Journal of Cryptographic Engineering manuscript No.
(will be inserted by the editor)

Scalar Multiplication on Weierstraß Elliptic Curves from Co-Z Arithmetic

Raveen R. Goundar · Marc Joye · Atsuko Miyaji · Matthieu Rivain · Alexandre Venelli

the date of receipt and acceptance should be inserted later

Abstract In 2007, Meloni 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 (and register allocations) 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, Joye's double-add algorithm, and certain signed-digit algorithms, as well as faster (X, Y)-only variants for left-to-right versions. Further, the proposed implementations are regular, thereby offering a natural protection against a variety of implementation attacks.

Raveen R. Goundar
Independent researcher
P.O. Box 794, Ba, Fiji Islands
E-mail: raveen.rg@gmail.com

Marc Joye
Technicolor, Security & Content Protection Labs
1 av. de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
E-mail: marc.joye@technicolor.com

Atsuko Miyaji
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
E-mail: miyaji@jaist.ac.jp

Matthieu Rivain
CryptoExperts
41 Boulevard des Capucines, 75002 Paris, France
E-mail: matthieu.rivain@cryptoexperts.com

Alexandre Venelli
Inside Secure
Avenue Victoire, 13790 Rousset, France
E-mail: avenelli@insidefr.com

Keywords Elliptic curves · Meloni's technique · Jacobian coordinates · regular ladders · implementation attacks · embedded systems

1 Introduction

Elliptic curve cryptography (ECC), introduced independently by Koblitz [22] and Miller [29] 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, or non-standard scalar representations. See e.g. [2,5] for a survey of some techniques.

In this paper, we focus on scalar multiplication algorithms based on *co-Z arithmetic*. Co-Z arithmetic was introduced by Meloni in [28] as a means to efficiently add two projective points sharing the same Z-coordinate. The original co-Z addition formula of [28] greatly improves on the general point addition. The drawback is that this fast formula is by construction restricted to Euclidean addition chains (i.e., addition chains without doubling). The efficiency being depen-

dent 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" method. The resulting algorithm is efficient but still slower than its binary counterparts. Subsequent papers were published that show how to efficiently apply co-Z arithmetic to binary ladders from a *conjugate* co-Z addition formula [16,33]. Co-Z *left-to-right* binary algorithms making use of X- and Y-coordinates only were also proposed, leading to additional speed-ups [33,32]. This paper surveys these scalar multiplication algorithms and discusses their performance for various settings. Specifically, we describe efficient co-Z based versions of Montgomery ladder, Joye's double-add algorithm, and zeroless signed-digit algorithms. All these algorithms are *highly regular*, which make them naturally protected against SPA-type attacks [23] and safe-error attacks [34,35]. Moreover, they can be combined with other known countermeasures to protect them against further classes of attacks.

This paper only deals with *general* elliptic curves. We note that elliptic curves with special forms exist (including Montgomery curves, Edwards curves, Hessian curves, ...) which have performance advantages over general elliptic curves (see [3]). However, many applications require the compliance with arbitrarily chosen elliptic curves, which motivates the investigation of efficient scalar multiplication algorithms for general, form-free elliptic curves.

2 Preliminaries

Let $\mathbb{F}_q$ be a finite field of characteristic $\neq 2, 3$. Consider an elliptic curve E over $\mathbb{F}_q$ given by the Weierstraß equation $y^2 = x^3 + ax + b$, where $a, b \in \mathbb{F}_q$ and with discriminant $\Delta := -16(4a^3 + 27b^2) \neq 0$. This section explains how to get efficient arithmetic on elliptic curves over $\mathbb{F}_q$.

Point addition formulæ are based on different operations over $\mathbb{F}_q$ (multiplication, inversion, addition, and subtraction), which have different computational costs. In this paper, we denote by I, M, and S the cost of a field inversion, of a field multiplication, and of a field squaring, respectively. Typically, when q is a large prime, it is often assumed that (i) $\mathsf{I} \approx 100\mathsf{M}$, (ii) $\mathsf{S} = 0.8\mathsf{M}$, and (iii) the cost of field additions can be neglected. These assumptions are derived from the usual software implementations for field operations. When the latter are based on a hardware co-processor —as it is often the case in embedded systems— their costs become architecture-reliant. In general, a field inversion always costs a few dozens of multiplications, the cost of a field squaring is of the same order as that of a field multiplication (possibly a bit cheaper), and the cost of a field addition is clearly lower (although not always negligible).

Throughout the paper, the computational cost will be expressed as the number of I, M, and S. The various presented algorithms will be optimized so as to minimize the number of these operations. Moreover, whenever possible, a M will be traded against a S, usually at the expense of additional field additions. Of course, when field additions are costly or when field squarings are not faster than field multiplications, our algorithms can be adapted so as to get the best efficiency.

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 $\boldsymbol{O}$. We have $\boldsymbol{P} + \boldsymbol{O} = \boldsymbol{O} + \boldsymbol{P} = \boldsymbol{P}$ for any point $\boldsymbol{P}$ on E. Let now $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$ 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 $I_2 = {Z_2}^2$ [10].[1] 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

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

[1] Actually, with common-subexpression elimination, the formulæ reported by Cohen et al. in [10] requires $12\mathsf{M} + 4\mathsf{S}$. 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 [3, 24].

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 $N = {Z_1}^2$ [3]. 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 $a = -3$ [9], in which case point doubling costs $3\mathsf{M} + 5\mathsf{S}$. In the general case, point doubling can be sped up by representing points $(X_i : Y_i : Z_i)$ with an additional coordinate, namely $T_i = aZ_i{}^4$. This extended representation is referred to as *modified Jacobian coordinates* [10]. The cost of point doubling drops to $3\mathsf{M} + 5\mathsf{S}$ at the expense of a slower point addition.

Detailed formulæ are offered in [3]; see also [18] for memory usage.

2.2 Co-Z point addition

In [28], 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 $A_1 = Y_1(W_1 - W_2)$, $W_1 = X_1C$, $W_2 = X_2C$, $C = (X_1 - X_2)^2$ and $D = (Y_1 - Y_2)^2$. This operation is referred to as 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

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

The corresponding operation is denoted ZADDU (i.e., ZADD with update) and is presented in Alg. 1. It is readily seen that it requires $\underline{5\mathsf{M} + 2\mathsf{S}}$. Moreover, as detailed in Alg. 19 (Appendix C), only 6 field registers are required.

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.

Algorithm 1 Co-Z 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 $Z_3 = \lambda Z$ for some $\lambda \neq 0$

1: **function** ZADDU($\boldsymbol{P}, \boldsymbol{Q}$)
2: $C \leftarrow (X_1 - X_2)^2$
3: $W_1 \leftarrow X_1C$; $W_2 \leftarrow X_2C$
4: $D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
5: $X_3 \leftarrow D - W_1 - W_2$
6: $Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
7: $Z_3 \leftarrow Z(X_1 - X_2)$
8: $X_1 \leftarrow W_1$; $Y_1 \leftarrow A_1$; $Z_1 \leftarrow Z_3$
▷ $\boldsymbol{R} = (X_3 : Y_3 : Z_3)$, $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$
9: **end function**

3.1 Left-to-right methods

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 Alg. 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** $\boldsymbol{R_0}$

Although efficient (in both memory and computation), the left-to-right binary method may be subject to SPA-type attacks [23]. From a power trace, an adversary able to distinguish between point doublings and point additions can 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 Alg. 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* [11], now appears as a regular succession of a point doubling fol-

lowed by a point addition. Unfortunately, it now becomes subject to safe-error attacks [34,35]. 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).

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: $\quad b \leftarrow k_i$; $\boldsymbol{R_{1-b}} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{R_b}$
4: $\quad \boldsymbol{R_b} \leftarrow 2\boldsymbol{R_b}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

A scalar multiplication algorithm featuring a regular structure without dummy operation is the so-called *Montgomery ladder* [30] (see also [21]). It is detailed in Alg. 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 [30] (see also [7,12,19,27]).

3.2 Right-to-left methods

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 have $k\boldsymbol{P} = \sum_{k_i=1} 2^i \boldsymbol{P}$. 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 given hereafter.

It suffers from the same deficiency as the one of the left-to-right variant (Alg. 2); namely, it is not protected against SPA-type attacks. Again, the insertion

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: $\quad$ **if** $(k_i = 1)$ **then** $\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + \boldsymbol{R_1}$
4: $\quad \boldsymbol{R_1} \leftarrow 2\boldsymbol{R_1}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

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 Alg. 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 $\boldsymbol{R_{-1}}$ [20]. It is easily seen that in Alg. 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* [20]. As in Montgomery ladder, it always repeats a same pattern of effective operations and requires only two (point) registers. The algorithm is given in Alg. 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 $\boldsymbol{R_{b-1}} \leftarrow \boldsymbol{R_{b-1}} + \boldsymbol{R_1} = \boldsymbol{R_{b-1}} + (\boldsymbol{R_0} + \boldsymbol{R_{-1}}) = 2\boldsymbol{R_{b-1}} + \boldsymbol{R_{-b}}$.

3.3 Signed-digit methods

Noting that subtracting boils down to adding the additive inverse, the binary methods (Algs. 2 and 4) easily extend to signed-digit representations, that is, when scalar k is represented with digits in the set $\{-1, 0, 1\}$. The resulting methods are well adapted to the elliptic curve setting since the computation of an inverse is

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: $\quad b \leftarrow k_i$
4: $\quad \boldsymbol{R_{1-b}} \leftarrow 2\boldsymbol{R_{1-b}} + \boldsymbol{R_b}$
5: **end for**
6: **return** $\boldsymbol{R_0}$

a cheap operation on elliptic curves. As a reminder, if $\boldsymbol{P} = (X_1 : Y_1 : Z_1)$ then $-\boldsymbol{P} = (X_1 : -Y_1 : Z_1)$. Signed-digit representations are not unique. Among them, we note the non-adjacent form (NAF), which is often used as it has an average density of non-zero digits of only 1/3 [31]. For our purposes, in order to prevent SPA-type attacks, we rather consider what we call the *zeroless signed-digit expansion* (ZSD). Given an odd integer k, we express it with digits in $\{-1, 1\}$ (i.e., without the zero digit).

The ZSD expansion can be obtained "on-the-fly" from the binary expansion. Let $k = \sum_{i=0}^{n-1} k_i\, 2^i$ where $k_i \in \{0, 1\}$ and $k_0 = 1$ (i.e., k is assumed odd). We observe that for every $w > 1$, we have $1 = 2^w - \sum_{j=0}^{w-1} 2^j$. It follows that any group of w bits $00\ldots01$ in the binary expansion of k can be equivalently replaced with the group of w signed digits $1\bar{1}\bar{1}\ldots\bar{1}$ (where $\bar{1} = -1$). The ZSD expansion of an odd integer k, $k = \sum_{i=0}^{n-1} \kappa_i\, 2^i$ with $\kappa_i \in \{-1, 1\}$, is therefore given by

$$\begin{cases} \kappa_{n-1} = 1\,, \\ \kappa_i = (-1)^{1+k_{i+1}} \quad \text{for } n-2 \geqslant i \geqslant 0\ . \end{cases}$$

We so obtain the two following algorithms for evaluating the scalar multiplication $\boldsymbol{Q} = k\boldsymbol{P}$. Algorithm 6 processes scalar k from the left to the right while Algorithm 7 processes it from the right to the left.

