Co-Z Addition Formulæ and Binary Ladders on Elliptic Curves
(Extended Abstract)

Raveen R. Goundar[1], Marc Joye[2], and Atsuko Miyaji[1]

[1] Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
raveen.rg@gmail.com,miyaji@jaist.ac.jp
[2] Technicolor, Security & Content Protection Labs
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
marc.joye@technicolor.com

Abstract. Meloni recently introduced a new type of arithmetic on elliptic curves when adding projective points sharing the same Z-coordinate. This paper presents further co-Z addition formulæ for various point additions on Weierstraß elliptic curves. It explains how the use of conjugate point addition and other implementation tricks allow one to develop efficient scalar multiplication algorithms making use of co-Z arithmetic. Specifically, this paper describes efficient co-Z based versions of Montgomery ladder and Joye's double-add algorithm. Further, the resulting implementations are protected against a large variety of implementation attacks.

Key words: Elliptic curves, Meloni's technique, Jacobian coordinates, regular binary ladders, implementation attacks, embedded systems.

1 Introduction

Elliptic curve cryptography (ECC), introduced independently by Koblitz [16] and Miller [23] in the mid-eighties, shows an increasing impact in our everyday lives where the use of memory-constrained devices such as smart cards and other embedded systems is ubiquitous. Its main advantage resides in a smaller key size. The efficiency of ECC is dominated by an operation called *scalar multiplication*, denoted as $k\boldsymbol{P}$ where $\boldsymbol{P} \in E(\mathbb{F}_q)$ is a rational point on an elliptic curve $E/\mathbb{F}_q$ and k acts as a secret scalar. This means adding a point $\boldsymbol{P}$ on elliptic curve E, k times. In constrained environments, scalar multiplication is usually implemented through binary methods, which take on input the binary representation of scalar k.

There are many techniques proposed in the literature aiming at improving the efficiency of ECC. They rely on explicit addition formulæ, alternative curve parameterizations, extended point representations, extended coordinate systems, or higher-radix or non-standard scalar representations. See e.g. [1] for a survey of some techniques.

In this paper, we target the basic operation, namely the point addition. More specifically, we propose *new co-Z addition formulæ*. Co-Z arithmetic was introduced by Meloni in [22] as a means to efficiently add two projective points sharing the same Z-coordinate. The initial co-Z addition formula proposed by Meloni greatly improves on the general point addition. The drawback is that this fast formula is by construction limited to Euclidean addition chains. The efficiency being dependent on the length of the chain, Meloni suggests to represent scalar k in the computation of $k\boldsymbol{P}$ with the so-called Zeckendorf's representation and proposes a "Fibonacci-and-add" algorithm. The resulting algorithm is efficient but still slower than its binary counterparts. We take a completely different approach in this paper and consider *conjugate point addition* [11, 19]. The basic observation is that the addition of two points, $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$, yields almost for free the value of their difference, $\boldsymbol{S} = \boldsymbol{P} - \boldsymbol{Q}$. This combined operation is referred to as a conjugate point addition. We propose efficient conjugate point addition formulæ making use of co-Z arithmetic and develop a new strategy for the efficient implementation of scalar multiplications. Specifically, we show that the Montgomery ladder [24] and its dual version [14] can be adapted to accommodate our new co-Z formulæ. As a result, we get efficient co-Z based scalar multiplication algorithms using the regular binary representation.

Last but not least, our scalar multiplication algorithms resist against certain implementation attacks. Because they are built on *highly* regular algorithms, our algorithms inherit of their security features. In particular, they are naturally protected against SPA-type attacks [17] and safe-error attacks [26, 27]. Moreover, they can be combined with other known countermeasures to protect against other classes of attacks. Finally, we note that, unlike [5, 9, 13, 21], our version of the Montgomery ladder makes use of the complete point coordinates and so offers a better resistance against (regular) fault attacks [4].

2 Preliminaries

Let $\mathbb{F}_q$ be a finite field with characteristic $\neq 2, 3$. Consider an elliptic curve E over $\mathbb{F}_q$ given by the Weierstraß equation $y^2 = x^3 + ax + b$, with discriminant $\Delta = -16(4a^3 + 27b^2) \neq 0$. This section explains how to get efficient arithmetic on elliptic curves over $\mathbb{F}_q$. The efficiency is measured in terms of field multiplications and squarings. The cost of field additions is neglected. We let M and S denote the cost of a multiplication and of a squaring in $\mathbb{F}_q$, respectively. A typical ratio is $\mathsf{S}/\mathsf{M} = 0.8$.

2.1 Jacobian coordinates

In order to avoid the computation of inverses in $\mathbb{F}_q$, it is advantageous to make use of Jacobian coordinates. A finite point (x, y) is then represented by a triplet $(X : Y : Z)$ such that $x = X/Z^2$ and $y = Y/Z^3$. The curve equation becomes

$$E_{/\mathbb{F}_q} : Y^2 = X^3 + aXZ^4 + bZ^6 \ .$$

The point at infinity, $\boldsymbol{O}$, is the only point with a Z-coordinate equal to 0. It is represented by $\boldsymbol{O} = (1 : 1 : 0)$. Note that, for any nonzero $\lambda \in \mathbb{F}_q$, the triplets $(\lambda^2 X : \lambda^3 Y : \lambda Z)$ represent the same point.

It is well known that the set of points on an elliptic curve form a group under the chord-and-tangent law. The neutral element is the point at infinity $\boldsymbol{O}$. Let $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z_2)$ be two points on E, with $\boldsymbol{P}, \boldsymbol{Q} \neq \boldsymbol{O}$. The inverse of $\boldsymbol{P}$ is $-\boldsymbol{P} = (X_1 : -Y_1 : Z_1)$. If $\boldsymbol{P} = -\boldsymbol{Q}$ then $\boldsymbol{P} + \boldsymbol{Q} = \boldsymbol{O}$. If $\boldsymbol{P} \neq \pm\boldsymbol{Q}$ then their sum $\boldsymbol{P} + \boldsymbol{Q}$ is given by $(X_3 : Y_3 : Z_3)$ where

$$X_3 = R^2 + G - 2V, \;\; Y_3 = R(V - X_3) - 2K_1 G, \;\; Z_3 = ((Z_1 + Z_2)^2 - I_1 - I_2)H$$

with $R = 2(K_1 - K_2)$, $G = FH$, $V = U_1 F$, $K_1 = Y_1 J_2$, $K_2 = Y_2 J_1$, $F = (2H)^2$, $H = U_1 - U_2$, $U_1 = X_1 I_2$, $U_2 = X_2 I_1$, $J_1 = I_1 Z_1$, $J_2 = I_2 Z_2$, $I_1 = {Z_1}^2$ and $I_2 = {Z_2}^2$ [7].[†] We see that that the addition of two (different) points requires $\underline{11\mathsf{M} + 5\mathsf{S}}$.

