Published in A. Nitaj and D. Pointcheval, Ed., *Progress in Cryptology − AFRICACRYPT 2011*, vol. 6737 of *Lecture Notes in Computer Science*, pp. 170–187, Springer, 2011.

Memory-Constrained Implementations of Elliptic Curve Cryptography in Co-Z Coordinate Representation

Michael Hutter[1], Marc Joye[2], and Yannick Sierra[3]

[1] TU Graz, Institute for Applied Information Processing and Communications
Inffeldgasse 16a, 8010 Graz, Austria
`michael.hutter@iaik.tugraz.at`

[2] Technicolor, Security & Content Protection Labs
1 avenue de Belle Fontaine, 35576 Cesson-Sévigné Cedex, France
`marc.joye@technicolor.com`

[3] Oberthur Technologies
71-73 rue des Hautes Pâtures, 92726 Nanterre Cedex, France
`y.sierra@oberthur.com`

Abstract. It has been recently shown that sharing a common coordinate in elliptic curve cryptography implementations improves the performance of scalar multiplication. This paper presents new formulæ for elliptic curves over prime fields that provide efficient point addition and doubling using the Montgomery ladder. All computations are performed in a common projective Z-coordinate representation to reduce the memory requirements of low-resource implementations. In addition, all given formulæ make only use of *out-of-place* operations therefore insuring that it requires no additional memory for any implementation of the underlying finite-field operations whatsoever. Our results outperform existing solutions in terms of memory and speed and allow a fast and secure implementation suitable for low-resource devices and embedded systems.

Keywords: Public-key cryptography, elliptic curves, co-Z coordinates, out-of-place formulæ, Montgomery ladder, embedded systems.

1 Introduction

Elliptic curve cryptography (ECC) [17, 27] has gained much importance in the field of low-resource devices such as smart cards and Radio Frequency Identification (RFID) devices. The main benefits of ECC compared to traditional cryptographic primitives like RSA [30] are the significant improvements in terms of speed and memory. In fact, memory is one of the most expensive resources in the design of embedded systems which encourages the use of ECC on such platforms. In this paper, we present new formulæ for ECC implementations that allow very efficient (speed-wise and memory-wise) computations especially applicable to resource-constrained devices.

Among the most resource-consuming operation in ECC implementations is the scalar multiplication. A secret scalar k is multiplied with a point $\boldsymbol{P}$ on an elliptic curve $E(\mathbb{F}_q)$ resulting in the point $\boldsymbol{Q}$. This operation is used in many cryptographic primitives which rely on the intractability of solving the elliptic curve discrete logarithm problem (ECDLP), *i.e.* finding the discrete logarithm for $\boldsymbol{Q}$ with respect to the elliptic curve point $\boldsymbol{P}$.

In view of embedded systems, where memory and computational power are scarce resources, there exist many proposals to improve the scalar multiplication. One of the most prominent methods is the so-called Montgomery ladder [28]. First, it allows one to omit the y-coordinate of the involved elliptic curve points which lowers the memory requirements for low-resource designs. Second, it implicitly provides resistance against certain implementation attacks [16, 20, 24] which encourages its use in security-related applications.

Another improvement was proposed by Meloni [25] in 2007. He showed that points on an elliptic curve can be added quickly when they share a common coordinate, *e.g.* the projective Z-coordinate. Meloni applied the formula to specific Euclid addition chains to perform a scalar multiplication. However, the observation not only improves the speed of ECC implementations but reduces even the memory requirements by one coordinate as practically shown by Lee and Verbauwhede [22] over binary fields.

Recently, Goundar *et al.* [10] extended the idea of Meloni and provided formulæ over prime fields that can be even applied to classical binary scalar multiplication methods. They introduced a new operation (*conjugate co-Z addition*) that can be used together with the addition formula of Meloni to perform fast computations with points sharing the same Z-coordinate (co-Z arithmetic). However, the method has not been applied to the x-coordinate only version of the Montgomery ladder so far.