Algorithm 6 Left-to-right signed-digit method

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

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

4 Basic Algorithms with Co-Z Formulæ

In [28], 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 algorithms is to generate two points with the same Z-coordinate at every bit execution of scalar k.

Algorithm 7 Right-to-left signed-digit method

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_1, k_0)_2 \in \mathbb{N}$ with $k_0 = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

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

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 described in [14,25]. 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. Alg. 1). The additional cost for getting $\boldsymbol{P} - \boldsymbol{Q}$ from $\boldsymbol{P} + \boldsymbol{Q}$ is thus of only $1\mathsf{M} + 1\mathsf{S}$. The resulting algorithm is presented in Alg. 8. The total cost for the ZADDC operation is of $\underline{6\mathsf{M} + 3\mathsf{S}}$ and requires 7 field registers; see Alg. 20 (Appendix C).

Algorithm 8 Conjugate co-Z 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 $\boldsymbol{S} \leftarrow \boldsymbol{P} - \boldsymbol{Q} = (\overline{X_3} : \overline{Y_3} : Z_3)$

1: **function** ZADDC($\boldsymbol{P}, \boldsymbol{Q}$)
2: $\quad C \leftarrow (X_1 - X_2)^2$
3: $\quad W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
4: $\quad D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
5: $\quad X_3 \leftarrow D - W_1 - W_2$
6: $\quad Y_3 \leftarrow (Y_1 - Y_2)(W_1 - X_3) - A_1$
7: $\quad Z_3 \leftarrow Z(X_1 - X_2)$
8: $\quad \overline{D} \leftarrow (Y_1 + Y_2)^2$
9: $\quad \overline{X_3} \leftarrow \overline{D} - W_1 - W_2$
10: $\quad \overline{Y_3} \leftarrow (Y_1 + Y_2)(W_1 - \overline{X_3}) - A_1$
$\quad\quad \triangleright\ \boldsymbol{R} = (X_3 : Y_3 : Z_3)$, $\boldsymbol{S} = (\overline{X_3} : \overline{Y_3} : Z_3)$
11: **end function**

In the following, we describe several scalar multiplication algorithms based on ZADDU and ZADDC op-

erations. We further note Jac2aff the algorithm that converts the Jacobian coordinates of a point into its affine coordinates, the cost of which is $\underline{1\mathsf{I}+3\mathsf{M}+1\mathsf{S}}$.

4.1 Left-to-right algorithms

The main loop of Montgomery ladder (Alg. 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 8, we can compute $(\boldsymbol{R_{1-b}}, \boldsymbol{T}) \leftarrow \mathrm{ZADDC}(\boldsymbol{R_b}, \boldsymbol{R_{1-b}})$. 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 $(\boldsymbol{R_b}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{T})$. 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 § 5.1) 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 § 4.4).

Putting all together, we obtain the implementation depicted in Alg. 9 for the Montgomery ladder. Remark that register $\boldsymbol{R_b}$ plays the role of temporary register $\boldsymbol{T}$.

4.2 Right-to-left algorithms

As noticed in [20], Joye's double-add algorithm (Alg. 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

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

Algorithm 9 Montgomery ladder with co-Z addition formulæ

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

1: $(\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{DBLU}(\boldsymbol{P})$
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** $\mathrm{Jac2aff}(\boldsymbol{R_0})$

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

$$\begin{aligned}&(\boldsymbol{T}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_{1-b}}, \boldsymbol{R_b})\\ &(\boldsymbol{R_{1-b}}, \boldsymbol{T}) \leftarrow \mathrm{ZADDU}(\boldsymbol{T}, \boldsymbol{R_{1-b}})\end{aligned}$$

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 § 6.1). Alternative strategies are described in [20]. 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 § 4.4) ensures that this is done so that the Z-coordinates are the same.

The complete algorithm is depicted in Alg. 10. As for our implementation of the Montgomery ladder (i.e.,

Alg. 9), remark that temporary register $\boldsymbol{T}$ is played by register $\boldsymbol{R_b}$.

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

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

1: $b \leftarrow k_1$; $(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{TPLU}(\boldsymbol{P})$
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** $\mathrm{Jac2aff}(\boldsymbol{R_0})$

It is striking to see the resemblance (or duality) between Algorithm 9 and Algorithm 10: they involve the same co-Z operations (but in reverse order) and scan scalar k in reverse directions.

4.3 Signed-digit algorithms

A similar observation can be drawn for the signed-digit algorithms and their unsigned counterparts. If we compare Algorithm 6 with Algorithm 5, we see that they scan scalar k in reverse directions and respectively repeat the operations $\boldsymbol{R_0} \leftarrow 2\boldsymbol{R_0} + (\kappa)\boldsymbol{R_1}$ (where $\kappa = \pm 1$) and $\boldsymbol{R_{1-b}} \leftarrow 2\boldsymbol{R_{1-b}} + \boldsymbol{R_b}$. Except for the sign, this is essentially the same operation. Likewise, Algorithm 7 and Algorithm 3 scan scalar k in reverse directions and respectively repeat the operations $\boldsymbol{R_0} \leftarrow \boldsymbol{R_0} + (\kappa)\boldsymbol{R_1}$; $\boldsymbol{R_1} \leftarrow 2\boldsymbol{R_1}$ and $\boldsymbol{R_{1-b}} \leftarrow \boldsymbol{R_{1-b}} + \boldsymbol{R_b}$; $\boldsymbol{R_b} \leftarrow 2\boldsymbol{R_{1-b}}$. As a consequence, by taking into account the sign, we obtain analogously to the previous section two more co-Z scalar multiplication algorithms. They are depicted in Algs. 11 and 12.

Algorithm 11 Left-to-right signed-digit algorithm with co-Z addition formulæ

Input: $\boldsymbol{P} = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}_{\geqslant 3}$ with $k_0 = k_{n-1} = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $(\boldsymbol{R_0}, \boldsymbol{R_1}) \leftarrow \mathrm{TPLU}(\boldsymbol{P})$
2: **for** $i = n-2$ to 1 **do**
3: $\quad \kappa \leftarrow (-1)^{1+k_i}$
4: $\quad (\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_0}, (\kappa)\boldsymbol{R_1})$
5: $\quad (\boldsymbol{R_0}, \boldsymbol{R_1}) \leftarrow \mathrm{ZADDC}(\boldsymbol{R_1}, \boldsymbol{R_0})$; $\boldsymbol{R_1} \leftarrow (\kappa)\boldsymbol{R_1}$
6: **end for**
7: **return** $\mathrm{Jac2aff}(\boldsymbol{R_0})$

Algorithm 12 Right-to-left signed-digit algorithm with co-Z addition formulæ

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

1: $\kappa \leftarrow (-1)^{1+k_1}$; $(\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{DBLU}(\boldsymbol{P})$; $\boldsymbol{R_0} \leftarrow (\kappa)\boldsymbol{R_0}$
2: **for** $i = 2$ to $n-1$ **do**
3: $\quad \kappa \leftarrow (-1)^{1+k_i}$
4: $\quad (\boldsymbol{R_0}, \boldsymbol{R_1}) \leftarrow \mathrm{ZADDC}((\kappa)\boldsymbol{R_1}, \boldsymbol{R_0})$
5: $\quad (\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{ZADDU}(\boldsymbol{R_0}, \boldsymbol{R_1})$; $\boldsymbol{R_1} \leftarrow (\kappa)\boldsymbol{R_1}$
6: **end for**
7: $\boldsymbol{R_0} \leftarrow \mathrm{ZADD}(\boldsymbol{R_0}, \boldsymbol{R_1})$
8: **return** $\mathrm{Jac2aff}(\boldsymbol{R_0})$

4.4 Point doubling and tripling

Algorithms 9–12 require a point doubling or a point tripling operation for their initialization. We describe how this can be implemented.

Initial Point Doubling 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 $1\mathsf{M} + 5\mathsf{S}$:

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

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 [28]. 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 Point Tripling 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 [26]. 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 $\underline{6\mathsf{M} + 7\mathsf{S}}$.

Concerning the memory requirements, the two algorithms, namely DBLU and TPLU, can be implemented using at most 6 field registers (see Algs. 21 and 22, Appendix C).

5 Enhanced Algorithms

5.1 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 $10\mathsf{M}+4\mathsf{S}$ by two consecutive applications of the ZADDU function (Alg. 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 (Alg. 5) and the corresponding algorithm we got using co-Z arithmetic (Alg. 10), 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: $W_1' \leftarrow X_1C'$; $W_2' \leftarrow X_2C'$
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: $W_1 \leftarrow X_3'C$; $W_2 \leftarrow X_1C$
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 in Alg. 13. It clearly appears that the ZDAU operation only requires $\underline{9\mathsf{M}+7\mathsf{S}}$. Moreover, it can be implemented using 8 field registers; see Alg. 23 (Appendix C).

Algorithm 13 Co-Z 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 $\boldsymbol{Q} \leftarrow (\lambda^2X_2 : \lambda^3Y_2 : Z_3)$ with $Z_3 = \lambda Z$ for some $\lambda \neq 0$

1: **function** ZDAU($\boldsymbol{P},\boldsymbol{Q}$)
2: $C' \leftarrow (X_1 - X_2)^2$
3: $W_1' \leftarrow X_1C'$; $W_2' \leftarrow X_2C'$
4: $D' \leftarrow (Y_1 - Y_2)^2$; $A_1' \leftarrow Y_1(W_1' - W_2')$
5: $\hat{X}_3' \leftarrow D' - W_1' - W_2'$
6: $C \leftarrow (\hat{X}_3' - W_1')^2$
7: $Y_3' \leftarrow [(Y_1 - Y_2) + (W_1' - \hat{X}_3')]^2 - D' - C - 2A_1'$
8: $W_1 \leftarrow 4\hat{X}_3'C$; $W_2 \leftarrow 4W_1'C$
9: $D \leftarrow (Y_3' - 2A_1')^2$; $A_1 \leftarrow Y_3'(W_1 - W_2)$
10: $X_3 \leftarrow D - W_1 - W_2$; $Y_3 \leftarrow (Y_3' - 2A_1')(W_1 - X_3) - A_1$
11: $Z_3 \leftarrow Z\big((X_1 - X_2 + \hat{X}_3' - W_1')^2 - C' - C\big)$
12: $\overline{D} \leftarrow (Y_3' + 2A_1')^2$
13: $X_2 \leftarrow \overline{D} - W_1 - W_2$; $Y_2 \leftarrow (Y_3' + 2A_1')(W_1 - X_2) - A_1$
14: $Z_2 \leftarrow Z_3$
▷ $\boldsymbol{R} = (X_3 : Y_3; Z_3)$, $\boldsymbol{Q} = (X_2 : Y_2 : Z_2)$
15: **end function**

The combined ZDAU operation immediately gives rise to an alternative implementation of Joye's double-add algorithm (Alg. 5). Compared to our first implementation (Alg. 10), the cost per bit now amounts to $\underline{9\mathsf{M}+7\mathsf{S}}$ (instead of $11\mathsf{M}+5\mathsf{S}$). The resulting algorithm is presented in Alg. 14.

