P}$, the multiplication between $(Y_2 - Y_1)$ and $(B - X_3)$ can be traded for a square of their sum, without requiring additional field multiplications or squarings but only field additions. Let us now denote $2\boldsymbol{P} + \boldsymbol{Q} = (X_4, Y_4)$ and $\boldsymbol{Q} = (X_4', Y_4')$ the two co-Z points resulting from the conjugate addition between $\boldsymbol{P} + \boldsymbol{Q} = (X_3, Y_3)$ and $\boldsymbol{P} = (X_1', Y_1')$. Let us also denote $A_2 = (X_1' - X_3)^2$, $B_2 = X_3 A_2$, $C_2 = X_1' A_2$, *etc.* the intermediate results of the conjugate addition. In order to deal with $2Y_3$ instead of Y_3, the conjugate addition must be slightly modified. For such a purpose, one just need to replace Y_1', B_2, and C_2 by $2Y_1'$, $4B_2$, and $4C_2$ respectively. It can then be checked that applying the usual co-Z conjugate addition formula (see (5) and (6)) results in the replacement of (X_4, Y_4) and (X_4', Y_4') by $(4X_4, 8Y_4)$ and $(4X_4', 8Y_4')$ respectively. This modification does not affect the soundness of the result since we know that there exists Z_4 such that $2\boldsymbol{P} + \boldsymbol{Q} \equiv (X_4 : Y_4 : Z_4)$ and $\boldsymbol{Q} \equiv (X_4' : Y_4' : Z_4)$, which implies $2\boldsymbol{P} + \boldsymbol{Q} \equiv (4X_4 : 8Y_4 : 2Z_4)$ and $\boldsymbol{Q} \equiv (4X_4' : 8Y_4' : 2Z_4)$. To summarize, the computation of $2Y_3$ instead of Y_3 trades one field multiplication for one field squaring and 4 field additions. Moreover, the quadrupling of B_2 and C_2 involves 4 supplementary field additions (the doubling of Y_1' being already involved in the computation of $2Y_3$). On the whole, the trick suggested in [GJM10a] enables trading $1\mathbf{M}$ for $1\mathbf{S} + 8\mathrm{A}$. In practice such a strategy should only be followed if the field addition cost is really negligible. Algorithm 16 gives the low-level description of the double-add algorithm based on the above trick with the conjugate addition implemented as in Algorithm 15.

Algorithm 16 (X, Y)-only co-Z double-add with update – XYCZ-DA

Input: (X_1, Y_1) and (X_2, Y_2) s.t. $\boldsymbol{P} \equiv (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} \equiv (X_2 : Y_2 : Z)$ for some $Z \in \mathbb{F}_q$, $\boldsymbol{P}, \boldsymbol{Q} \in E(\mathbb{F}_q)$
Output: (X_4, Y_4) and (X_4', Y_4') s.t. $2\boldsymbol{P} + \boldsymbol{Q} \equiv (X_4 : Y_4 : Z_4)$ and $\boldsymbol{Q} \equiv (X_4' : Y_4' : Z_4)$ for some $Z_4 \in \mathbb{F}_q$

1. $(T_1 = X_1, T_2 = Y_1, T_3 = X_2, T_4 = Y_2)$

– First addition:

1. $T_5 \leftarrow T_3 - T_1$ $[X_2 - X_1]$
2. $T_5 \leftarrow T_5^2$ $[(X_2 - X_1)^2 = A_1]$
3. $T_1 \leftarrow T_1 \times T_5$ $[X_1 A_1 = B_1]$
4. $T_3 \leftarrow T_3 \times T_5$ $[X_2 A_1 = C_1]$
5. $T_4 \leftarrow T_4 - T_2$ $[Y_2 - Y_1]$
6. $T_5 \leftarrow T_4^2$ $[(Y_2 - Y_1)^2 = D_1]$
7. $T_6 \leftarrow T_5 - T_1$ $[D_1 - B_1]$
8. $T_6 \leftarrow T_6 - T_3$ $[X_3]$
9. $T_4 \leftarrow T_4 + T_1$ $[Y_2 - Y_1 + B_1]$
10. $T_4 \leftarrow T_4 - T_6$ $[Y_2 - Y_1 + B_1 - X_3]$
11. $T_4 \leftarrow T_4^2$ $[(Y_2 - Y_1 + B_1 - X_3)^2]$
12. $T_4 \leftarrow T_4 - T_5$ $[(Y_2 - Y_1 + B_1 - X_3)^2 - D_1]$
13. $T_5 \leftarrow T_1 - T_6$ $[B_1 - X_3]$
14. $T_5 \leftarrow T_5^2$ $[(B_1 - X_3)^2 = A_2]$
15. $T_4 \leftarrow T_4 - T_5$ $[(Y_2 - Y_1 + B_1 - X_3)^2 - D_1 - A_2]$
16. $T_3 \leftarrow T_3 - T_1$ $[C_1 - B_1]$
17. $T_2 \leftarrow T_2 \times T_3$ $[Y_1(C_1 - B_1)]$
18. $T_2 \leftarrow 2T_2$ $[2Y_1(C_1 - B_1)]$
19. $T_4 \leftarrow T_4 - T_2$ $[2Y_3]$