The double of $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$ (i.e., when $\boldsymbol{P} = \boldsymbol{Q}$) is given by $(\mathrm{X}(2\boldsymbol{P}) : \mathrm{Y}(2\boldsymbol{P}) : \mathrm{Z}(2\boldsymbol{P}))$ where

$$\mathrm{X}(2\boldsymbol{P}) = M^2 - 2S, \;\; \mathrm{Y}(2\boldsymbol{P}) = M(S - \mathrm{X}(2\boldsymbol{P})) - 8L, \;\; \mathrm{Z}(2\boldsymbol{P}) = (Y_1 + Z_1)^2 - E - N$$

with $M = 3B + a N^2$, $S = 2((X_1 + E)^2 - B - L)$, $L = E^2$, $B = {X_1}^2$, $E = {Y_1}^2$ and $N = {Z_1}^2$ [2]. Hence, the double of a point can be obtained with $\underline{1\mathsf{M} + 8\mathsf{S} + 1\mathsf{c}}$, where c denotes the cost of a multiplication by curve parameter a.

An interesting case is when curve parameter a is $a = -3$, in which case point doubling costs $3\mathsf{M} + 5\mathsf{S}$ [6]. In the general case, point doubling can be sped up by representing points $(X_i : Y_i : Z_i)$ with an additional coordinate, namely $T_i = a{Z_i}^4$. This extended representation is referred to as *modified Jacobian coordinates* [7]. The cost of point doubling drops to $3\mathsf{M} + 5\mathsf{S}$ at the expense of a slower point addition.

2.2 Co-Z point addition

In [22], Meloni considers the case of adding two (different) points having the same Z-coordinate. When points $\boldsymbol{P}$ and $\boldsymbol{Q}$ share the same Z-coordinate, say $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$, then their sum $\boldsymbol{P} + \boldsymbol{Q} = (X_3 : Y_3 : Z_3)$ can be evaluated faster as

$$X_3 = D - W_1 - W_2, \;\; Y_3 = (Y_1 - Y_2)(W_1 - X_3) - A_1, \;\; Z_3 = Z(X_1 - X_2)$$

with $A_1 = Y_1(W_1 - W_2)$, $W_1 = X_1 C$, $W_2 = X_2 C$, $C = (X_1 - X_2)^2$ and $D = (Y_1 - Y_2)^2$. This operation is referred to as the ZADD operation. The key observation in Meloni's addition is that the computation of $\boldsymbol{R} = \boldsymbol{P} + \boldsymbol{Q}$ yields for free an equivalent representation for input point $\boldsymbol{P}$ with its Z-coordinate equal to that of output point $\boldsymbol{R}$, namely

$$(X_1(X_1 - X_2)^2 : Y_1(X_1 - X_2)^3 : Z_3) = (W_1 : A_1 : Z_3) \sim \boldsymbol{P} \;\; .$$

[†] Actually, Cohen et al. in [7] reports formulæ in $12\mathsf{M} + 4\mathsf{S}$. The above formulæ in $11\mathsf{M} + 5\mathsf{S}$ are essentially the same: A multiplication is traded against a squaring in the expression of Z_3 by computing $Z_1 \cdot Z_2$ as $(Z_1 + Z_2)^2 - {Z_1}^2 - {Z_2}^2$. See [2, 18].

The corresponding operation is denoted ZADDU (i.e., ZADD with update) and is presented in Algorithm 1. It is readily seen that it requires $\underline{5\mathsf{M} + 2\mathsf{S}}$.

Algorithm 1 Co-Z point addition with update (ZADDU)

Require: $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$
Ensure: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow \text{ZADDU}(\boldsymbol{P}, \boldsymbol{Q})$ where $\boldsymbol{R} \leftarrow \boldsymbol{P} + \boldsymbol{Q} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : Z_3)$ with $Z_3 = \lambda Z_1$ for some $\lambda \neq 0$

function ZADDU($\boldsymbol{P}, \boldsymbol{Q}$)
  $C \leftarrow (X_1 - X_2)^2$
  $W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
  $D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
  $X_3 \leftarrow D - W_1 - W_2$; $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$; $Z_3 \leftarrow Z(X_1 - X_2)$
  $X_1 \leftarrow W_1$; $Y_1 \leftarrow A_1$; $Z_1 \leftarrow Z_3$
end function

3 Binary Scalar Multiplication Algorithms

This section discusses known scalar multiplication algorithms. Given a point $\boldsymbol{P}$ in $E(\mathbb{F}_q)$ and a scalar $k \in \mathbb{N}$, the *scalar multiplication* is the operation consisting in calculating $\boldsymbol{Q} = k\boldsymbol{P}$ — that is, $\boldsymbol{P} + \cdots + \boldsymbol{P}$ (k times).

We focus on binary methods, taking on input the binary representation of scalar k, $k = (k_{n-1}, \ldots, k_0)_2$ with $k_i \in \{0, 1\}$, $0 \leqslant i \leqslant n - 1$. The corresponding algorithms present the advantage of demanding low memory requirements and are therefore well suited for memory-constrained devices like smart cards.

A classical method for evaluating $\boldsymbol{Q} = k\boldsymbol{P}$ exploits the obvious relation that $k\boldsymbol{P} = 2(\lfloor k/2 \rfloor \boldsymbol{P})$ if k is even and $k\boldsymbol{P} = 2(\lfloor k/2 \rfloor \boldsymbol{P}) + \boldsymbol{P}$ if k is odd. Iterating the process then yields a scalar multiplication algorithm, left-to-right scanning scalar k. The resulting algorithm, also known as *double-and-add algorithm*, is depicted in Algorithm 2. It requires two (point) registers, $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$. Register $\boldsymbol{R_0}$ acts as an accumulator and register $\boldsymbol{R_1}$ is used to store the value of input point $\boldsymbol{P}$.