Algorithm 14 Joye's double-add algorithm with co-Z addition formulæ (II)

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

1: $b \leftarrow k_1$; $(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \text{TPLU}(\boldsymbol{P})$
2: **for** $i = 2$ to $n-1$ **do**
3: $b \leftarrow k_i$
4: $(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \text{ZDAU}(\boldsymbol{R_{1-b}}, \boldsymbol{R_b})$
5: **end for**
6: **return** Jac2aff($\boldsymbol{R_0}$)

The ZDAU operation also applies to the left-to-right signed-digit algorithm (Alg. 6) but a faster variant is presented hereafter (see § 5.2.2).

Similar savings can be obtained for our implementation of the Montgomery ladder (Alg. 9) and of the right-to-left signed-digit algorithm (Alg. 12). 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 $2\mathsf{S}$, a possible way is to keep track of the squared difference of the X-coordinates; see Appendix B.

5.2 (X, Y)-only operations

In [33], Venelli and Dassance astutely notice that the ZADDU and ZADDC operations do not involve the Z-coordinate of the input points for updating the X- and Y-coordinates. From this observation, they suggest to use the Montgomery ladder for the computation of $\boldsymbol{Q} = k\boldsymbol{P}$ with the X- and Y-coordinates *only*. The Z-coordinate of output point $\boldsymbol{Q}$ is recovered at the end of the computation. It was subsequently observed in [32] that the same trick applies to the zeroless signed-digit left-to-right algorithm.

In the sequel, the prime symbol ($'$) is used to denote operations that do not involve the Z-coordinate. For instance, ZADDU$'$ denotes the operation obtained by discarding the Z-coordinates in Alg. 1. This operation costs $\underline{4\mathsf{M} + 2\mathsf{S}}$ and requires 5 field registers. On the other hand, ZADDC$'$ operation costs $\underline{5\mathsf{M} + 3\mathsf{S}}$ and requires 6 field registers.[2]

5.2.1 Montgomery ladder

As aforementioned, the co-Z Montgomery ladder (see Alg. 9) can be rewritten so as to only process X- and Y-coordinates. Namely, registers $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ contains only the X- and Y-coordinates of points and operations ZADDC and ZADDU in Alg. 9 can be replaced with operations ZADDC$'$ and ZADDU$'$, respectively. But we can do better by defining operation ZACAU$'$ as the combination of operation ZADDC$'$ followed by operation ZADDU$'$. Using the same trick as in § 5.1, we can trade $1\mathsf{M}$ against $1\mathsf{S}$. This is achieved by adding the squared difference of the X-coordinates as an input to ZACAU$'$. A detailed implementation provided in Alg. 18 (Appendix B) yields a cost of $8\mathsf{M} + 6\mathsf{S}$ and requires 6 field registers; see Alg. 26 (Appendix C). As a result, the cost per bit of Algorithm 15 amounts to only $\underline{8\mathsf{M} + 6\mathsf{S}}$.

Then at the end of the loop, we need to recover the final Z-coordinate in order to get the affine coordinates of output point $\boldsymbol{Q} = k\boldsymbol{P}$. To this purpose, it can be checked that the last iteration (i.e., $i = 0$) of the Montgomery ladder, as depicted in Alg. 9, evaluates

$$(\boldsymbol{R_{k_0}}, \boldsymbol{R_{1-k_0}}) \leftarrow \mathrm{ZADDU}(\mathrm{ZADDC}(\boldsymbol{R_{k_0}}, \boldsymbol{R_{1-k_0}})) \ .$$

To avoid confusion, we use superscripts (in) and (out) to denote the input and output values — we also use superscript (tmp) to denote the intermediate values after the ZADDC operation. With this notation, the previous line is equivalently rewritten as $(\boldsymbol{R_{k_0}}^{(\mathrm{out})}, \boldsymbol{R_{1-k_0}}^{(\mathrm{out})}) = \mathrm{ZADDU}(\mathrm{ZADDC}(\boldsymbol{R_{k_0}}^{(\mathrm{in})}, \boldsymbol{R_{1-k_0}}^{(\mathrm{in})}))$, or in two steps as

$$(\boldsymbol{R_{k_0}}^{(\mathrm{out})}, \boldsymbol{R_{1-k_0}}^{(\mathrm{out})}) = \mathrm{ZADDU}(\boldsymbol{R_{1-k_0}}^{(\mathrm{tmp})}, \boldsymbol{R_{k_0}}^{(\mathrm{tmp})})$$

with

$$(\boldsymbol{R_{1-k_0}}^{(\mathrm{tmp})}, \boldsymbol{R_{k_0}}^{(\mathrm{tmp})}) := \mathrm{ZADDC}(\boldsymbol{R_{k_0}}^{(\mathrm{in})}, \boldsymbol{R_{1-k_0}}^{(\mathrm{in})}) \ .$$

Furthermore, as the Montgomery ladder keeps invariant the value of $\boldsymbol{R_1} - \boldsymbol{R_0} = \boldsymbol{P}$, we have $\boldsymbol{R_{k_0}}^{(\mathrm{tmp})} = \boldsymbol{R_{k_0}}^{(\mathrm{in})} - \boldsymbol{R_{1-k_0}}^{(\mathrm{in})} = (-1)^{1-k_0}\boldsymbol{P}$ and therefore

$$\mathrm{X}(\boldsymbol{P})\,\mathrm{Z}(\boldsymbol{P})\,\mathrm{Y}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})}) = (-1)^{1-k_0}\,\mathrm{X}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})\,\mathrm{Z}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})\,\mathrm{Y}(\boldsymbol{P}) \ .$$

Hence, letting $\mathrm{Z}(\boldsymbol{Q})$ denote the Z-coordinate of $\boldsymbol{Q} = \boldsymbol{R_0}^{(\mathrm{out})}$, it follows from the definition of ZADDU that

$$\begin{aligned}
\mathrm{Z}(\boldsymbol{Q}) &= \mathrm{Z}(\boldsymbol{R_{k_0}}^{(\mathrm{out})}) = \mathrm{Z}(\boldsymbol{R_{1-k_0}}^{(\mathrm{out})}) \\
&= \mathrm{Z}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})\big(\mathrm{X}(\boldsymbol{R_{1-k_0}}^{(\mathrm{tmp})}) - \mathrm{X}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})\big) \\
&= \mathrm{Z}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})(-1)^{1-k_0}\,\Delta_{\mathrm{X}}^{(\mathrm{tmp})} \\
&= \frac{\mathrm{X}(\boldsymbol{P})\,\mathrm{Z}(\boldsymbol{P})\,\mathrm{Y}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})}{\mathrm{X}(\boldsymbol{R_{k_0}}^{(\mathrm{tmp})})\,\mathrm{Y}(\boldsymbol{P})}\,\Delta_{\mathrm{X}}^{(\mathrm{tmp})} \ .
\end{aligned}$$

where $\Delta_{\mathrm{X}}^{(\mathrm{tmp})} := \mathrm{X}(\boldsymbol{R_0}^{(\mathrm{tmp})}) - \mathrm{X}(\boldsymbol{R_1}^{(\mathrm{tmp})})$. We therefore obtain an (X, Y)-only implementation of the Montgomery ladder; see Alg. 15. Note that using this formula, the affine coordinates of output point $\boldsymbol{Q}$ are recovered with a cost of $1\mathsf{I} + 8\mathsf{M} + 1\mathsf{S}$.

The complete algorithm is given below.

Algorithm 15 Montgomery ladder with (X, Y)-only co-Z addition formulæ

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

1: $(\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{DBLU}'(\boldsymbol{P})$
2: $C \leftarrow (\mathrm{X}(\boldsymbol{R_0}) - \mathrm{X}(\boldsymbol{R_1}))^2$
3: **for** $i = n - 2$ down to 1 **do**
4: $\quad b \leftarrow k_i$
5: $\quad (\boldsymbol{R_b}, \boldsymbol{R_{1-b}}, C) \leftarrow \mathrm{ZACAU}'(\boldsymbol{R_b}, \boldsymbol{R_{1-b}}, C)$
6: **end for**
7: $b \leftarrow k_0$; $(\boldsymbol{R_{1-b}}, \boldsymbol{R_b}) \leftarrow \mathrm{ZADDC}'(\boldsymbol{R_b}, \boldsymbol{R_{1-b}})$
8: $(x_P, y_P) \leftarrow \boldsymbol{P}$
9: $Z \leftarrow x_P\,\mathrm{Y}(\boldsymbol{R_b})(\mathrm{X}(\boldsymbol{R_0}) - \mathrm{X}(\boldsymbol{R_1}))$; $\lambda \leftarrow y_P\,\mathrm{X}(\boldsymbol{R_b})$
10: $(\boldsymbol{R_b}, \boldsymbol{R_{1-b}}) \leftarrow \mathrm{ZADDU}'(\boldsymbol{R_{1-b}}, \boldsymbol{R_b})$
11: **return** $\Big(\big(\frac{\lambda}{Z}\big)^2\,\mathrm{X}(\boldsymbol{R_0}), \big(\frac{\lambda}{Z}\big)^3\,\mathrm{Y}(\boldsymbol{R_0})\Big)$

[2] It clearly appears from Algs. 19 and 20 (in Appendix C) that discarding the Z-coordinate enables to save $1\mathsf{M}$ as well as 1 field register.

5.2.2 Signed-digit algorithm

(X, Y)-only co-Z operations can also be used with our left-to-right signed-digit algorithm (Alg. 11). More precisely, we can perform a ZADDU$'$ followed by a ZADDC$'$ to obtain (X, Y)-only double-add operation with co-Z update: ZDAU$'$. The total cost of this operation is hence of $9\mathsf{M} + 5\mathsf{S}$ but can be reduced to $\underline{8\mathsf{M} + 6\mathsf{S}}$ using a standard M/S trade-off. Moreover, ZDAU$'$ can be implemented using only 6 field registers; see Alg. 24 (Appendix C).