In this paper, we present new formulæ for elliptic curves over finite fields of characteristic $q \neq 2, 3$ that apply the co-Z method to the Montgomery ladder scalar multiplication. The given formulæ perform a differential addition-and-doubling operation of elliptic curve points using x-coordinates only, *i.e.* two projective X-coordinates of the involved points and a common Z-coordinate. It shows that the formulæ lead to very efficient scalar multiplications especially suitable to low-resource devices. In addition, we consider the practical constraint imposed by the implementations of both the modular multiplication and the modular squaring which may not support the result to be written *in-place*, that is overwriting one of the operands. This constraint is common in practice since it allows to save memory with many efficient implementations of those operations as discussed later and it can be imposed by the hardware accelerator when one is available. Unfortunately this typically implies the need of more memory than claimed in order to implement formulæ which have been designed with *in-place* operations. To our best knowledge, it is indeed the first paper that provides formulæ that use *out-of-place* operations guaranteing that no additional memory is necessary even when the finite-field arithmetic computations do not support *in-place* results. Our outcomes improve the state of the art in low-resource ECC implementations in terms of both memory and speed.

The rest of this paper is organized as follows. In Section 2, we briefly introduce elliptic curve cryptography. Section 3 describes different scalar-multiplication methods including the Montgomery ladder. Section 4 presents new formulæ for (differential) addition-and-doubling and projective coordinate recovery in co-Z coordinates. Section 5 discusses the difference between *in-place* versus *out-of-place* formulæ for ECC. In Section 6, the results are discussed in terms of security and performance. Conclusions are drawn in Section 7.

2 Preliminaries

This section introduces some elementary background on elliptic curves. We refer the reader to *e.g.* [11] for further details.

An elliptic curve E over a finite field $\mathbb{F}_q$ of characteristic $\neq 2, 3$ can be defined by the short Weierstraß equation

$$E : y^2 = x^3 + ax + b\,,$$

where $a, b \in \mathbb{F}_q$ are curve parameters satisfying $4a^3+27b^2 \neq 0$ and $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ represents a point on the elliptic curve. The set of all points on the elliptic curve together with the point at infinity $\boldsymbol{O}$ is denoted by $E(\mathbb{F}_q)$. It forms an (additively written) abelian group with the point at infinity $\boldsymbol{O}$ as the identity element.

Scalar multiplication. The main operation in elliptic curve cryptography (ECC) is the scalar multiplication, $\boldsymbol{Q} = k\boldsymbol{P}$, where $\boldsymbol{P}$ and $\boldsymbol{Q}$ are points on the curve E and k is a scalar such that $0 \leq k < \mathrm{ord}_E(\boldsymbol{P})$. The security of ECC primitives relies on the intractability to solve the elliptic curve discrete logarithm problem (ECDLP), *i.e.* determining k from $\boldsymbol{P}$ and $\boldsymbol{Q}$.

Point representation. The scalar multiplication uses two basic operations that are addition and doubling of points. The points can be represented in several coordinate systems. Points in affine coordinates are represented by two coordinates x and y but involve the computation of inversions in $\mathbb{F}_q$ which are relatively expensive operations. Due to these reasons, most implementations represent the points in projective coordinates. In homogeneous projective coordinates, each affine point (x, y) is represented by three coordinates (X, Y, Z) where $x = X/Z$ and $y = Y/Z$. Another coordinate system that is widely used in practice is the Jacobian projective coordinate system. There, the relation $x = X/Z^2$ and $y = Y/Z^3$ is used to represent the points. The curve equation in Jacobian coordinates becomes $E : Y^2 = X^3 + aXZ^4 + bZ^6$.

Point addition. Let $\boldsymbol{P_1} = (X_1, Y_1, Z_1)$ and $\boldsymbol{P_2} = (X_2, Y_2, 1)$ be two points represented in Jacobian projective coordinates on the curve. Then the sum $\boldsymbol{P_1} + \boldsymbol{P_2} = (X_3, Y_3, Z_3)$ (also known as *mixed* sum since $Z_2 = 1$), is given by

$$\begin{cases} X_3 = (Y_2{Z_1}^3 - Y_1)^2 - (X_2{Z_1}^2 - X_1)^2(X_1 + X_2 Z_1^2) \\ Y_3 = (Y_2{Z_1}^3 - Y_1)(X_1(X_2{Z_1}^2 - X_1)^2 - X_3) - Y_1(X_2{Z_1}^2 - X_1)^3 \\ Z_3 = (X_2{Z_1}^2 - X_1)Z_1 \end{cases} . \quad (1)$$