Algorithm 2 Left-to-right binary method

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2: **for** $i = n - 1$ down to 0 **do**
3: $\boldsymbol{R_0} \leftarrow 2\boldsymbol{R_0}$
4: **if** $(k_i = 1)$ **then** $\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

Algorithm 3 Montgomery ladder

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2: **for** $i = n - 1$ down to 0 **do**
3: $b \leftarrow k_i$; $\boldsymbol{R_{1-b}} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{R_b}$
4: $\boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

Although efficient (memory- and computation-wise), the left-to-right binary method is subject to SPA-type attacks [17]. From a power trace, an adversary able to distinguish between point doublings and point additions can easily recover the value of scalar k. A simple countermeasure is to insert a dummy point addition when scalar bit k_i is 0. Using an additional (point) register, say $\boldsymbol{R_{-1}}$, Line 4 in Algorithm 2 can be replaced with $\boldsymbol{R_{-k_i}} \leftarrow \boldsymbol{R_{-k_i}} + \boldsymbol{R_1}$. The so-obtained algorithm, called *double-and-add-always algorithm* [8], now appears as a regular succession of a point doubling followed by a point addition. However, it also becomes subject to safe-error attacks [26, 27]. By timely inducing a fault at iteration i during the point addition $\boldsymbol{R_{-k_i}} \leftarrow \boldsymbol{R_{-k_i}} + \boldsymbol{R_1}$, an adversary can determine whether the operation is dummy or not by checking the correctness of the output, and so deduce the value of scalar bit k_i. If the output is correct then $k_i = 0$ (dummy point addition); if not, $k_i = 1$ (effective point addition).

A scalar multiplication algorithm featuring a regular structure without dummy operation is the so-called *Montgomery ladder* [24] (see also [15]). It is detailed in Algorithm 3. Each iteration is comprised of a point addition followed by a point doubling. Further, compared to the double-and-add-always algorithm, it only requires two (point) registers and all involved operations are effective. Montgomery ladder provides thus a natural protection against SPA-type attacks and safe-error attacks. A useful property of Montgomery ladder is that its main loop keeps invariant the difference between $\boldsymbol{R_1}$ and $\boldsymbol{R_0}$. Indeed, if we let ${\boldsymbol{R_b}}^{(\text{new})} = \boldsymbol{R_b} + \boldsymbol{R_{1-b}}$ and ${\boldsymbol{R_{1-b}}}^{(\text{new})} = 2\boldsymbol{R_{1-b}}$ denote the registers after the updating step, we observe that ${\boldsymbol{R_b}}^{(\text{new})} - {\boldsymbol{R_{1-b}}}^{(\text{new})} = (\boldsymbol{R_b} + \boldsymbol{R_{1-b}}) - 2\boldsymbol{R_{1-b}} = \boldsymbol{R_b} - \boldsymbol{R_{1-b}}$. This allows one to compute scalar multiplications on elliptic curves using the x-coordinate only [24] (see also [5, 9, 13, 21]).

Algorithm 4 Right-to-left binary method

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2: **for** $i = 0$ to $n - 1$ **do**
3: **if** $(k_i = 1)$ **then** $\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$
4: $\boldsymbol{R_1} \leftarrow 2\boldsymbol{R_1}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

Algorithm 5 Joye's double-add

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2: **for** $i = 0$ to $n - 1$ **do**
3: $b \leftarrow k_i$
4: $\boldsymbol{R_{1-b}} \leftarrow 2\boldsymbol{R_{1-b}} + \boldsymbol{R_b}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

There exists a right-to-left variant of Algorithm 2. This is another classical method for evaluating $\boldsymbol{Q} = k\boldsymbol{P}$. It stems from the observation that, letting $k = \sum_{i=0}^{n-1} k_i 2^i$ the binary expansion of k, we can write $k\boldsymbol{P} = \sum_{k_i=1} 2^i\boldsymbol{P}$. A first (point) register $\boldsymbol{R_0}$ serves as an accumulator and a second (point) register $\boldsymbol{R_1}$ is used to contain the successive values of $2^i\boldsymbol{P}$, $0 \leqslant i \leqslant n - 1$. When $k_i = 1$, $\boldsymbol{R_1}$ is added to $\boldsymbol{R_0}$. Register $\boldsymbol{R_1}$ is then updated as $\boldsymbol{R_1} \leftarrow 2\boldsymbol{R_1}$ so that at iteration i it contains $2^i\boldsymbol{P}$. The detailed algorithm is presented in Algorithm 4. It suffers from the same deficiency as the one of the left-to-right variant (Algorithm 2);

namely, it is not protected against SPA-type attacks. Again, the insertion of a dummy point addition when $k_i = 0$ can preclude these attacks. Using an additional (point) register, say $\boldsymbol{R_{-1}}$, Line 3 in Algorithm 4 can be replaced with $\boldsymbol{R_{k_i-1}} \leftarrow \boldsymbol{R_{k_i-1}} + \boldsymbol{R_1}$. But the resulting implementation is then prone to safe-error attacks. The right way to implement it is to effectively make use of *both* $\boldsymbol{R_0}$ and $\boldsymbol{R_{-1}}$ [14]. It is easily seen that in Algorithm 4 when using the dummy point addition (i.e., when Line 3 is replaced with $\boldsymbol{R_{k_i-1}} \leftarrow \boldsymbol{R_{k_i-1}} + \boldsymbol{R_1}$), register $\boldsymbol{R_{-1}}$ contains the "complementary" value of $\boldsymbol{R_0}$. Indeed, before entering iteration i, we have $\boldsymbol{R_0} = \sum_{k_j=1} 2^j \boldsymbol{P}$ and $\boldsymbol{R_{-1}} = \sum_{k_j=0} 2^j \boldsymbol{P}$, $0 \leqslant j \leqslant i-1$. As a result, we have $\boldsymbol{R_0} + \boldsymbol{R_{-1}} = \sum_{j=0}^{i-1} 2^j \boldsymbol{P} = (2^i - 1)\boldsymbol{P}$. Hence, initializing $\boldsymbol{R_{-1}}$ to $\boldsymbol{P}$, the successive values of $2^i \boldsymbol{P}$ can be equivalently obtained from $\boldsymbol{R_0} + \boldsymbol{R_{-1}}$. Summing up, the right-to-left binary method becomes

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{O}$; $\boldsymbol{R_{-1}} \leftarrow \boldsymbol{P}$; $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$
2: **for** $i = 0$ to $n - 1$ **do**
3: $\quad b \leftarrow k_i$; $\boldsymbol{R_{b-1}} \leftarrow \boldsymbol{R_{b-1}} + \boldsymbol{R_1}$
4: $\quad \boldsymbol{R_1} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_{-1}}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

Performing a point addition when $k_i = 0$ in the previous algorithm requires one more (point) register. When memory is scarce, an alternative is to rely on *Joye's double-add algorithm* [14]. As in Montgomery ladder, it always repeats a same pattern of [effective] operations and requires only two (point) registers. The algorithm is given in Algorithm 5. It corresponds to the above algorithm where $\boldsymbol{R_{-1}}$ is renamed as $\boldsymbol{R_1}$. Observe that the for-loop in the above algorithm can be rewritten into a single step as $\boldsymbol{R_{b-1}} \leftarrow \boldsymbol{R_{b-1}} + \boldsymbol{R_1} = \boldsymbol{R_{b-1}} + (\boldsymbol{R_0} + \boldsymbol{R_{-1}}) = 2\boldsymbol{R_{b-1}} + \boldsymbol{R_{-b}}$.