A further optimization of Alg. 11 is possible. When $\kappa = (-1)^{1+k_i}$ is equal to -1 (i.e., when $k_i = 0$), point in $\boldsymbol{R_1}$ is inverted prior to ZADDU and ZADDC operations and is then re-inverted thereafter. A better alternative is to switch the sign of $\boldsymbol{R_1}$ at the ith iteration if and only if $(-1)^{1+k_i} \neq (-1)^{1+k_{i+1}}$. Namely, we process $\boldsymbol{R_1} \leftarrow (-1)^b \boldsymbol{R_1}$ where $b = k_i \oplus k_{i+1}$.

At the end of the loop, $\boldsymbol{R_0}$ contains the X- and Y-coordinates of $k\boldsymbol{P}$ and $\boldsymbol{R_1}$ contains those of $(-1)^{1+k_1}\boldsymbol{P}$. Consequently, we can recover the complete coordinates of output point $\boldsymbol{Q} = k\boldsymbol{P}$ since $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ share the same Z-coordinate. After correcting the sign of $\boldsymbol{R_1}$ as $\boldsymbol{R_1} \leftarrow (-1)^{1+k_1}\boldsymbol{R_1}$, we get

$$\boldsymbol{P} = (x_P, y_P) \sim (X(\boldsymbol{R_1}) : Y(\boldsymbol{R_1}) : Z)$$

where $Z := Z(\boldsymbol{R_1}) = Z(\boldsymbol{R_0})$ is the final common Z-coordinate of $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$. From $x_P = \mathrm{X}(\boldsymbol{R_1})/Z^2$ and $y_P = \mathrm{Y}(\boldsymbol{R_1})/Z^3$, we immediately have

$$\frac{x_P}{y_P} = Z \cdot \frac{\mathrm{X}(\boldsymbol{R_1})}{\mathrm{Y}(\boldsymbol{R_1})}$$

and so the affine coordinates of $\boldsymbol{Q} = k\boldsymbol{P}$ are recovered as

$$k\boldsymbol{P} = \big(\lambda^2\,\mathrm{X}(\boldsymbol{R_0}), \lambda^3\,\mathrm{Y}(\boldsymbol{R_0})\big)$$

with

$$\lambda = Z^{-1} = \frac{y_P\,\mathrm{X}(\boldsymbol{R_1})}{x_P\,\mathrm{Y}(\boldsymbol{R_1})}\ .$$

The cost for this final step is of $1\mathsf{I} + 6\mathsf{M} + 1\mathsf{S}$. The complete algorithm is detailed in Alg. 16.

Algorithm 16 Left-to-right signed-digit algorithm with (X, Y)-only co-Z addition formulæ

Input: $\boldsymbol{P} = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}_{\geqslant 3}$ with $k_0 = k_{n-1} = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $(\boldsymbol{R_0}, \boldsymbol{R_1}) \leftarrow \mathrm{TPLU}'(\boldsymbol{P})$
2: **for** $i = n - 2$ down to 1 **do**
3: $\quad b \leftarrow k_i \oplus k_{i+1}$
4: $\quad \boldsymbol{R_1} \leftarrow (-1)^b \boldsymbol{R_1}$
5: $\quad (\boldsymbol{R_0}, \boldsymbol{R_1}) \leftarrow \mathrm{ZDAU}'(\boldsymbol{R_0}, \boldsymbol{R_1})$
6: **end for**
7: $\boldsymbol{R_1} \leftarrow (-1)^{1+k_1}\boldsymbol{R_1}$
8: $(x_P, y_P) \leftarrow \boldsymbol{P}$; $\lambda \leftarrow \frac{y_P\,\mathrm{X}(\boldsymbol{R_1})}{x_P\,\mathrm{Y}(\boldsymbol{R_1})}$
9: **return** $\big(\lambda^2\,\mathrm{X}(\boldsymbol{R_0}), \lambda^3\,\mathrm{Y}(\boldsymbol{R_0})\big)$

6 Discussion

6.1 Security considerations

When not properly implemented, scalar multiplication algorithms may be vulnerable to implementation attacks such as *side-channel analysis* (SCA). This kind of attacks exploits the physical information leakage produced by a device during a cryptographic computation. This includes the power consumption or the electromagnetic radiation [23, 15, 1]. Scalar multiplication implementations are vulnerable to two main types of side-channel attacks: *simple power analysis* (SPA) and *differential power analysis* (DPA). The latter uses correlations between the leakage and processed data and can usually be efficiently defeated by the use of randomization techniques [2, Chapter 29]. On the other hand, SPA-type attacks can recover the secret scalar from a single leakage trace (even in the presence of data randomization).

A classical protection against SPA-type attacks is to render the scalar multiplication algorithm regular, so that it repeats the same operation flow, regardless of the processed scalar. Different techniques are proposed in the litterature in order to obtain such regular algorithms. A first option is to make addition and doubling patterns indistinguishable. This can be achieved by using unified formulæ for point addition and point doubling [7] or by relying on *side-channel atomicity* whose principle is to build point addition and point doubling algorithms from the same atomic pattern of field operations [8]. Another option 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 initiated by Coron in [11] with the double-and-add-always algorithm (see § 3.1). Unfortunately, as it uses a dummy operation, it becomes subject to another class of attacks against implementations, the so-called safe-error attacks [34, 35], a special class of fault attacks [4, 6]. In contrast, the so-called *highly* regular algorithms, such as the Montgomery ladder or Joye's double-add, are naturally protected against both SPA-type attacks and safe-error attacks as every computed operation is effective. We remark that X-only versions of the Montgomery ladder ([7, 12, 19]) do not permit to check that

the output point belongs to the original curve and so may be subject to (classical) fault attacks, as was demonstrated in [13].

The scalar multiplication algorithms proposed in Section 4 are built from *highly* regular algorithms and maintain the same regular pattern of instructions without using dummy instructions. Algorithms 9 and 15 are based on Montgomery ladder whereas Algorithms 10 and 14 are based on Joye's double-add. Hence, our implementations inherit the same security features. It is also readily verified that our signed-digit algorithms (Algorithms 11, 12 and 16) always evaluates the same pattern of operations. Note that for the actual implementation of these algorithms to be regular, the conditional point inversion must be implemented in a regular fashion (see Appendix A for such implementations). Yet an additional advantage of all the proposed algorithms is that they made easy to assess the correctness of the computation by checking whether the output point belongs to the curve, which thwarts the fault attacks of [13].

6.2 Performance analysis

Table 1 summarizes the co-Z operation counts for the different addition formulæ introduced throughout the paper. The memory usage of most operations of Table 1 is detailed in Appendix C. Note that for certain (X, Y)-only co-Z algorithms, the memory count can be easily deduced from their co-Z counterpart. However, more complex (X, Y)-only operations like ZDAU$'$ and ZACAU$'$ need dedicated implementations (cf. Algs. 24 and 26) for a better memory usage.

Table 2 compares several regular implementations of scalar multiplication algorithms. The total cost is expressed for an n-bit scalar k. The total cost also includes the conversion to get the output point in affine coordinates. It turns out that the best performance is obtained with the co-Z Joye's double-add algorithm and the co-Z signed-digit algorithm for right-to-left algorithms and with the (X, Y)-only signed-digit algorithm for left-to-right algorithms. Remarkably, this latter algorithm as well as its unsigned counterpart outperforms in both speed and memory the X-only Montgomery ladder for general elliptic curves. Moreover, as explained in § 6.1, the presented co-Z implementations are protected against a variety of implementation attacks.

All in all, the two (X, Y)-only co-Z scalar multiplication algorithms can be considered as methods of choice for efficient and secure implementation of elliptic curve cryptography for general elliptic curves for memory-constrained devices.

References

1. Agrawal, D., Archambeault, B., Rao, J., Rohatgi, P.: The EM side-channel(s). In: B.S. Kaliski Jr., et al. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2002, *LNCS*, vol. 2523, pp. 29–45. Springer (2003)
2. Avanzi, R., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press (2005)
3. Bernstein, D.J., Lange, T.: Explicit-formulas database. `http://hyperelliptic.org/EFD/g1p/auto-shortw.html`
4. Biehl, I., Meyer, B., Müller, V.: Differential fault attacks on elliptic curve cryptosystems. In: M. Bellare (ed.) Advances in Cryptology – CRYPTO 2000, *LNCS*, vol. 1880, pp. 131–146. Springer (2000)
5. Blake, I.F., Seroussi, G., Smart, N.P. (eds.): Advances in Elliptic Curve Cryptography, *London Mathematical Society Lecture Note Series*, vol. 317. Cambridge University Press (2005)
6. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. Journal of Cryptology **14**(2), 110–119 (2001). Extended abstract in Proc. of EUROCRYPT '97
7. Brier, E., Joye, M.: Weierstraß elliptic curves and side-channel attacks. In: D. Naccache, P. Paillier (eds.) Public Key Cryptography (PKC 2002), *LNCS*, vol. 2274, pp. 335–345. Springer (2002)
8. Chevallier-Mames, B., Ciet, M., Joye, M.: Low-cost solutions for preventing simple side-channel analysis: Side-channel atomicity. IEEE Transactions on Computers **53**(6), 760–768 (2004)
9. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of numbers generated by addition in formal groups and new primality and factorization tests. Advances in Applied Mathematics **7**(4), 385–434 (1986)
10. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: K. Ohta, D. Pei (eds.) Advances in Cryptology – ASIACRYPT '98, *LNCS*, vol. 1514, pp. 51–65. Springer (1998)
11. Coron, J.S.: Resistance against differential power analysis for elliptic curve cryptosystems. In: Ç.K. Koç, C. Paar (eds.) Cryptographic Hardware and Embedded Systems (CHES '99), *LNCS*, vol. 1717, pp. 292–302. Springer (1999)
12. Fischer, W., Giraud, C., Knudsen, E.W., Seifert, J.P.: 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/`
13. Fouque, P.A., Lercier, R., Réal, D., Valette, F.: Fault attack on elliptic curve Montgomery ladder implementation. In: L. Breveglieri, et al. (eds.) Fault Diagnosis and Tolerance in Cryptography (FDTC 2008), pp. 92–98. IEEE Computer Society (2008)
14. Galbraith, S., Lin, X., Scott, M.: A faster way to do ECC. Presented at 12th Workshop on Elliptic Curve Cryptography (ECC 2008), Utrecht, The Netherlands (2008). Slides available at URL `http://www.hyperelliptic.org/tanja/conf/ECC08/slides/Mike-Scott.pdf`
15. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic analysis: Concrete results. In: Ç.K. Koç, D. Naccache, C. Paar (eds.) Cryptographic Hardware and Embedded Systems – CHES 2001, *LNCS*, vol. 2162, pp. 251–261. Springer (2001)

Table 1 Best operation counts and memory usage for various co-Z addition formulæ.