The formula for point doubling, $2\boldsymbol{P_1} = (X_4, Y_4, Z_4)$, is given by

$$\begin{cases} X_4 = (3{X_1}^2 + a{Z_1}^4)^2 - 8X_1{Y_1}^2 \\ Y_4 = (3{X_1}^2 + a{Z_1}^4)(4X_1{Y_1}^2 - X_3) - 8{Y_1}^4 \\ Z_4 = 2Y_1Z_1 \end{cases} . \quad (2)$$

To evaluate the costs of the given formulæ we denote by M the cost of a field multiplication and by S the cost of a field squaring. For multiplications with fixed parameters such as the curve parameters, we use the notation $\mathsf{M}_\star$ (*e.g.* M_a, M_b). Additions and subtractions are later assumed to have the same complexity and are represented by add.

Evaluating formulæ (1) and (2) in terms of computational cost shows that a point addition needs $7\mathsf{M} + 4\mathsf{S}$ if $Z_2 = 1$ [11]. Point doubling can be performed with $4\mathsf{M} + 4\mathsf{S}$ or $1\mathsf{M} + 8\mathsf{S} + 1\mathsf{M}_a$. For comparability reasons, we use the same performance metric as in the dedicated website Explicit Formulas Database (EFD) [6].

Co-$\boldsymbol{Z}$ arithmetic. In 2007, Meloni proposed new point addition and doubling formulæ in Jacobian coordinates where the two involved points share the same Z-coordinate [25]. We refer to this coordinate system as the *co-Z coordinate system*. When the two points satisfy this condition, the addition of two points can be evaluated much faster than an addition in Jacobian coordinates (actually even faster than a doubling operation in Jacobian coordinates). Let $\boldsymbol{P_1} = (X_1, Y_1, Z)$ and $\boldsymbol{P_2} = (X_2, Y_2, Z)$ the two points that share the same Z-coordinate, then the sum of the two points, $\boldsymbol{P_1} + \boldsymbol{P_2} = \boldsymbol{P_3} = (X_3, Y_3, Z_3)$, is given by

$$\begin{cases} X_3 = (Y_2 - Y_1)^2 - X_2(X_2 - X_1)^2 - X_1(X_2 - X_1)^2 \\ Y_3 = (Y_2 - Y_1)[X_1(X_2 - X_1)^2 - X_3] - Y_1(X_2 - X_1)^3 \\ Z_3 = Z(X_2 - X_1) \end{cases} . \quad (3)$$

This addition only requires $5\mathsf{M} + 2\mathsf{S}$. As observed in [25], the given formulæ have the advantage of providing an equivalent representation $\boldsymbol{P_1'}$ of the point $\boldsymbol{P_1} = (X_1, Y_1, Z)$ such that the points $\boldsymbol{P1'}$ and $\boldsymbol{P3}$ have the same Z-coordinate value. Namely $\boldsymbol{P_1'} = (X_1\lambda^2, Y_1\lambda^3, Z\lambda)$ with $\lambda = (X_2 - X_1)$, is calculated without any additional cost since the coordinates are already computed as intermediate values in the addition formula (cf. Eq. (3)).

3 Scalar Multiplication Methods

There exist several algorithms to perform the scalar multiplication.

One of the most common methods is the *double-and-add* algorithm (a.k.a. left-to-right binary method), shown in Algorithm 1. It takes the binary representation of the scalar k as an input and processes the bits from left to right. A point doubling operation is performed at every iteration whereas point addition is only performed if the bit value, k_i, is 1.

Algorithm 1 Double-and-add

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$, with $k_{n-1} \neq 0$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

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

The method has the advantage that it provides a very efficient point multiplication but suffers from that it may leak information about the secret scalar k via physical side-channels [20, 24]. In Simple Power Analysis (SPA) attacks, an adversary tries to recover the scalar k by measuring the power-consumption traces during scalar multiplication. If a difference between the operations of point addition and point doubling can be observed in the traces, then the scalar k is revealed bit-by-bit.