4 New Implementations

In [22], Meloni exploited the ZADD operation to propose scalar multiplications based on Euclidean addition chains and Zeckendorf's representation. In this section, we aim at making use of ZADD-like operations when designing scalar multiplication algorithms based on the classical binary representation. The crucial factor for implementing such an algorithm is to generate two points with the same Z-coordinate at every bit execution of scalar k.

To this end, we introduce a new operation referred to as *conjugate co-Z addition* and denoted ZADDC (for ZADD conjugate), using the efficient caching technique as described in [11, 19]. This operation evaluates $(X_3 : Y_3 : Z_3) = \boldsymbol{P} + \boldsymbol{Q} = \boldsymbol{R}$ with $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$, together with the value of $\boldsymbol{P} - \boldsymbol{Q} = \boldsymbol{S}$ where $\boldsymbol{S}$ and $\boldsymbol{R}$ share the same Z-coordinate equal to Z_3. We have $-\boldsymbol{Q} = (X_2 : -Y_2 : Z)$. Hence, letting $(\overline{X_3} : \overline{Y_3} : Z_3) = \boldsymbol{P} - \boldsymbol{Q}$, it is easily verified that $\overline{X_3} = (Y_1 + Y_2)^2 - W_1 - W_2$ and $\overline{Y_3} = (Y_1 + Y_2)(W_1 - \overline{X_3}) - A_1$, where W_1, W_2 and A_1 are computed during the course of $\boldsymbol{P} + \boldsymbol{Q}$ (cf. Algorithm 1). The additional cost for getting $\boldsymbol{P} - \boldsymbol{Q}$ from $\boldsymbol{P} + \boldsymbol{Q}$ is thus of only $1\mathsf{M} + 1\mathsf{S}$. Hence, the total cost for the ZADDC operation is of $\underline{6\mathsf{M} + 3\mathsf{S}}$. The detailed algorithm is given hereafter.

Algorithm 6 Conjugate co-Z point addition (ZADDC)

Require: $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$
Ensure: $(\boldsymbol{R}, \boldsymbol{S}) \leftarrow \text{ZADDC}(\boldsymbol{P}, \boldsymbol{Q})$ where $\boldsymbol{R} \leftarrow \boldsymbol{P} + \boldsymbol{Q} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{S} \leftarrow \boldsymbol{P} - \boldsymbol{Q} = (\overline{X_3} : \overline{Y_3} : Z_3)$

function ZADDC($\boldsymbol{P}, \boldsymbol{Q}$)
$C \leftarrow (X_1 - X_2)^2$
$W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
$D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
$X_3 \leftarrow D - W_1 - W_2$; $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$; $Z_3 \leftarrow Z(X_1 - X_2)$
$\overline{D} \leftarrow (Y_1 + Y_2)^2$
$\overline{X_3} \leftarrow \overline{D} - W_1 - W_2$; $\overline{Y_3} \leftarrow (Y_1 + Y_2)(W_1 - \overline{X_3}) - A_1$
end function

4.1 Left-to-right scalar multiplication

The main loop of Montgomery ladder (Algorithm 3) repeatedly evaluates the same two operations, namely

$$\boldsymbol{R_{1-b}} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{R_b}; \boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b} \ .$$

We explain hereafter how to efficiently carry out this computation using co-Z arithmetic for elliptic curves.

First note that $2\boldsymbol{R_b}$ can equivalently be rewritten as $(\boldsymbol{R_b} + \boldsymbol{R_{1-b}}) + (\boldsymbol{R_b} - \boldsymbol{R_{1-b}})$. So if $\boldsymbol{T}$ represents a temporary (point) register, the main loop of Montgomery ladder can be replaced with

$$\begin{aligned} &\boldsymbol{T} \leftarrow \boldsymbol{R_b} - \boldsymbol{R_{1-b}} \\ &\boldsymbol{R_{1-b}} \leftarrow \boldsymbol{R_b} + \boldsymbol{R_{1-b}}; \boldsymbol{R_b} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{T} \ . \end{aligned}$$

Suppose now that $\boldsymbol{R_b}$ and $\boldsymbol{R_{1-b}}$ share the same Z-coordinate. Using Algorithm 6, we can compute $(\boldsymbol{R_{1-b}}, \boldsymbol{T}) \leftarrow \text{ZADDC}(\boldsymbol{R_b}, \boldsymbol{R_{1-b}})$. This requires $6\mathsf{M} + 3\mathsf{S}$. At this stage, observe that $\boldsymbol{R_{1-b}}$ and $\boldsymbol{T}$ have the same Z-coordinate. Hence, we can directly apply Algorithm 1 to get $(\boldsymbol{R_b}, \boldsymbol{R_{1-b}}) \leftarrow \text{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{T})$. This requires $5\mathsf{M} + 2\mathsf{S}$. Again, observe that $\boldsymbol{R_b}$ and $\boldsymbol{R_{1-b}}$ share the same Z-coordinate at the end of the computation. The process can consequently be iterated. The total cost per bit amounts to $11\mathsf{M} + 5\mathsf{S}$ but can be reduced to $\underline{9\mathsf{M} + 7\mathsf{S}}$ (see § 4.4) by trading two (field) multiplications against two (field) squarings.

In the original Montgomery ladder, registers $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ are respectively initialized with point at infinity $\boldsymbol{O}$ and input point $\boldsymbol{P}$. Since $\boldsymbol{O}$ is the only point with its Z-coordinate equal to 0, assuming that $k_{n-1} = 1$, we start the loop counter at $i = n - 2$ and initialize $\boldsymbol{R_0}$ to $\boldsymbol{P}$ and $\boldsymbol{R_1}$ to $2\boldsymbol{P}$. It remains to ensure that the representations of $\boldsymbol{P}$ and $2\boldsymbol{P}$ have the same Z-coordinate. This is achieved thanks to the DBLU operation (see § 4.3).