– Intermediate result:

$(T_6 = X_3, T_4 = 2Y_3, T_1 = X_1', T_2 = 2Y_1', T_5 = A_2)$

– Conjugate addition:

20. $T_6 \leftarrow T_6 \times T_5$ $[X_3 A_2 = B_2]$
21. $T_6 \leftarrow 4T_6$ $[4B_2]$
22. $T_1 \leftarrow T_1 \times T_5$ $[X_1' A_2 = C_2]$
23. $T_1 \leftarrow 4T_1$ $[4C_2]$
24. $T_5 \leftarrow T_2 - T_4$ $[2(Y_1' - Y_3)]$
25. $T_5 \leftarrow T_5^2$ $[(Y_1' - Y_3)^2 = 4D_2]$
26. $T_5 \leftarrow T_5 - T_6$ $[4(D_2 - B_2)]$
27. $T_5 \leftarrow T_5 - T_1$ $[4X_4]$
28. $T_3 \leftarrow T_2 + T_4$ $[2(Y_3 + Y_1')]$
29. $T_3 \leftarrow T_3^2$ $[4(Y_3 + Y_1')^2 = 4F_2]$
30. $T_3 \leftarrow T_3 - T_6$ $[4(F_2 - B_2)]$
31. $T_3 \leftarrow T_3 - T_1$ $[4X_4']$
32. $T_1 \leftarrow T_1 - T_6$ $[4(C_2 - B_2)]$
33. $T_1 \leftarrow T_1 \times T_4$ $[8Y_3(C_2 - B_2) = 8E_2]$
34. $T_2 \leftarrow T_2 - T_4$ $[2(Y_1' - Y_3)]$
35. $T_4 \leftarrow 2T_4$ $[4Y_3]$
36. $T_4 \leftarrow T_4 + T_2$ $[2(Y_1' + Y_3)]$
37. $T_3 \leftarrow T_3 - T_6$ $[4(X_4' - B_2)]$
38. $T_4 \leftarrow T_4 \times T_3$ $[8(Y_1' + Y_3)(X_4' - B_2)]$
39. $T_4 \leftarrow T_4 - T_1$ $[8Y_4']$
40. $T_3 \leftarrow T_3 + T_6$ $[X_4']$
41. $T_6 \leftarrow T_6 - T_5$ $[4(B_2 - X_4)]$
42. $T_2 \leftarrow T_2 \times T_6$ $[8(Y_1' - Y_3)(B_2 - X_4)]$
43. $T_2 \leftarrow T_2 - T_1$ $[8Y_4]$

return $\big((T_5, T_2), (T_3, T_4)\big)$

Eventually, Table 2 summarizes the computational cost and memory consumption of the different implementations. The last row of the table gives the cost of the double-add implementation using the above trick with the conjugate addition implemented as in Algorithm 14.

Table 2. Low-level implementation performances.