In [4], Coron proposes a simple countermeasure that involves a dummy point addition operation if the scalar bit is set to 0. The so-called *double-and-add always* method actually prevents SPA attacks but becomes vulnerable to safe-error attacks, as shown in [35]. A fault can be induced during the computation and an adversary can check whether the final result is correct or not. If the fault is injected during a dummy addition, the result is still correct and the corresponding bit of the scalar is 0. If the result is incorrect, the scalar bit is 1.

Another scalar-multiplication method that is commonly used is known as the *Montgomery ladder* [28] and is depicted in Algorithm 2. The method presents several advantages for cryptographic applications.

First, the Montgomery ladder implicitly offers security against implementation attacks [16]. Since it performs the same curve operations in every loop iteration, an attacker cannot distinguish individual bits of the secret scalar by simply observing a side-channel trace and so prevents SPA-type attacks. Furthermore, the Montgomery ladder has a very regular structure and does not use dummy operations. This prevents fault-injection based safe-error attacks.

Algorithm 2 Montgomery ladder

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$, with $k_{n-1} \neq 0$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

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

Second, group operations can be performed without the need of y-coordinates. Montgomery originally applied the technique to special (Montgomery form) elliptic curves as a way to speed up the elliptic curve factoring method. The technique was subsequently generalized to Weierstraß form curves [2, 7, 15, 14].

Let $\boldsymbol{P_1} = (x_1, y_1)$ and $\boldsymbol{P_2} = (x_2, y_2)$ be two points on the elliptic curve $E : y^2 = x^3 + ax + b$ and x_D the x-coordinate of their difference $\boldsymbol{D} = \boldsymbol{P_2} - \boldsymbol{P_1}$. Then the x-coordinate of the sum $\boldsymbol{P_1} + \boldsymbol{P_2}$, say x_3, is given by

$$x_3 = \frac{2(x_1 + x_2)(x_1 x_2 + a) + 4b}{(x_1 - x_2)^2} - x_D \ . \tag{4}$$

Alternatively, the x-coordinate of $\boldsymbol{P_1} + \boldsymbol{P_2}$ can be obtained in a multiplicative way as

$$x_3 = \frac{-4b(x_1 + x_2)(x_1 x_2 - a)^2}{x_D (x_1 - x_2)^2} \ . \tag{4'}$$

The x-coordinate of $2\boldsymbol{P_2}$, say x_4, can be expressed from the x-coordinate of $\boldsymbol{P_2}$ as

$$x_4 = \frac{({x_2}^2 - a)^2 - 8b x_2}{4({x_2}^3 + a x_2 + b)} \ . \tag{5}$$

It is worth noticing that the Montgomery ladder keeps invariant the difference of the involved points throughout the entire scalar multiplication. Indeed, from the description in Algorithm 2, it is easily seen that $\boldsymbol{R_1} - \boldsymbol{R_0} = (\boldsymbol{R_1} + \boldsymbol{R_0}) - 2\boldsymbol{R_0}$ when $b = 0$, and $\boldsymbol{R_1} - \boldsymbol{R_0} = 2\boldsymbol{R_1} - (\boldsymbol{R_0} + \boldsymbol{R_1})$ when $b = 1$. Hence, $\boldsymbol{D} := \boldsymbol{R_1} - \boldsymbol{R_0} = \boldsymbol{P}$. Consequently, $\boldsymbol{R_1}$ will contain the value of $(k+1)\boldsymbol{P}$ at the end of the algorithm. When the calculation is performed using x-coordinates only, this allows one to recover the y-coordinate of $k\boldsymbol{P}$. Letting (x_1, y_1) the coordinates of $\boldsymbol{Q} = k\boldsymbol{P}$, (x_D, y_D) the coordinates of $\boldsymbol{P}$ and x_2 the x-coordinate of $(k+1)\boldsymbol{P}$, one has

$$y_1 = \frac{2b + (a + x_D x_1)(x_D + x_1) - x_2(x_D - x_1)^2}{2y_D} \ . \tag{6}$$

This is useful for cryptographic schemes needing the y-coordinate of the resulting point; for example, in the verification of an ECDSA digital signature [29].

4 New x-Coordinate Only Formulæ

This section presents new x-coordinate only formulæ for Weierstraß elliptic curves. We first provide the formulæ for addition and doubling of points in the co-Z coordinate representation. Second, we give formulæ for efficient differential addition-and-doubling in the same coordinate representation. Third, we discuss optimizations when applying dynamic ECC parameters and give appropriate formulæ to recover the full coordinates of the output point.