Putting all together, we so obtain the following implementation of the Montgomery ladder. Remark that register $\boldsymbol{R_b}$ plays the role of temporary register $\boldsymbol{T}$.

Algorithm 7 Montgomery ladder with co-Z addition formulæ

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_{n-1} = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\boldsymbol{R_0} \leftarrow \boldsymbol{P}$; $(\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{DBLU}(\boldsymbol{R_0})$
2: **for** $i = n - 2$ down to 0 **do**
3: $\quad b \leftarrow k_i$
4: $\quad (\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{ZADDC}(\boldsymbol{R_b}, \boldsymbol{R_{1-b}})$
5: $\quad (\boldsymbol{R_b}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{R_b})$
6: **end for**
7: **return** $\boldsymbol{R_0}$

4.2 Right-to-left scalar multiplication algorithm

As noticed in [14], Joye's double-add algorithm (Algorithm 5) is to some extent the dual of the Montgomery ladder. This appears more clearly by performing the double-add operation of the main loop, $\boldsymbol{R_{1-b}} \leftarrow 2\boldsymbol{R_{1-b}} + \boldsymbol{R_b}$, in two steps as

$$\boldsymbol{T} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{R_b}; \boldsymbol{R_{1-b}} \leftarrow \boldsymbol{T} + \boldsymbol{R_{1-b}}$$

using some temporary register $\boldsymbol{T}$. If, at the beginning of the computation, $\boldsymbol{R_b}$ and $\boldsymbol{R_{1-b}}$ have the same Z-coordinate, two consecutive applications of the ZADDU algorithm allows one to evaluate the above expression with $2 \times (5\mathsf{M} + 2\mathsf{S})$. Moreover, one has to take care that $\boldsymbol{R_b}$ and $\boldsymbol{R_{1-b}}$ have the same Z-coordinate at the end of the computation in order to make the process iterative. This can be done with an additional $3\mathsf{M}$.

But there is a more efficient way to get the equivalent representation for $\boldsymbol{R_b}$. The value of $\boldsymbol{R_b}$ is unchanged during the evaluation of

$$(\boldsymbol{T}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}); (\boldsymbol{R_{1-b}}, \boldsymbol{T}) \leftarrow \mathrm{ZADDU}(\boldsymbol{T}, \boldsymbol{R_{1-b}})$$

and thus $\boldsymbol{R_b} = \boldsymbol{T} - \boldsymbol{R_{1-b}}$ —where $\boldsymbol{R_{1-b}}$ is the initial input value. The latter ZADDU operation can therefore be replaced with a ZADDC operation; i.e.,

$$(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{ZADDC}(\boldsymbol{T}, \boldsymbol{R_{1-b}})$$

to get the expected result. The advantage of doing so is that $\boldsymbol{R_b}$ and $\boldsymbol{R_{1-b}}$ have the same Z-coordinate without additional work. This yields a total cost per bit of $11\mathsf{M} + 5\mathsf{S}$ for the main loop.

It remains to ensure that registers $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ are initialized with points sharing the same Z-coordinate. For the Montgomery ladder, we assumed that k_{n-1} was equal to 1. Here, we will assume that k_0 is equal to 1 to avoid to deal with the point at infinity. This condition can be automatically satisfied using certain DPA-type countermeasures (see § 5.2). Alternative strategies are described in [14]. The value $k_0 = 1$ leads to $\boldsymbol{R_0} \leftarrow \boldsymbol{P}$ and $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$. The two registers have obviously the same Z-coordinate but are not different. The trick is to start the loop counter at $i = 2$ and to initialize $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ according the bit value of k_1. If $k_1 = 0$ we end up with $\boldsymbol{R_0} \leftarrow \boldsymbol{P}$ and $\boldsymbol{R_1} \leftarrow 3\boldsymbol{P}$, and conversely if

$k_1 = 1$ with $\boldsymbol{R_0} \leftarrow 3\boldsymbol{P}$ and $\boldsymbol{R_1} \leftarrow \boldsymbol{P}$. The TPLU operation (see § 4.3) ensures that this is done so that the Z-coordinates are the same.

The complete resulting algorithm is depicted below. As for our implementation of the Montgomery ladder (Algorithm 7), remark that temporary register $\boldsymbol{T}$ is played by register $\boldsymbol{R_b}$.

Algorithm 8 Joye's double-add algorithm with co-Z addition formulæ

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $b \leftarrow k_1$; $\boldsymbol{R_b} \leftarrow \boldsymbol{P}$; $(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{TPLU}(\boldsymbol{R_b})$
2: **for** $i = 2$ to $n - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad (\boldsymbol{R_b}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{R_b})$
5: $\quad (\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{ZADDC}(\boldsymbol{R_b}, \boldsymbol{R_{1-b}})$
6: **end for**
7: **return** $\boldsymbol{R_0}$

It is striking to see the resemblance (or duality) between Algorithm 7 and Algorithm 8: they involve the same co-Z operations (but in reverse order) and scan scalar k in reverse directions.

4.3 Point doubling and tripling

Algorithms 7 and 8 respectively require a point doubling and a point tripling operation updating the input point. We describe how this can be implemented.

Initial Doubling Point We have seen in Section 2 that the double of point $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$ can be obtained with $1\mathsf{M} + 8\mathsf{S} + 1\mathsf{c}$. By setting $Z_1 = 1$, the cost drops to $1\mathsf{M} + 5\mathsf{S}$:

$$\mathrm{X}(2\boldsymbol{P}) = M^2 - 2S, \quad \mathrm{Y}(2\boldsymbol{P}) = M(S - \mathrm{X}(2\boldsymbol{P})) - 8L, \quad \mathrm{Z}(2\boldsymbol{P}) = 2Y_1$$

with $M = 3B + a$, $S = 2((X_1 + E)^2 - B - L)$, $L = E^2$, $B = {X_1}^2$, and $E = {Y_1}^2$. Since $Z(2\boldsymbol{P}) = 2Y_1$, it follows that

$$(S : 8L : Z(2\boldsymbol{P})) \sim \boldsymbol{P} \quad \text{with } S = 4X_1{Y_1}^2 \text{ and } L = {Y_1}^4$$

is an equivalent representation for point $\boldsymbol{P}$. Updating point $\boldsymbol{P}$ such that its Z-coordinate is equal to that of $2\boldsymbol{P}$ comes thus for free. We let $(2\boldsymbol{P}, \tilde{\boldsymbol{P}}) \leftarrow \mathrm{DBLU}(\boldsymbol{P})$ denote the corresponding operation, where $\tilde{\boldsymbol{P}} \sim \boldsymbol{P}$ and $\mathrm{Z}(\tilde{\boldsymbol{P}}) = \mathrm{Z}(2\boldsymbol{P})$. The cost of DBLU operation (doubling with update) is $\underline{1\mathsf{M} + 5\mathsf{S}}$.