Operation	Notation	# regs.	Cost
Point addition:			
– Co-Z addition with update (Alg. 19)	ZADDU	6	$5\mathsf{M}+2\mathsf{S}$
– (X,Y)-only co-Z addition with update[a]	ZADDU′	5	$4\mathsf{M}+2\mathsf{S}$
– Conjugate co-Z addition (Alg. 20)	ZADDC	7	$6\mathsf{M}+3\mathsf{S}$
– (X,Y)-only conjugate co-Z addition[b]	ZADDC′	6	$5\mathsf{M}+3\mathsf{S}$
Point doubling-addition:			
– Co-Z doubling-addition with update (Alg. 23)[c]	ZDAU	8	$9\mathsf{M}+7\mathsf{S}$
– (X,Y)-only co-Z doubling-addition with update (Alg. 24)	ZDAU′	6	$8\mathsf{M}+6\mathsf{S}$
– Co-Z conjugate-addition–addition with update (Alg. 25)[d]	ZACAU	8	$9\mathsf{M}+7\mathsf{S}$
– (X,Y)-only co-Z conjugate-addition–addition with update (Alg. 26)	ZACAU′	6	$8\mathsf{M}+6\mathsf{S}$
Point doubling and tripling:			
– Co-Z doubling (Alg. 21)	DBLU	6	$1\mathsf{M}+5\mathsf{S}$
– (X,Y)-only co-Z doubling[e]	DBLU′	5	$1\mathsf{M}+5\mathsf{S}$
– Co-Z tripling (Alg. 22)	TPLU	6	$6\mathsf{M}+7\mathsf{S}$
– (X,Y)-only co-Z tripling[f]	TPLU′	5	$5\mathsf{M}+7\mathsf{S}$

[a] Obtained from Alg. 19.
[b] Obtained from Alg. 20.
[c] Similarly to ZACAU, it is also possible to derive an implementation requiring $10\mathsf{M}+6\mathsf{S}$ with only 7 field registers.
[d] The implementation offered by Alg. 25 actually costs $10\mathsf{M}+6\mathsf{S}$ with only 7 field registers. But the same M/S trade-off as for ZDAU applies, leading to an implementation costing $9\mathsf{M}+7\mathsf{S}$ at the expense of one more register. See Appendix B.
[e] Obtained from Alg. 21.
[f] Obtained from Alg. 22.

Table 2 Comparison of regular scalar multiplication algorithms.

Algorithm	Main op.	# regs.	Total cost
Right-to-left algorithms:			
– Basic Joye's double-add (Alg. 5)	DA[a]	10	$n(13\mathsf{M}+8\mathsf{S})+1\mathsf{I}+3\mathsf{M}+1\mathsf{S}$
– Co-Z Joye's double-add (Alg. 14)[b]	ZDAU	8	$n(9\mathsf{M}+7\mathsf{S})+1\mathsf{I}-9\mathsf{M}-6\mathsf{S}$
– Co-Z signed-digit algorithm (Alg. 17)[c]	ZACAU	8	$n(9\mathsf{M}+7\mathsf{S})+1\mathsf{I}-9\mathsf{M}-6\mathsf{S}$
Left-to-right algorithms:			
– Basic Montgomery ladder (Alg. 3)	DBL and ADD	8	$n(12\mathsf{M}+13\mathsf{S})+1\mathsf{I}+3\mathsf{M}+1\mathsf{S}$
– X-only Montgomery ladder [7,12,19]	MontADD[d]	7	$n(9\mathsf{M}+7\mathsf{S})+1\mathsf{I}+14\mathsf{M}+3\mathsf{S}$
– (X,Y)-only co-Z Montgomery ladder (Alg. 15)	ZACAU′	6	$n(8\mathsf{M}+6\mathsf{S})+1\mathsf{I}+1\mathsf{M}$
– (X,Y)-only co-Z signed-digit algorithm (Alg. 16)	ZDAU′	6	$n(8\mathsf{M}+6\mathsf{S})+1\mathsf{I}-5\mathsf{M}-4\mathsf{S}$

[a] With DA the general doubling-addition formula from [24].
[b] It is also possible to get an implementation with 7 field registers at the cost of $n(10\mathsf{M}+6\mathsf{S})+1\mathsf{I}-9\mathsf{M}-6\mathsf{S}$. See Appendix B.
[c] Idem.
[d] See [16, Appendix B] for a detailed implementation of MontADD. The cost assumes that multiplications by curve parameter a are negligible; e.g., $a=-3$.

16. Goundar, R.R., Joye, M., Miyaji, A.: Co-Z addition formulæ and binary ladders on elliptic curves. In: S. Mangard, F.X. Standaert (eds.) Cryptographic Hardware and Embedded Systems – CHES 2010, *LNCS*, vol. 6225, pp. 65–79. Springer (2010)
17. IEEE Std 1363-2000: IEEE Standard Specifications for Public-Key Cryptography. IEEE Computer Society (2000)
18. Izu, T., Möller, B., Takagi, T.: Improved elliptic curve multiplication methods reistant against side-channel attacks. In: A. Menezes, P. Sarkar (eds.) Progress in Cryptology – INDOCRYPT 2002, *LNCS*, vol. 2551, pp. 296–313. Springer (2002)
19. Izu, T., Takagi, T.: A fast parallel elliptic curve multiplication resistant against side channel attacks. In: D. Naccache, P. Paillier (eds.) Public Key Cryptography (PKC 2002), *LNCS*, vol. 2274, pp. 280–296. Springer (2002)
20. Joye, M.: Highly regular right-to-left algorithms for scalar multiplication. In: P. Paillier, I. Verbauwhede (eds.) Cryptographic Hardware and Embedded Systems – CHES 2007, *LNCS*, vol. 4727, pp. 135–147. Springer (2007)
21. Joye, M., Yen, S.M.: The Montgomery powering ladder. In: B.S. Kaliski Jr., et al. (eds.) Cryptographic Hardware and Embedded Systems – CHES 2002, *LNCS*, vol. 2523, pp. 291–302. Springer (2003)
22. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation **48**(177), 203–209 (1987)

23. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: M. Wiener (ed.) Advances in Cryptology – CRYPTO '99, *LNCS*, vol. 1666, pp. 388–397. Springer (1999)
24. Longa, P.: ECC Point Arithmetic Formulae (EPAF). `http://patricklonga.bravehost.com/jacobian.html`
25. Longa, P., Gebotys, C.H.: Novel precomputation schemes for elliptic curve cryptosystems. In: M. Abdalla, et al. (eds.) Applied Cryptography and Network Security (ACNS 2009), *LNCS*, vol. 5536, pp. 71–88. Springer (2009)
26. Longa, P., Miri, A.: New composite operations and precomputation for elliptic curve cryptosystems over prime fields. In: R. Cramer (ed.) Public Key Cryptography – PKC 2008, *LNCS*, vol. 4939, pp. 229–247. Springer (2008)
27. López, J., Dahab, R.: Fast multiplication on elliptic curves over $GF(2^m)$ without precomputation. In: Ç.K. Koç, C. Paar (eds.) Cryptographic Hardware and Embedded Systems (CHES '99), *LNCS*, vol. 1717, pp. 316–327. Springer (1999)
28. Meloni, N.: New point addition formulæ for ECC applications. In: C. Carlet, B. Sunar (eds.) Arithmetic of Finite Fields (WAIFI 2007), *LNCS*, vol. 4547, pp. 189–201. Springer (2007)
29. Miller, V.S.: Use of elliptic curves in cryptography. In: H.C. Williams (ed.) Advances in Cryptology – CRYPTO '85, *LNCS*, vol. 218, pp. 417–426. Springer (1985)
30. Montgomery, P.L.: Speeding up the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (1987)
31. Morain, F., Olivos, J.: Speeding up the computations on an elliptic curve using addition-subtraction chains. RAIRO Informatique théorique et applications **24**(6), 531–543 (1990)
32. Rivain, M.: Fast and regular algorithms for scalar multiplication over elliptic curves. Cryptology ePrint Archive, Report 2011/338 (2011). `http://eprint.iacr.org/`
33. Venelli, A., Dassance, F.: Faster side-channel resistant elliptic curve scalar multiplication. Contemporary Mathematics **521**, 29–40 (2010)
34. Yen, S.M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Transactions on Computers **49**(9), 967–970 (2000)
35. Yen, S.M., Kim, S., Lim, S., Moon, S.J.: A countermeasure against one physical cryptanalysis may benefit another attack. In: K. Kim (ed.) Information Security and Cryptology – ICISC 2001, *LNCS*, vol. 2288, pp. 414–427. Springer (2002)

A 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 $\boldsymbol{P} = (X : Y : Z)$ and $b \in \{0, 1\}$. A first solution is to process the following steps:

1: $T_0 \leftarrow Y$
2: $T_1 \leftarrow -Y$
3: $Y \leftarrow T_b$

This solution is very simple and efficient: it only costs one field negation for computing $-Y$ (other steps being processed by pointer arithmetic of negligible cost). However, when $b = 0$, the negation of Y is a dummy operation which renders the implementation subject to safe-error attacks. Indeed, by injecting a fault in field register T_1 and checking the correctness, one could see whether T_1 were 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 leads to the following solution:

1: $r \xleftarrow{\$} \{0, 1\}$
2: $T_r \leftarrow Y$
3: $T_{r \oplus 1} \leftarrow -Y$
4: $Y \leftarrow T_{r \oplus b}$

An alternative solution, with no dummy operations, runs as follows:

1: $T_0 \leftarrow Y$
2: $T_1 \leftarrow -Y$
3: $Y \leftarrow 2T_b + T_{b \oplus 1}$

This solution nevertheless implies further field operations.

B ZACAU and ZACAU′ Operations

ZACAU is defined as the successive application of ZADDC and ZADDU. Arithmetically, it takes a pair of co-Z points $(\boldsymbol{P}, \boldsymbol{Q})$ and computes the co-Z pair $(2\boldsymbol{P}, \boldsymbol{P} + \boldsymbol{Q})$. This operation serves as the building block for the co-Z Montgomery ladder (Alg. 9) as well as of the co-Z right-to-left signed-digit algorithm (Alg. 12). For completeness, we present the latter algorithm hereafter. It immediately follows from Algorithm 12 using the trick of § 5.2.2.