Let $\boldsymbol{P_1} = (X_1, Y_1, Z)$ and $\boldsymbol{P_2} = (X_2, Y_2, Z)$ be two points on the Weierstraß elliptic curve $E : Y^2Z = X^3 + aXZ^2 + bZ^3$ in *homogeneous*[†] projective coordinates that share the same Z-coordinate. Then, the x-coordinate of the addition of the two points, $\mathrm{x}(\boldsymbol{P_1} + \boldsymbol{P_2}) = (X_3, Z_3)$, can be evaluated as

$$\begin{cases} X_3 = 2(X_1 + X_2)(X_1X_2 + aZ^2) + 4bZ^3 - x_D Z(X_1 - X_2)^2 \\ Z_3 = Z(X_1 - X_2)^2 \end{cases} , \qquad (7)$$

where $\boldsymbol{D} = \boldsymbol{P_2} - \boldsymbol{P_1} = (x_D, y_D)$ is the difference of the points $\boldsymbol{P_1}$ and $\boldsymbol{P_2}$ in affine coordinates. Note that the formula performs the point addition with x-coordinates only, thus no Y-coordinate is used. The point addition needs $5\mathsf{M} + 2\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$ to get the resulting x-coordinate $\mathrm{x}(\boldsymbol{P_1} + \boldsymbol{P_2})$.

The x-coordinate of a point doubling operation, $\mathrm{x}(2\boldsymbol{P_2}) = (X_4, Z_4)$, needs $4\mathsf{M} + 3\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$ and can be evaluated as

$$\begin{cases} X_4 = ({X_2}^2 - aZ^2)^2 - 8bZ^3X_2 \\ Z_4 = Z[4X_2({X_2}^2 + aZ^2) + 4bZ^3] \end{cases} . \qquad (8)$$

Applying formulæ (7) and (8) to the Montgomery ladder needs three additional multiplications to project the resulting x-coordinates $\mathrm{x}(\boldsymbol{R_0}) = (X_3, Z_3)$ and $\mathrm{x}(\boldsymbol{R_1}) = (X_4, Z_4)$ to a common Z-coordinate. An equivalent representation for $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ can be obtained by evaluating

$$X_1' = X_3Z_4 \,, \quad X_2' = X_4Z_3 \,, \text{ and } \quad Z' = Z_3Z_4 \,,$$

resulting in $\boldsymbol{R_0} \cong (X_1', Z')$ and $\boldsymbol{R_1} \cong (X_2', Z')$ sharing the same Z-coordinate. The total complexity for one Montgomery ladder loop iteration is therefore $12\mathsf{M} + 5\mathsf{S} + 2\mathsf{M}_a + 2\mathsf{M}_{4b}$. In the following, we show how to reduce the complexity for differential addition-and-doubling to only $\underline{9\mathsf{M} + 5\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}}$.

4.1 Differential Addition-And-Doubling

By combining the projective formulæ given by Eqs. (7) and (8) and class equivalences to have the same Z-coordinate, we obtain

[†] Previous works considered Jacobian coordinates when applying co-Z arithmetic on elliptic curves over fields of characteristic $\neq 2, 3$.

$$\begin{cases} X_1' = V[2(X_1 + X_2)(X_1X_2 + aZ^2) + 4bZ^3 - x_DZU] \\ X_2' = U[({X_2}^2 - aZ^2)^2 - 8bZ^3X_2] \\ Z' = UVZ \end{cases} , \tag{9}$$

where $U = (X_1 - X_2)^2$ and $V = 4X_2({X_2}^2 + aZ^2) + 4bZ^3$. The points $\mathrm{x}(\boldsymbol{R_0}) = (X_1, Z)$ and $\mathrm{x}(\boldsymbol{R_1}) = (X_2, Z)$ get added and doubled resulting in the points $\mathrm{x}(\boldsymbol{R_0'}) = (X_1', Z')$ and $\mathrm{x}(\boldsymbol{R_1'}) = (X_2', Z')$. The formula reduces the complexity to $10\mathsf{M} + 4\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$.