Initial Tripling Point The triple of $\boldsymbol{P} = (X_1 : Y_1 : 1)$ can be evaluated as $3\boldsymbol{P} = \boldsymbol{P} + 2\boldsymbol{P}$ using co-Z arithmetic [20]. From $(2\boldsymbol{P}, \tilde{\boldsymbol{P}}) \leftarrow \mathrm{DBLU}(\boldsymbol{P})$, this can be obtained as $\mathrm{ZADDU}(\tilde{\boldsymbol{P}}, 2\boldsymbol{P})$ with $5\mathsf{M} + 2\mathsf{S}$ and no additional cost to update $\boldsymbol{P}$ for its Z-coordinate becoming equal to that of $3\boldsymbol{P}$. The corresponding operation, tripling with update, is denoted $\mathrm{TPLU}(\boldsymbol{P})$ and its total cost is of $\underline{6\mathsf{M} + 7\mathsf{S}}$.

4.4 Combined double-add operation

A point doubling-addition is the evaluation of $\boldsymbol{R} = 2\boldsymbol{P} + \boldsymbol{Q}$. This can be done in two steps as $\boldsymbol{T} \leftarrow \boldsymbol{P} + \boldsymbol{Q}$ followed by $\boldsymbol{R} \leftarrow \boldsymbol{P} + \boldsymbol{T}$. If $\boldsymbol{P}$ and $\boldsymbol{Q}$ have the same Z-coordinate, this requires 10M + 4S by two consecutive applications of the ZADDU function (Algorithm 1).

Things are slightly more complex if we wish that $\boldsymbol{R}$ and $\boldsymbol{Q}$ share the same Z-coordinate at the end of the computation. But if we compare the original Joye's double-add algorithm (Algorithm 5) and the corresponding algorithm we got using co-Z arithmetic (Algorithm 8), this is actually what is achieved. We can compute $(\boldsymbol{T}, \boldsymbol{P}) \leftarrow \text{ZADDU}(\boldsymbol{P}, \boldsymbol{Q})$ followed by $(\boldsymbol{R}, \boldsymbol{Q}) \leftarrow \text{ZADDC}(\boldsymbol{T}, \boldsymbol{P})$. We let $(\boldsymbol{R}, \boldsymbol{Q}) \leftarrow \text{ZDAU}(\boldsymbol{P}, \boldsymbol{Q})$ denote the corresponding operation (ZDAU stands for *co-Z double-add with update*).

Algorithmically, we have:

1: $C' \leftarrow (X_1 - X_2)^2$
2: $W'_1 \leftarrow X_1 C'$; $W'_2 \leftarrow X_2 C'$
3: $D' \leftarrow (Y_1 - Y_2)^2$; $A'_1 \leftarrow Y_1(W'_1 - W'_2)$
4: $X'_3 \leftarrow D' - W'_1 - W'_2$; $Y'_3 \leftarrow (Y_1 - Y_2)(W'_1 - X'_3) - A'_1$; $Z'_3 \leftarrow Z(X_1 - X_2)$
5: $X_1 \leftarrow W'_1$; $Y_1 \leftarrow A'_1$; $Z_1 \leftarrow Z'_3$
6: $C \leftarrow (X'_3 - X_1)^2$
7: $W_1 \leftarrow X'_3 C$; $W_2 \leftarrow X_1 C$
8: $D \leftarrow (Y'_3 - Y_1)^2$; $A_1 \leftarrow Y'_3(W_1 - W_2)$
9: $X_3 \leftarrow D - W_1 - W_2$; $Y_3 \leftarrow (Y'_3 - Y_1)(W_1 - X_3) - A_1$; $Z_3 \leftarrow Z'_3(X'_3 - X_1)$
10: $\overline{D} \leftarrow (Y'_3 + Y_1)^2$
11: $X_2 \leftarrow \overline{D} - W_1 - W_2$; $Y_2 \leftarrow (Y'_3 + Y_1)(W_1 - X_2) - A_1$; $Z_2 \leftarrow Z_3$

A close inspection of the above algorithm shows that two (field) multiplications can be traded against two (field) squarings. Indeed, with the same notations, we have:

$$2Y'_3 = (Y_1 - Y_2 + W'_1 - X'_3)^2 - D' - C - 2A'_1 \ .$$

Also, we can skip the intermediate computation of $Z'_3 = Z(X_1 - X_2)$ and obtain directly $2Z_3 = 2Z(X_1 - X_2)(X'_3 - X_1)$ as

$$2Z_3 = Z\Big((X_1 - X_2 + X'_3 - X_1)^2 - C' - C\Big) \ .$$

These modifications (in Lines 4 and 9) require some rescaling. For further optimization, some redundant or unused variables are suppressed. The resulting algorithm is detailed hereafter (Algorithm 9). It clearly appears that the ZDAU operation only requires $\underline{9\mathsf{M} + 7\mathsf{S}}$.