Algorithm 17 Right-to-left signed-digit algorithm with co-Z addition formulæ (II)

Input: $\boldsymbol{P} = (x_P, y_P) \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}_{\geqslant 3}$ with $k_0 = k_{n-1} = 1$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

1: $\kappa \leftarrow (-1)^{1+k_1}$; $\boldsymbol{R_0} \leftarrow (\kappa)\boldsymbol{P}$ $(\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{DBLU}(\boldsymbol{R_0})$
2: **for** $i = 2$ down to $n - 1$ **do**
3: $\quad b \leftarrow k_i \oplus k_{i-1}$
4: $\quad \boldsymbol{R_1} \leftarrow (-1)^b \boldsymbol{R_1}$
5: $\quad (\boldsymbol{R_1}, \boldsymbol{R_0}) \leftarrow \mathrm{ZACAU}(\boldsymbol{R_1}, \boldsymbol{R_0})$
6: **end for**
7: $\boldsymbol{R_0} \leftarrow \mathrm{ZADD}(\boldsymbol{R_0}, \boldsymbol{R_1})$
8: **return** $\mathrm{Jac2aff}(\boldsymbol{R_0})$

In its basic form, ZACAU requires $10\mathsf{M} + 6\mathsf{S}$ using 7 field registers. The corresponding implementation is given in Alg. 25. With one more field register, the cost can be reduced to $\underline{9\mathsf{M} + 7\mathsf{S}}$ using a M/S trade-off similar to the one used for ZDAU (see § 5.1).

We address below in more detail the (X, Y)-only version of ZACAU (i.e., ZACAU′), which is faster. For a point $\boldsymbol{P} = (X_1 : Y_1 : Z)$ given in Jacobian coordinates, we let $\boldsymbol{P'}$ denote the same point without the Z-coordinate; i.e., $\boldsymbol{P'} = (X_1 : Y_1)$. The ZACAU′ operation takes on input the X- and Y-coordinates of two points having the same Z-coordinate, $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$, and outputs the X- and Y-coordinates of two points having the same Z-coordinate, $\boldsymbol{R} = (X_3 : Y_3 : Z^*)$ and $\boldsymbol{S} = (X_4 : Y_4 : Z^*)$, such that

$$(\boldsymbol{R'}, \boldsymbol{S'}) = ((X_3 : Y_3), (X_4 : Y_4)) \\ := \mathrm{ZADDU'}(\mathrm{ZADDC'}(\boldsymbol{P'}, \boldsymbol{Q'})) \quad \text{with } \mathrm{Z}(\boldsymbol{R}) = \mathrm{Z}(\boldsymbol{S})$$

where $\boldsymbol{P}' = (X_1 : Y_1)$ and $\boldsymbol{Q}' = (X_2 : Y_2)$.

Moreover, in order to apply the S/M trade-off, we add a variable C that keeps track of the value of $(X_1 - X_2)^2$. This variable is updated and returned as an output of function ZACAU′. When used in the Montgomery ladder, note that the value is independent of the next bit: if $(X_3 : Y_3)$, $(X_4 : Y_4)$ denote the output points, since $(X_3 - X_4)^2 = (X_4 - X_3)^2$, we can in all cases return $C = (X_3 - X_4)^2$.

A detailed implementation of operation ZACAU′ is presented in Alg. 18. Note that some rescaling was applied.

Algorithm 18 (X, Y)-only co-Z conjugate-addition–addition with update (ZACAU′)

Require: $\boldsymbol{P}' = (X_1 : Y_1)$ and $\boldsymbol{Q}' = (X_2 : Y_2)$ for some $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$, and $C = (X_1 - X_2)^2$
Ensure: $(\boldsymbol{R}', \boldsymbol{S}', C) \leftarrow$ ZACAU′$(\boldsymbol{P}', \boldsymbol{Q}', C)$ where $\boldsymbol{R}' \leftarrow (X_3 : Y_3)$ and $\boldsymbol{S}' \leftarrow (X_4 : Y_4)$ for some $\boldsymbol{R} = 2\boldsymbol{P} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{S} = \boldsymbol{P} + \boldsymbol{Q} = (X_4 : Y_4 : Z_4)$ such that $Z_3 = Z_4$, and $C \leftarrow (X_3 - X_4)^2$

function ZACAU′$(\boldsymbol{P}', \boldsymbol{Q}', C)$
  $W_1 \leftarrow X_1 C$; $W_2 \leftarrow X_2 C$
  $D \leftarrow (Y_1 - Y_2)^2$; $A_1 \leftarrow Y_1(W_1 - W_2)$
  $X_1' \leftarrow D - W_1 - W_2$; $Y_1' \leftarrow (Y_1 - Y_2)(W_1 - X_1') - A_1$
  $\overline{D} \leftarrow (Y_1 + Y_2)^2$
  $X_2' \leftarrow \overline{D} - W_1 - W_2$; $Y_2' \leftarrow (Y_1 + Y_2)(W_1 - X_2') - A_1$
  $C' \leftarrow (X_1' - X_2')^2$
  $X_4 \leftarrow X_1' C'$; $W_2' \leftarrow X_2' C'$
  $D' \leftarrow (Y_1' - Y_2')^2$; $Y_4 \leftarrow Y_1'(X_4 - W_2')$
  $X_3 \leftarrow D' - X_4 - W_2'$
  $C \leftarrow (X_3 - X_4)^2$;
  $Y_3 \leftarrow (Y_1' - Y_2' + X_4 - X_3)^2 - D' - C - 2Y_4$
  $X_3 \leftarrow 4X_3$; $Y_3 \leftarrow 4Y_3$; $X_4 \leftarrow 4X_4$
  $Y_4 \leftarrow 8Y_4$; $C \leftarrow 16C$
  ▷ $\boldsymbol{R}' = (X_3 : Y_3)$, $\boldsymbol{S}' = (X_4 : Y_4)$, C
end function

C Memory Usage

We use the convention of [17]. The different field registers are considered as temporary variables and are denoted by T_i, $1 \leqslant i \leqslant 8$. Operations in place are permitted, which simply means for that a temporary variable can be composed (i.e., multiplied, added or subtracted) with another one and the result written back in the first temporary variable. When dealing with variables T_i, symbols $+$, $-$, $\times$, and $(\cdot)^2$ respectively stand for addition, subtraction, multiplication and squaring in the underlying field.

Algorithm 19 Co-Z addition with update (register allocation)

Require: $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$
Ensure: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow$ 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 $\lambda \neq 0$

1: **function** ZADDU$(\boldsymbol{P}, \boldsymbol{Q})$
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = Z$, $T_4 = X_2$, $T_5 = Y_2$
2:

1. $T_6 \leftarrow T_1 - T_4$	$\{X_1 - X_2\}$	8. $T_4 \leftarrow T_4 - T_1$	$\{D - W_1\}$
2. $T_3 \leftarrow T_3 \times T_6$	$\{Z_3\}$	9. $T_4 \leftarrow T_4 - T_6$	$\{X_3\}$
3. $T_6 \leftarrow {T_6}^2$	$\{C\}$	10. $T_6 \leftarrow T_1 - T_6$	$\{W_1 - W_2\}$
4. $T_1 \leftarrow T_1 \times T_6$	$\{W_1\}$	11. $T_2 \leftarrow T_2 \times T_6$	$\{A_1\}$
5. $T_6 \leftarrow T_6 \times T_4$	$\{W_2\}$	12. $T_6 \leftarrow T_1 - T_4$	$\{W_1 - X_3\}$
6. $T_5 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	13. $T_5 \leftarrow T_5 \times T_6$	$\{Y_3 + A_1\}$
7. $T_4 \leftarrow {T_5}^2$	$\{D\}$	14. $T_5 \leftarrow T_5 - T_2$	$\{Y_3\}$

$\boldsymbol{R} = (T_4 : T_5 : T_3)$, $\boldsymbol{P} = (T_1 : T_2 : T_3)$
3: **end function**

Algorithm 20 Conjugate co-Z addition (register allocation)

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

1: **function** ZADDC$(\boldsymbol{P}, \boldsymbol{Q})$
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = Z$, $T_4 = X_2$, $T_5 = Y_2$
2:

1. $T_6 \leftarrow T_1 - T_4$	$\{X_1 - X_2\}$	13. $T_1 \leftarrow T_1 - T_6$	$\{X_3\}$
2. $T_3 \leftarrow T_3 \times T_6$	$\{Z_3\}$	14. $T_6 \leftarrow T_6 - T_7$	$\{W_2 - W_1\}$
3. $T_6 \leftarrow {T_6}^2$	$\{C\}$	15. $T_6 \leftarrow T_6 \times T_2$	$\{-A_1\}$
4. $T_7 \leftarrow T_1 \times T_6$	$\{W_1\}$	16. $T_2 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$
5. $T_6 \leftarrow T_6 \times T_4$	$\{W_2\}$	17. $T_5 \leftarrow 2T_5$	$\{2Y_2\}$
6. $T_1 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$	18. $T_5 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$
7. $T_4 \leftarrow {T_1}^2$	$\{\overline{D}\}$	19. $T_7 \leftarrow T_7 - T_4$	$\{W_1 - \overline{X_3}\}$
8. $T_4 \leftarrow T_4 - T_7$	$\{\overline{D} - W_1\}$	20. $T_5 \leftarrow T_5 \times T_7$	$\{\overline{Y_3} + A_1\}$
9. $T_4 \leftarrow T_4 - T_6$	$\{\overline{X_3}\}$	21. $T_5 \leftarrow T_5 + T_6$	$\{\overline{Y_3}\}$
10. $T_1 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	22. $T_7 \leftarrow T_4 + T_7$	$\{W_1\}$
11. $T_1 \leftarrow {T_1}^2$	$\{D\}$	23. $T_7 \leftarrow T_7 - T_1$	$\{W_1 - X_3\}$
12. $T_1 \leftarrow T_1 - T_7$	$\{D - W_1\}$	24. $T_2 \leftarrow T_2 \times T_7$	$\{Y_3 + A_1\}$
		25. $T_2 \leftarrow T_2 + T_6$	$\{Y_3\}$

$\boldsymbol{R} = (T_1 : T_2 : T_3)$, $\boldsymbol{S} = (T_4 : T_5 : T_3)$
3: **end function**

Algorithm 21 Co-Z doubling with update (register allocation)

Require: $\boldsymbol{P} = (X_1 : Y_1 : 1)$
Ensure: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow$ DBLU$(\boldsymbol{P})$ where $\boldsymbol{R} \leftarrow 2\boldsymbol{P} = (X_2 : Y_2 : Z_2)$ and $\boldsymbol{P} \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : \lambda)$ with $\lambda = Z_2$

1: **function** DBLU$(\boldsymbol{P})$
$T_0 = a$, $T_1 = X_1$, $T_2 = Y_1$
2:

1. $T_3 \leftarrow 2T_2$	$\{Z_2\}$	10. $T_0 \leftarrow T_0 + T_5$	$\{a + B\}$
2. $T_2 \leftarrow {T_2}^2$	$\{E\}$	11. $T_5 \leftarrow 2T_5$	$\{2B\}$
3. $T_4 \leftarrow T_1 + T_2$	$\{X_1 + E\}$	12. $T_0 \leftarrow T_0 + T_5$	$\{M\}$
4. $T_4 \leftarrow {T_4}^2$	$\{(X_1 + E)^2\}$	13. $T_4 \leftarrow {T_0}^2$	$\{M^2\}$
5. $T_5 \leftarrow {T_1}^2$	$\{B\}$	14. $T_5 \leftarrow 2T_1$	$\{2S\}$
6. $T_4 \leftarrow T_4 - T_5$	$\{(X_1 + E)^2 - B\}$	15. $T_4 \leftarrow T_4 - T_5$	$\{X_2\}$
7. $T_2 \leftarrow {T_2}^2$	$\{L\}$	16. $T_2 \leftarrow 8T_2$	$\{8L\}$
8. $T_4 \leftarrow T_4 - T_2$	$\{(X_1 + E)^2 - B - L\}$	17. $T_5 \leftarrow T_1 - T_4$	$\{S - X_2\}$
9. $T_1 \leftarrow 2T_4$	$\{S\}$	18. $T_5 \leftarrow T_5 \times T_0$	$\{M(S - X_2)\}$
		19. $T_5 \leftarrow T_5 - T_2$	$\{Y_2\}$

$\boldsymbol{R} = (T_4 : T_5 : T_3)$, $\boldsymbol{P} = (T_1 : T_2 : T_3)$
3: **end function**

Algorithm 22 Co-Z tripling with update (register allocation)

Require: $\boldsymbol{P} = (X_1 : Y_1 : 1)$
Ensure: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow \text{TPLU}(\boldsymbol{P})$ where $\boldsymbol{R} \leftarrow 3\boldsymbol{P} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{P} \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : \lambda)$ with $\lambda = Z_3$

1: **function** TPLU($\boldsymbol{P}$)
2: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow \text{DBLU}(\boldsymbol{P})$
3: $(\boldsymbol{R}, \boldsymbol{P}) \leftarrow \text{ZADDU}(\boldsymbol{P}, \boldsymbol{R})$
4: **end function**

Algorithm 23 Co-Z doubling-addition with update (register allocation)

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 $\lambda \neq 0$

1: **function** ZDAU($\boldsymbol{P}, \boldsymbol{Q}$)
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = Z$, $T_4 = X_2$, $T_5 = Y_2$

1. $T_6 \leftarrow T_1 - T_4$	$\{X_1 - X_2\}$	
2. $T_7 \leftarrow {T_6}^2$	$\{C'\}$	
3. $T_1 \leftarrow T_1 \times T_7$	$\{W_1'\}$	
4. $T_4 \leftarrow T_4 \times T_7$	$\{W_2'\}$	
5. $T_5 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	
6. $T_8 \leftarrow T_1 - T_4$	$\{W_1' - W_2'\}$	
7. $T_2 \leftarrow T_2 \times T_8$	$\{A_1'\}$	
8. $T_2 \leftarrow 2T_2$	$\{2A_1'\}$	
9. $T_8 \leftarrow {T_5}^2$	$\{D'\}$	
10. $T_4 \leftarrow T_8 - T_4$	$\{D' - W_2'\}$	
2: 11. $T_4 \leftarrow T_4 - T_1$	$\{\hat{X}_3'\}$	
12. $T_4 \leftarrow T_4 - T_1$	$\{\hat{X}_3' - W_1'\}$	
13. $T_6 \leftarrow T_4 + T_6$	$\{X_1 - X_2 + \hat{X}_3' - W_1'\}$	
14. $T_6 \leftarrow {T_6}^2$	$\{(X_1 - X_2 + \hat{X}_3' - W_1')^2\}$	
15. $T_6 \leftarrow T_6 - T_7$	$\{(X_1 - X_2 + \hat{X}_3' - W_1')^2 - C'\}$	
16. $T_5 \leftarrow T_5 - T_4$	$\{Y_1 - Y_2 + W_1' - \hat{X}_3'\}$	
17. $T_5 \leftarrow {T_5}^2$	$\{(Y_1 - Y_2 + W_1' - \hat{X}_3')^2\}$	
18. $T_5 \leftarrow T_5 - T_8$	$\{Y_3' + C + 2A_1'\}$	
19. $T_5 \leftarrow T_5 - T_2$	$\{Y_3' + C\}$	
20. $T_7 \leftarrow {T_4}^2$	$\{C\}$	
21. $T_5 \leftarrow T_5 - T_7$	$\{Y_3'\}$	
22. $T_8 \leftarrow 4T_7$	$\{4C\}$	
23. $T_6 \leftarrow T_6 - T_7$	$\{(X_1 - X_2 + \hat{X}_3' - W_1')^2 - C' - C\}$	
24. $T_3 \leftarrow T_3 \times T_6$	$\{Z_3\}$	
25. $T_6 \leftarrow T_1 \times T_8$	$\{W_2\}$	
26. $T_1 \leftarrow T_1 + T_4$	$\{\hat{X}_3'\}$	
27. $T_8 \leftarrow T_8 \times T_1$	$\{W_1\}$	
28. $T_7 \leftarrow T_2 + T_5$	$\{Y_3' + 2A_1'\}$	
29. $T_2 \leftarrow T_5 - T_2$	$\{Y_3' - 2A_1'\}$	
30. $T_1 \leftarrow T_8 - T_6$	$\{W_1 - W_2\}$	
31. $T_5 \leftarrow T_5 \times T_1$	$\{A_1\}$	
32. $T_6 \leftarrow T_6 + T_8$	$\{W_1 + W_2\}$	
33. $T_1 \leftarrow {T_2}^2$	$\{D\}$	
34. $T_1 \leftarrow T_1 - T_6$	$\{X_3\}$	
35. $T_4 \leftarrow T_8 - T_1$	$\{W_1 - X_3\}$	
36. $T_2 \leftarrow T_2 \times T_4$	$\{Y_3 + A_1\}$	
37. $T_2 \leftarrow T_2 - T_5$	$\{Y_3\}$	
38. $T_4 \leftarrow {T_7}^2$	$\{\overline{D}\}$	
39. $T_4 \leftarrow T_4 - T_6$	$\{X_2\}$	
40. $T_8 \leftarrow T_8 - T_4$	$\{W_1 - X_2\}$	
41. $T_7 \leftarrow T_7 \times T_8$	$\{Y_2 + A_1\}$	
42. $T_5 \leftarrow T_7 - T_5$	$\{Y_2\}$	

$\boldsymbol{R} = (T_1 : T_2 : T_3)$, $\boldsymbol{Q} = (T_4 : T_5 : T_3)$
3: **end function**

Algorithm 24 (X, Y)-only co-Z doubling-addition with update (register allocation)

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

1: **function** $\mathrm{ZDAU}'(\boldsymbol{P}', \boldsymbol{Q}')$
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = X_2$, $T_4 = Y_2$

1. $T_5 \leftarrow T_1 - T_3$	$\{X_1 - X_2\}$	
2. $T_5 \leftarrow {T_5}^2$	$\{C'\}$	
3. $T_1 \leftarrow T_1 \times T_5$	$\{W_1'\}$	
4. $T_3 \leftarrow T_3 \times T_5$	$\{W_2'\}$	
5. $T_4 \leftarrow T_2 - T_4$	$\{Y_1 - Y_2\}$	
6. $T_5 \leftarrow T_1 - T_3$	$\{W_1' - W_2'\}$	
7. $T_2 \leftarrow T_2 \times T_5$	$\{A_1'\}$	
8. $T_2 \leftarrow 2T_2$	$\{2A_1'\}$	
9. $T_5 \leftarrow {T_4}^2$	$\{D'\}$	
10. $T_3 \leftarrow T_5 - T_3$	$\{D' - W_2'\}$	
11. $T_3 \leftarrow T_3 - T_1$	$\{\hat{X}_3'\}$	
12. $T_6 \leftarrow T_1 - T_3$	$\{W_1' - \hat{X}_3'\}$	
13. $T_4 \leftarrow T_4 + T_6$	$\{Y_1 - Y_2 + W_1' - \hat{X}_3'\}$	
14. $T_4 \leftarrow {T_4}^2$	$\{(Y_1 - Y_2 + W_1' - \hat{X}_3')^2\}$	
15. $T_4 \leftarrow T_4 - T_5$	$\{Y_3' + C + 2A_1'\}$	
16. $T_4 \leftarrow T_4 - T_2$	$\{Y_3' + C\}$	
17. $T_5 \leftarrow {T_6}^2$	$\{C\}$	
18. $T_4 \leftarrow T_4 - T_5$	$\{Y_3'\}$	
19. $T_5 \leftarrow 4T_5$	$\{4C\}$	
20. $T_6 \leftarrow T_3 \times T_5$	$\{W_1\}$	
21. $T_5 \leftarrow T_5 \times T_1$	$\{W_2\}$	
22. $T_3 \leftarrow T_4 - T_2$	$\{Y_3' - 2A_1'\}$	
23. $T_1 \leftarrow {T_3}^2$	$\{D\}$	
24. $T_1 \leftarrow T_1 - T_6$	$\{D - W_1\}$	
25. $T_1 \leftarrow T_1 - T_5$	$\{X_3\}$	
26. $T_3 \leftarrow T_2 + T_4$	$\{Y_3' + 2A_1'\}$	
27. $T_3 \leftarrow {T_3}^2$	$\{\overline{D}\}$	
28. $T_3 \leftarrow T_3 - T_6$	$\{\overline{D} - W_1\}$	
29. $T_3 \leftarrow T_3 - T_5$	$\{X_2\}$	
30. $T_5 \leftarrow T_6 - T_5$	$\{W_1 - W_2\}$	
31. $T_5 \leftarrow T_5 \times T_4$	$\{A_1\}$	
32. $T_4 \leftarrow T_2 + T_4$	$\{Y_3' + 2A_1'\}$	
33. $T_2 \leftarrow 2T_2$	$\{4A_1'\}$	
34. $T_2 \leftarrow T_4 - T_2$	$\{Y_3' - 2A_1'\}$	
35. $T_6 \leftarrow T_6 - T_1$	$\{W_1 - X_3\}$	
36. $T_2 \leftarrow T_2 \times T_6$	$\{Y_3 + A_1\}$	
37. $T_2 \leftarrow T_2 - T_5$	$\{Y_3\}$	
38. $T_6 \leftarrow T_6 + T_1$	$\{W_1\}$	
39. $T_6 \leftarrow T_6 - T_3$	$\{W_1 - X_2\}$	
40. $T_4 \leftarrow T_4 \times T_6$	$\{Y_2 + A_1\}$	
41. $T_4 \leftarrow T_4 - T_5$	$\{Y_2\}$	

2:
$\boldsymbol{R} = (T_1 : T_2)$, $\boldsymbol{Q} = (T_3 : T_4)$
3: **end function**

Algorithm 25 Co-Z conjugate-addition–addition with update (ZACAU) (register allocation)