Operation	Implementation	Costs	# field registers
Addition	Algorithm 13	$4\mathbf{M} + 2\mathbf{S} + 7\mathrm{A}$	5
addition	Algorithm 15	$5\mathbf{M} + 3\mathbf{S} + 16\mathrm{A}$	6
Conjugate	Algorithm 14	$5\mathbf{M} + 3\mathbf{S} + 11\mathrm{A}$	7
Double-add	Algo. 13 + Algo. 15	$9\mathbf{M} + 5\mathbf{S} + 23\mathrm{A}$	6
	Algo. 13 + Algo. 14	$9\mathbf{M} + 5\mathbf{S} + 18\mathrm{A}$	7
	Algorithm 16	$8\mathbf{M} + 6\mathbf{S} + 31\mathrm{A}$	6
	n/a	$8\mathbf{M} + 6\mathbf{S} + 26\mathrm{A}$	7

B Coordinate Recovery Algorithms

This section provides the formal description of the coordinate recovery algorithm described in Section 2. This algorithm takes the (X, Y)-coordinates of two co-Z points $\boldsymbol{P}$ and $\boldsymbol{Q}$ as well as the affine coordinates of $\boldsymbol{P}$, and it computes the affine coordinates of $\boldsymbol{Q}$. This algorithm has a computational cost of $1\mathbf{I} + 6\mathbf{M} + 1\mathbf{S}$.

Algorithm 17 Coordinate Recovery – CoordRec

Input: $\boldsymbol{P} = (x, y)$, $\boldsymbol{P}' = (X_P, Y_P)$ s.t. $(X_P/Z^2, Y_P/Z^3) = (x, y)$ for some Z, and $\boldsymbol{Q}' = (X_Q, Y_Q)$
Output: $\boldsymbol{Q} = (X_Q/Z^2, Y_Q/Z^3)$

1. $T_1 \leftarrow x \times Y_P$	$[xY]$	5. $T_2 \leftarrow T_1^2$	$[Z^{-2}]$
2. $T_1 \leftarrow T_1^{-1}$	$[(xY)^{-1}]$	6. $T_1 \leftarrow T_2 \times T_1$	$[Z^{-3}]$
3. $T_1 \leftarrow T_1 \times X_P$	$[X(xY)^{-1}]$	7. $x_Q \leftarrow X_Q \times T_2$	
4. $T_1 \leftarrow T_1 \times y$	$[Xy(xY)^{-1} = Z^{-1}]$	8. $y_Q \leftarrow Y_Q \times T_1$	

return (x_Q, y_Q)

C Initial Point Doubling and Tripling

We describe hereafter the doubling and tripling operations taking the affine coordinates of a point $\boldsymbol{P}$ and computing the (X, Y)-only co-Z points $(\boldsymbol{Q}, \boldsymbol{P})$ where $\boldsymbol{Q}$ either equals $[2]\boldsymbol{P}$ or $[3]\boldsymbol{P}$. Both operations start by a Jacobian doubling of $\boldsymbol{P} = (x_1, y_1, 1)$ which yields $\boldsymbol{Q} = (X_2, Y_2, Z) = [2]\boldsymbol{P}$. Afterward $\boldsymbol{P}$ is updated to be co-Z to $\boldsymbol{Q}$ by $\boldsymbol{P} \equiv (Z^2 x_P : Z^3 y_P : Z)$. For the tripling, an additional co-Z addition with update is performed to obtain a pair of co-Z points $([3]\boldsymbol{P}, \boldsymbol{P})$. From the Jacobian doubling formula (see (3)), we see that the doubling of a point $\boldsymbol{P}$ in affine coordinates can be simplified. Let $\boldsymbol{P} = (x_1, y_1, 1)$, we have $[2]\boldsymbol{P} = (X_2, Y_2, Z_2)$ where $X_2 = B^2 - 2A$, $Y_2 = B(A - X_2) - 8y_1^4$ and $Z_2 = 2y_1$, with $A = 4x_1y_1^2$, $B = 3x_1^2 + a$. Note that the update of $\boldsymbol{P}$ can be deduced from $\boldsymbol{P} \equiv (x_1 Z_2^2 : y_1 Z_2^3 : Z_2) = (4x_1y_1^2 : 8y_1^4 : 2y_1)$. The low level description of the doubling operation is depicted in Algorithm 18. Its computational cost is $2\mathbf{M} + 4\mathbf{S} + 10\mathrm{A}$ and it requires 6 field registers.