Algorithm 9 Co-Z point doubling-addition with update (ZDAU)

Require: $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$
Ensure: $(\boldsymbol{R}, \boldsymbol{Q}) \leftarrow \text{ZDAU}(\boldsymbol{P}, \boldsymbol{Q})$ where $\boldsymbol{R} \leftarrow 2\boldsymbol{P} + \boldsymbol{Q} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{Q} \leftarrow (\lambda^2 X_2 : \lambda^3 Y_2 : Z_3)$ with $Z_3 = \lambda Z$ for some $\lambda \neq 0$

function ZDAU($\boldsymbol{P}, \boldsymbol{Q}$)
$C' \leftarrow (X_1 - X_2)^2$
$W_1' \leftarrow X_1 C'$; $W_2' \leftarrow X_2 C'$
$D' \leftarrow (Y_1 - Y_2)^2$; $A_1' \leftarrow Y_1(W_1' - W_2')$
$\hat{X}_3' \leftarrow D' - W_1' - W_2'$
$C \leftarrow (\hat{X}_3' - W_1')^2$
$Y_3' \leftarrow [(Y_1 - Y_2) + (W_1' - \hat{X}_3')]^2 - D' - C - 2A_1'$
$W_1 \leftarrow 4\hat{X}_3' C$; $W_2 \leftarrow 4W_1' C$
$D \leftarrow (Y_3' - 2A_1')^2$; $A_1 \leftarrow Y_3'(W_1 - W_2)$
$X_3 \leftarrow D - W_1 - W_2$; $Y_3 \leftarrow (Y_3' - 2A_1')(W_1 - X_3) - A_1$
$Z_3 \leftarrow Z\big((X_1 - X_2 + \hat{X}_3' - W_1')^2 - C' - C\big)$
$\overline{D} \leftarrow (Y_3' + 2A_1')^2$
$X_2 \leftarrow \overline{D} - W_1 - W_2$; $Y_2 \leftarrow (Y_3' + 2A_1')(W_1 - X_2) - A_1$; $Z_2 \leftarrow Z_3$
end function

The combined ZDAU operation immediately gives rise to an alternative implementation of Joye's double-add algorithm (Algorithm 5). Compared to our first implementation (Algorithm 8), the cost per bit amounts to $\underline{9\mathsf{M} + 7\mathsf{S}}$ (instead of $11\mathsf{M} + 5\mathsf{S}$).

Algorithm 10 Joye's double-add algorithm with co-Z addition formulæ (II)

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $b \leftarrow k_1$; $\boldsymbol{R}_b \leftarrow \boldsymbol{P}$; $(\boldsymbol{R}_{\mathbf{1}-b}, \boldsymbol{R}_b) \leftarrow \text{TPLU}(\boldsymbol{R}_b)$
2: **for** $i = 2$ to $n - 1$ **do**
3: $\quad b \leftarrow k_i$
4: $\quad (\boldsymbol{R}_{\mathbf{1}-b}, \boldsymbol{R}_b) \leftarrow \text{ZDAU}(\boldsymbol{R}_{\mathbf{1}-b}, \boldsymbol{R}_b)$
5: **end for**
6: **return** $\boldsymbol{R}_{\mathbf{0}}$

Similar savings can be obtained for our implementation of the Montgomery ladder (i.e., Algorithm 7). However, as the ZADDU and ZADDC operations appear in reverse order, it is more difficult to handle. It is easy to trade $1\mathsf{M}$ against $1\mathsf{S}$. In order to trade $2\mathsf{M}$ against $2\mathsf{S}$, one has to consider two bits of scalar k at a time so as to allow to have the ZADDC operation performed prior to the ZADDU operation. The two previous M/S trade-offs can then be applied.

5 Discussion

5.1 Performance analysis

Table 1 summarizes the cost of different types of addition and doubling-addition formulæ on elliptic curves. Each type of formula presents its own advantages depending on the coordinate system and the underlying scalar multiplication algorithm. Symbols $\mathcal{J}$ and $\mathcal{A}$ respectively stand for Jacobian coordinates and affine coordinates.

Table 1. Performance comparison of addition and doubling-addition formulæ

Operation	Notation	System	Cost
Point addition:			
– General addition [2]	ADD	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	11M + 5S
– Co-Z addition [22]	ZADD	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	5M + 2S
– Co-Z addition with update [22][a]	ZADDU	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	5M + 2S
– General conjugate addition [19]	ADDC	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	12M + 6S
– Conjugate co-Z addition (Alg. 6)	ZADDC	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	6M + 3S
Point doubling-addition:			
– General doubling-addition [18]	DA	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	13M + 8S
– Mixed doubling-addition [20]	mDA	$(\mathcal{J},\mathcal{A}) \to \mathcal{J}$	11M + 7S
– Co-Z doubling-addition with update (Alg. 9)	ZDAU	$(\mathcal{J},\mathcal{J}) \to \mathcal{J}$	9M + 7S

[a] See also Algorithm 1.

For the sake of comparison, we consider the typical ratio S/M = 0.8. Similar results can easily be derived for other ratios. We see that the co-Z addition (with or without update) improves the general addition by a speed-up factor of 56%. Almost as well, our conjugate co-Z addition formula improves the general conjugate addition by a factor of 50%. For the doubling-addition operations, our co-Z formula (including the update) is always faster; it is even faster than the best mixed doubling-addition formula. It yields a respective speed-up factor of 25% and of 12% compared to the general doubling-addition and to the mixed doubling-addition. In addition to speed, our new formulæ are also very efficient memory-wise. See [12, Appendix A] for detailed register allocations.

Table 2 compares the performance of our co-Z implementations with previous ones. Our improved right-to-left co-Z scalar multiplication algorithm (i.e., Algorithm 10) requires 9M + 7S per bit of scalar k. An application of Joye's double-add algorithm with the best doubling-addition (DA) formula [18] requires 13M + 8S per bit. Hence, with the usual ratio S/M = 0.8, our co-Z version of Joye's double-add algorithm yields a speed-up factor of 25%.

Furthermore, our left-to-right co-Z algorithm (i.e., Algorithm 7 as modified in § 4.4) offers a speed competitive with known implementations of Montgomery

Table 2. Performance comparison of scalar multiplication algorithm

Algorithm	Operations	Cost per bit
Joye's double-add algorithm [14]: R → L		
– Basic version	DA	13M + 8S
– Co-Z version (Algorithm 10)	ZDAU	9M + 7S
Montgomery ladder [24]: L → R		
– Basic version	DBL and ADD	14M + 10S[a]
– *X*-only version [5, 9, 13]	XDBL and XADD	9M + 7S[b]
– Co-Z version (Algorithm 7)	ZADDC and ZADDU	9M + 7S[c]

[a] The cost assumes that curve parameter a is equal to -3. This allows the use of the faster point doubling formula: 3M + 5S instead of 1M + 8S + 1c; cf. Section 2.

[b] The cost assumes that multiplications by curve parameter a are negligible; e.g., $a = -3$. It also assumes that input point $\boldsymbol{P}$ is given in affine coordinates; i.e., $Z(\boldsymbol{P}) = 1$. See [12, Appendix B] for a detailed implementation.

[c] With the improvements mentioned in § 4.4. The direct implementation of Algorithm 7 has a cost of 11M + 5S per bit.

ladder for *general*† elliptic curves. It only requires 9M + 7S per bit of scalar k. Moreover, we note that this cost is *independent* of the curve parameters.