Require: $\boldsymbol{P} = (X_1 : Y_1 : Z)$ and $\boldsymbol{Q} = (X_2 : Y_2 : Z)$ with $\mathrm{Z}(\boldsymbol{P}) = \mathrm{Z}(\boldsymbol{Q})$, and $C = (X_1 - X_2)^2$
Ensure: $(\boldsymbol{R}, \boldsymbol{S}, C) \leftarrow \mathrm{ZACAU}(\boldsymbol{P}, \boldsymbol{Q}, C)$ where $\boldsymbol{R} \leftarrow 2\boldsymbol{P} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{S} \leftarrow \boldsymbol{P} + \boldsymbol{Q} = (X_4 : Y_4 : Z_3)$ with $C \leftarrow (X_3 - X_4)^2$

1: **function** $\mathrm{ZACAU}(\boldsymbol{P}, \boldsymbol{Q}, C)$
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = Z$, $T_4 = X_2$, $T_5 = Y_2$, $T_6 = C$

1. $T_7 \leftarrow T_1 - T_4$	$\{X_1 - X_2\}$	
2. $T_3 \leftarrow T_3 \times T_7$	$\{Z'\}$	
3. $T_7 \leftarrow T_4 \times T_6$	$\{W_2\}$	
4. $T_6 \leftarrow T_6 \times T_1$	$\{W_1\}$	
5. $T_1 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$	
6. $T_4 \leftarrow {T_1}^2$	$\{\overline{D}\}$	
7. $T_4 \leftarrow T_4 - T_6$	$\{\overline{D} - W_1\}$	
8. $T_4 \leftarrow T_4 - T_7$	$\{X_2'\}$	
9. $T_1 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	
10. $T_1 \leftarrow {T_1}^2$	$\{D\}$	
11. $T_1 \leftarrow T_1 - T_6$	$\{D - W_1\}$	
12. $T_1 \leftarrow T_1 - T_7$	$\{X_1'\}$	
13. $T_7 \leftarrow T_7 - T_6$	$\{W_2 - W_1\}$	
14. $T_7 \leftarrow T_7 \times T_2$	$\{-A_1\}$	
15. $T_2 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	
16. $T_5 \leftarrow 2T_5$	$\{2Y_2\}$	
17. $T_5 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$	
18. $T_6 \leftarrow T_6 - T_4$	$\{W_1 - X_2'\}$	
19. $T_5 \leftarrow T_5 \times T_6$	$\{Y_2' + A_1\}$	
20. $T_5 \leftarrow T_5 + T_7$	$\{Y_2'\}$	
21. $T_6 \leftarrow T_4 + T_6$	$\{W_1\}$	
22. $T_6 \leftarrow T_6 - T_1$	$\{W_1 - X_1'\}$	
23. $T_2 \leftarrow T_2 \times T_6$	$\{Y_1' + A_1\}$	
24. $T_2 \leftarrow T_2 + T_7$	$\{Y_1'\}$	
25. $T_6 \leftarrow T_1 - T_4$	$\{X_1' - X_2'\}$	
26. $T_3 \leftarrow T_3 \times T_6$	$\{Z_3\}$	
27. $T_6 \leftarrow {T_6}^2$	$\{C'\}$	
28. $T_7 \leftarrow T_4 \times T_6$	$\{W_2'\}$	
29. $T_4 \leftarrow T_1 \times T_6$	$\{X_4\}$	
30. $T_6 \leftarrow T_2 - T_5$	$\{Y_1' - Y_2'\}$	
31. $T_7 \leftarrow T_4 - T_7$	$\{X_4 - W_2'\}$	
32. $T_5 \leftarrow T_2 \times T_7$	$\{Y_4\}$	
33. $T_2 \leftarrow {T_6}^2$	$\{D'\}$	
34. $T_1 \leftarrow T_2 + T_7$	$\{D' + X_4 - W_2'\}$	
35. $T_1 \leftarrow T_1 - T_4$	$\{D' - W_2'\}$	
36. $T_1 \leftarrow T_1 - T_4$	$\{X_3\}$	
37. $T_7 \leftarrow T_1 - T_4$	$\{X_3 - X_4\}$	
38. $T_6 \leftarrow T_6 - T_7$	$\{Y_1' - Y_2' + X_4 - X_3\}$	
39. $T_6 \leftarrow {T_6}^2$	$\{(Y_1' - Y_2' + X_4 - X_3)^2\}$	
40. $T_2 \leftarrow T_6 - T_2$	$\{(Y_1' - Y_2' + X_4 - X_3)^2 - D'\}$	
41. $T_6 \leftarrow {T_7}^2$	$\{C\}$	
42. $T_2 \leftarrow T_2 - T_6$	$\{(Y_1' - Y_2' + X_4 - X_3)^2 - D' - C\}$	
43. $T_5 \leftarrow 2T_5$	$\{2Y_4\}$	
44. $T_2 \leftarrow T_2 - T_5$	$\{Y_3\}$	
45. $T_1 \leftarrow 4T_1$	$\{4X_3\}$	
46. $T_2 \leftarrow 4T_2$	$\{4Y_3\}$	
47. $T_3 \leftarrow 2T_3$	$\{2Z_3\}$	
48. $T_4 \leftarrow 4T_4$	$\{4X_4\}$	
49. $T_5 \leftarrow 4T_5$	$\{8Y_4\}$	
50. $T_6 \leftarrow 16T_6$	$\{16C\}$	

2:
$\boldsymbol{R} = (T_1 : T_2 : T_3)$, $\boldsymbol{S} = (T_4 : T_5 : T_3)$, $C = T_6$
3: **end function**

Algorithm 26 (X, Y)-only co-Z conjugate-addition–addition with update (ZACAU′) (register allocation)

Require: $\boldsymbol{P'} = (X_1 : Y_1)$ and $\boldsymbol{Q'} = (X_2 : Y_2)$ with $\mathrm{Z}(\boldsymbol{P}) = \mathrm{Z}(\boldsymbol{Q})$, and $C = (X_1 - X_2)^2$
Ensure: $(\boldsymbol{R'}, \boldsymbol{S'}, C) \leftarrow \mathrm{ZACAU'}(\boldsymbol{P'}, \boldsymbol{Q'}, C)$ where $\boldsymbol{R'} \leftarrow (X_3 : Y_3)$ and $\boldsymbol{S'} \leftarrow (X_4 : Y_4)$ for some $\boldsymbol{R} = 2\boldsymbol{P} = (X_3 : Y_3 : Z_3)$ and $\boldsymbol{S} = \boldsymbol{P} + \boldsymbol{Q} = (X_4 : Y_4 : Z_3)$ with $C \leftarrow (X_3 - X_4)^2$

1: **function** ZACAU′($\boldsymbol{P'}, \boldsymbol{Q'}, C$)
$T_1 = X_1$, $T_2 = Y_1$, $T_3 = C$, $T_4 = X_2$, $T_5 = Y_2$

	1. $T_6 \leftarrow T_3 \times T_4$	$\{W_2\}$	24. $T_3 \leftarrow {T_3}^2$	$\{C'\}$
	2. $T_3 \leftarrow T_3 \times T_1$	$\{W_1\}$	25. $T_6 \leftarrow T_3 \times T_4$	$\{W_2'\}$
	3. $T_1 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$	26. $T_4 \leftarrow T_1 \times T_3$	$\{X_4\}$
	4. $T_4 \leftarrow {T_1}^2$	$\{\overline{D}\}$	27. $T_3 \leftarrow T_2 - T_5$	$\{Y_1' - Y_2'\}$
	5. $T_4 \leftarrow T_4 - T_3$	$\{\overline{D} - W_1\}$	28. $T_6 \leftarrow T_4 - T_6$	$\{X_4 - W_2'\}$
	6. $T_4 \leftarrow T_4 - T_6$	$\{X_2'\}$	29. $T_5 \leftarrow T_2 \times T_6$	$\{Y_4\}$
	7. $T_1 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	30. $T_2 \leftarrow {T_3}^2$	$\{D'\}$
	8. $T_1 \leftarrow {T_1}^2$	$\{D\}$	31. $T_1 \leftarrow T_2 + T_6$	$\{D' + X_4 - W_2'\}$
	9. $T_1 \leftarrow T_1 - T_3$	$\{D - W_1\}$	32. $T_1 \leftarrow T_1 - T_4$	$\{D' - W_2'\}$
	10. $T_1 \leftarrow T_1 - T_6$	$\{X_1'\}$	33. $T_1 \leftarrow T_1 - T_4$	$\{X_3\}$
	11. $T_6 \leftarrow T_6 - T_3$	$\{W_2 - W_1\}$	34. $T_6 \leftarrow T_1 - T_4$	$\{X_3 - X_4\}$
2:	12. $T_6 \leftarrow T_6 \times T_2$	$\{-A_1\}$	35. $T_3 \leftarrow T_3 - T_6$	$\{Y_1' - Y_2' + X_4 - X_3\}$
	13. $T_2 \leftarrow T_2 - T_5$	$\{Y_1 - Y_2\}$	36. $T_3 \leftarrow {T_3}^2$	$\{(Y_1' - Y_2' + X_4 - X_3)^2\}$
	14. $T_5 \leftarrow 2T_5$	$\{2Y_2\}$	37. $T_2 \leftarrow T_3 - T_2$	$\{(Y_1' - Y_2' + X_4 - X_3)^2 - D'\}$
	15. $T_5 \leftarrow T_2 + T_5$	$\{Y_1 + Y_2\}$	38. $T_3 \leftarrow {T_6}^2$	$\{C\}$
	16. $T_3 \leftarrow T_3 - T_4$	$\{W_1 - X_2'\}$	39. $T_2 \leftarrow T_2 - T_3$	$\{(Y_1' - Y_2' + X_4 - X_3)^2 - D' - C\}$
	17. $T_5 \leftarrow T_3 \times T_5$	$\{Y_2' + A_1\}$	40. $T_5 \leftarrow 2T_5$	$\{2Y_4\}$
	18. $T_5 \leftarrow T_5 + T_6$	$\{Y_2'\}$	41. $T_2 \leftarrow T_2 - T_5$	$\{Y_3\}$
	19. $T_3 \leftarrow T_3 + T_4$	$\{W_1\}$	42. $T_1 \leftarrow 4T_1$	$\{4X_3\}$
	20. $T_3 \leftarrow T_3 - T_1$	$\{W_1 - X_1'\}$	43. $T_2 \leftarrow 4T_2$	$\{4Y_3\}$
	21. $T_2 \leftarrow T_2 \times T_3$	$\{Y_1' + A_1\}$	44. $T_3 \leftarrow 16T_3$	$\{16C\}$
	22. $T_2 \leftarrow T_2 + T_6$	$\{Y_1'\}$	45. $T_4 \leftarrow 4T_4$	$\{4X_4\}$
	23. $T_3 \leftarrow T_1 - T_4$	$\{X_1' - X_2'\}$	46. $T_5 \leftarrow 4T_5$	$\{8Y_4\}$

$\boldsymbol{R'} = (T_1 : T_2)$, $\boldsymbol{S'} = (T_4 : T_5)$, $C = T_3$
3: **end function**