Algorithm 18 (X, Y)-only initial doubling with Co-Z Update – XYCZ-IDBL

Input: $\boldsymbol{P} = (x_1, y_1)$
Output: (X_2, Y_2) and (X_1', Y_1') s.t. $[2]\boldsymbol{P} \equiv (X_2 : Y_2 : Z_2)$ and $\boldsymbol{P} \equiv (X_1' : Y_1' : Z_2)$ for some $Z_2 \in \mathbb{F}_q$
$(T_1 = x_1, T_2 = y_1)$

1. $T_3 \leftarrow T_1^2$	$[x_1^2]$	9. $T_6 \leftarrow T_3^2$	$[B^2]$
2. $T_4 \leftarrow 2T_3$	$[2x_1^2]$	10. $T_6 \leftarrow T_6 - T_5$	$[B^2 - A]$
3. $T_3 \leftarrow T_3 + T_4$	$[3x_1^2]$	11. $T_6 \leftarrow T_6 - T_5$	$[X_2]$
4. $T_3 \leftarrow T_3 + a$	$[3x_1^2 + a = B]$	12. $T_1 \leftarrow T_5 - T_6$	$[A - X_2]$
5. $T_4 \leftarrow T_2^2$	$[y_1^2]$	13. $T_1 \leftarrow T_1 \times T_3$	$[B(A - X_2)]$
6. $T_4 \leftarrow 2T_4$	$[2y_1^2]$	14. $T_3 \leftarrow T_4^2$	$[4y_1^4]$
7. $T_5 \leftarrow 2T_4$	$[4y_1^2]$	15. $T_3 \leftarrow 2T_3$	$[8y_1^4 = Y_1']$
8. $T_5 \leftarrow T_5 \times T_1$	$[4x_1y_1^2 = X_1' = A]$	16. $T_1 \leftarrow T_1 - T_3$	$[Y_2]$

return $\big((T_6, T_1), (T_5, T_3)\big)$

Algorithm 19 gives the high-level description of the tripling operation. From the low-level algorithm for the doubling and the co-Z addition with update (see Algorithms 18 and 13), it is easy to check that the tripling can be computed at the cost of $6\mathbf{M} + 6\mathbf{S} + 17\mathrm{A}$ using 6 field registers.

Algorithm 19 (X, Y)-only initial tripling with Co-Z Update – XYCZ-ITPL

Input: $\boldsymbol{P} = (x_1, y_1)$
Output: (X_3, Y_3) and (X_1'', Y_1'') s.t. $[3]\boldsymbol{P} \equiv (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} \equiv (X_1'' : Y_1'' : Z_3)$ for some $Z_3 \in \mathbb{F}_q$
1. $\big((X_2, Y_2), (X_1', Y_1')\big) \leftarrow \mathsf{XYCZ\text{-}IDBL}(x_1, y_1)$
2. $\big((X_3, Y_3), (X_1'', Y_1'')\big) \leftarrow \mathsf{XYCZ\text{-}ADD}\big((X_1', Y_1'), (X_2, Y_2)\big)$
3. **return** $\big((X_3, Y_3), (X_1'', Y_1'')\big)$

D Regular Conditional Point Inversion

In this section, we provide solutions to implement the operation $\boldsymbol{P} \leftarrow (-1)^b \boldsymbol{P}$ in a regular way for some $\boldsymbol{P} = (X, Y)$ and $b \in \{0, 1\}$. A first solution is to process the following steps:

1. $R_0 \leftarrow Y$
2. $R_1 \leftarrow -Y$
3. $Y \leftarrow R_b$

This solution is very simple and efficient: it only costs $1\mathrm{A}$ for computing $-Y$ (other steps being processed by pointer arithmetic of negligible cost). However, when $b = 0$ the inversion of Y is a dummy operation which renders the implementation subject to *safe-error attacks* [YKLM02]. Indeed, by injecting a fault in the register R_1 and checking the result correctness, one could see whether R_1 is used (which would imply a faulty result) or not, and hence deduce the value of b. A simple countermeasure to avoid such a weakness consists in randomizing the buffer allocation, which yields to the following solution:

1. $r \leftarrow^{\$} \{0, 1\}$
2. $R_r \leftarrow Y$
3. $R_{r \oplus 1} \leftarrow -Y$
4. $Y \leftarrow R_{r \oplus b}$

Finally, if one still want to avoid any dummy operation, a third solution is possible:

1. $R_0 \leftarrow Y$
2. $R_1 \leftarrow -Y$
3. $Y \leftarrow R_b + R_b - R_{b \oplus 1}$

However this solution implies a total cost of $3\mathrm{A}$ which should deter its use in practice.