5.2 Security considerations

As explained in Section 3, Montgomery ladder and Joye's double-add algorithm are naturally protected against SPA-type attacks and safe-error attacks. Since our implementations are built on them and maintain the same regular pattern of instructions without using dummy instructions, they inherit of the same security features. Moreover, our proposed co-Z versions (i.e., Algorithms 7, 8 and 10) can be protected against DPA-type attacks; cf. [1, Chapter 29] for several methods.

Yet another important class of attacks against implementations are the fault attacks [3, 4]. An additional advantage of Algorithm 7 (and of Algorithms 8 and 10) is that it is easy to assess the correctness of the computation by checking whether the output point belongs to the curve. We remark that the *X*-only versions of Montgomery ladder ([5, 9, 13]) do not permit it and so may be subject to (regular) fault attacks, as was demonstrated in [10].

6 Conclusion

Co-Z arithmetic as developed by Meloni provides an extremely fast point addition formula. So far, their usage for scalar multiplication algorithms was confined to Euclidean addition chains and the Zeckendorf's representation. In this

† Montgomery introduced in [24] a curve shape that nicely combines with the *X*-only ladder, leading to a better cost per bit. But this shape does not cover all classes of elliptic curves. In particular, it does not apply to NIST recommended curves [25, Appendix D].

paper, we developed new strategies and proposed a co-Z conjugate point addition formula as well as other companion co-Z formulæ. The merit of our approach resides in that the fast co-Z arithmetic nicely combines with certain binary ladders. Specifically, we applied co-Z techniques to Montgomery ladder and Joye's double-add algorithm. The so-obtained implementations are efficient and protected against a variety of implementation attacks. All in all, the implementations presented in this paper constitute a method of choice for the efficient yet secure implementation of elliptic curve cryptography in embedded systems or other memory-constrained devices.

As a side result, this paper also proposed the fastest point doubling-addition formula.

Acknowledgments The authors would like to thank Jean-Luc Beuchat, Francisco Rodrìguez Henrìquez, Patrick Longa, and Francesco Sica for helpful discussions. We would also like to thank the anonymous referees for their useful comments. In particular, we thank the referee pointing out that the cost with the Montgomery ladder can be reduced to $9\mathsf{M} + 7\mathsf{S}$ per bit for general elliptic curves.

References

1. R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. CRC Press, 2005.
2. D. J. Bernstein and T. Lange. Explicit-formulas database. `http://www.hyperelliptic.org/EFD/jacobian.html`.
3. I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In M. Bellare, editor, *Advances in Cryptology – CRYPTO 2000*, volume 1880 of *LNCS*, pages 131–146. Springer, 2000.
4. D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2):110–119, 2001. Extended abstract in Proc. of EUROCRYPT '97.
5. E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography (PKC 2002)*, volume 2274 of *LNCS*, pages 335–345. Springer, 2002.
6. D. V. Chudnovsky and G. V. Chudnovsky. Sequences of numbers generated by addition in formal groups and new primality and factorization tests. *Advances in Applied Mathematics*, 7(4):385–434, 1986.
7. H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Advances in Cryptology – ASIACRYPT '98*, volume 1514 of *LNCS*, pages 51–65. Springer, 1998.
8. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *LNCS*, pages 292–302. Springer, 1999.
9. W. Fischer, C. Giraud, E. W. Knudsen, and J.-P. Seifert. Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks. Cryptology ePrint Archive, Report 2002/007, 2002. `http://eprint.iacr.org/`.

10. P.-A. Fouque, R. Lercier, D. Réal, and F. Valette. Fault attack on elliptic curve Montgomery ladder implementation. In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography (FDTC 2008)*, pages 92–98. IEEE Computer Society, 2008.
11. S. Galbraith, X. Lin, and M. Scott. A faster way to do ECC. Presented at 12th Workshop on Elliptic Curve Cryptography (ECC 2008), Utrecht, The Netherlands, Sept. 22–24, 2008. Slides available at URL `http://www.hyperelliptic.org/tanja/conf/ECC08/slides/Mike-Scott.pdf`.
12. R. R. Goundar, M. Joye, and A. Miyaji. Co-*Z* addition formulæ and binary ladders on elliptic curves. Cryptology ePrint Archive, Report 2010/353, 2010. `http://eprint.iacr.org/`.
13. T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography (PKC 2002)*, volume 2274 of *LNCS*, pages 280–296. Springer, 2002.
14. M. Joye. Highly regular right-to-left algorithms for scalar multiplication. In P. Paillier and I. Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems – CHES 2007*, volume 4727 of *LNCS*, pages 135–147. Springer, 2007.
15. M. Joye and S.-M. Yen. The Montgomery powering ladder. In B. S. Kaliski Jr. et al., editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *LNCS*, pages 291–302. Springer, 2003.
16. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, 1987.
17. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *LNCS*, pages 388–397. Springer, 1999.
18. P. Longa. ECC Point Arithmetic Formulae (EPAF). `http://patricklonga.bravehost.com/Jacobian.html`.
19. P. Longa and C. H. Gebotys. Novel precomputation schemes for elliptic curve cryptosystems. In M. Abdalla et al., editors, *Applied Cryptography and Network Security (ACNS 2009)*, volume 5536 of *LNCS*, pages 71–88. Springer, 2009.
20. P. Longa and A. Miri. New composite operations and precomputation for elliptic curve cryptosystems over prime fields. In R. Cramer, editor, *Public Key Cryptography – PKC 2008*, volume 4939 of *LNCS*, pages 229–247. Springer, 2008.
21. J. López and R. Dahab. Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *LNCS*, pages 316–327. Springer, 1999.
22. N. Meloni. New point addition formulæ for ECC applications. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *LNCS*, pages 189–201. Springer, 2007.
23. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – CRYPTO '85*, volume 218 of *LNCS*, pages 417–426. Springer, 1985.
24. P. L. Montgomery. Speeding up the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
25. National Institute of Standards and Technology. Digital Signature Standard (DSS). Federal Information Processing Standards Publication, FIPS PUB 186-3, June 2009.
26. S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.
27. S.-M. Yen, S. Kim, S. Lim, and S.-J. Moon. A countermeasure against one physical cryptanalysis may benefit another attack. In K. Kim, editor, *Information Security and Cryptology – ICISC 2001*, volume 2288 of *LNCS*, pages 414–427. Springer, 2002.