This can be further optimized by replacing the multiplication X_1X_2 involved in the previous formula with the equivalent expression $({X_1}^2 + {X_2}^2 - (X_1 - X_2)^2)/2$. The term can be multiplied with the leading factor 2 so that we finally obtain

$$\begin{cases} X_1' = V[(X_1 + X_2)({X_1}^2 + {X_2}^2 - U + 2aZ^2) + 4bZ^3 - x_DZU] \\ X_2' = U[({X_2}^2 - aZ^2)^2 - 8bZ^3X_2] \\ Z' = UVZ \end{cases} . \tag{10}$$

This latter formula can be evaluated with $9\mathsf{M} + 5\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$. Note that the formula overwrites the input coordinates X_1, X_2, and Z with the output variables X_1', X_2', and Z'. This avoids additional memory allocations for the output variables and avoids variable copying since the output variables serve as input variables for the next Montgomery loop iteration. Furthermore, the resulting points $\mathrm{x}(\boldsymbol{R_0}) = (X_1', Z')$ and $\mathrm{x}(\boldsymbol{R_1}) = (X_2', Z')$ share the same Z-coordinate and do not need any further updates. A detailed implementation is provided in Algorithm 5 (Appendix A).

4.2 $(\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z})$ Recovery

We now give the formula for the recovery of the full projective coordinates for output point $\boldsymbol{Q} = k\boldsymbol{P}$, from the x-coordinates $\boldsymbol{R_0} = (X_1, Z)$ and $\boldsymbol{R_1} = (X_2, Z)$ in co-Z representation available in memory at the end of the Montgomery ladder. First, we transform Eq. (6) from affine to projective coordinates and set $x_i = X_i/Z$ and $y_i = Y_i/Z$ $(i \in \{1, 2\})$. Then, we can calculate the representation of output point $\boldsymbol{Q}$ in the projective coordinates $\boldsymbol{Q} \cong (X_1', Y_1', Z_1')$ with

$$\begin{cases} X_1' = DX_1A \\ Y_1' = 2[(CX_1 + aA)(C + X_1) - X_2(C - X_1)^2] + 4bB \\ Z_1' = DB \end{cases} , \tag{11}$$

where $A = Z^2$, $B = ZA$, $C = x_DZ$, $D = 4y_D$. X_1, X_2, and Z are the coordinates of the elliptic curve points after scalar multiplication and $\boldsymbol{D} = (x_D, y_D)$ represents the invariant of the Montgomery ladder in affine coordinates (namely, input point $\boldsymbol{P}$). The given formula needs $8\mathsf{M} + 2\mathsf{S} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$. The affine coordinates of output point $\boldsymbol{Q}$ can then be calculated by one inversion and two multiplications, *i.e.*, $\boldsymbol{Q} = (x_1, y_1) = (X_1' \cdot {Z_1'}^{-1}, Y_1' \cdot {Z_1'}^{-1})$. See Algorithm 7 (Appendix A) for a detailed implementation.

4.3 Optimizations for Dynamic ECC Parameters

If the curve parameters such as a, b are not fixed by the implementation and are chosen dynamically, the formula given in Eq. (10) can be optimized. In this case, the curve parameters have to be handled in RAM and their memory allocation can therefore be re-used as working space as soon as they are not needed. The following formulæ allows to save one register compared to the implementation of Eq. (10) with a and b permanently occupying a full register in RAM. By initializing three additional coordinates $T_a = aZ^2$, $T_b = 4bZ^3$, and $T_D = x_D Z$, we can evaluate

$$\begin{cases} T'_D = T_D W \\ T'_a = T_a W^2 \\ T'_b = T_b W^3 \\ X'_1 = V[(X_1 + X_2)({X_1}^2 + {X_2}^2 - U + 2T_a) + T_b] - T'_D \\ X'_2 = U[({X_2}^2 - T_a)^2 - 2X_2 T_b] \end{cases} \tag{12}$$

to perform a differential addition-and-doubling operation, where $U = (X_1 - X_2)^2$, $V = 4X_2({X_2}^2 + T_a) + T_b$, and $W = UV$. The given formula reduces the memory requirements by one working register and increases the performance by $1\mathsf{M}$ if the relation $\mathsf{M}_a = \mathsf{M}_b = 1\mathsf{M}$ is given (however, in practice, one has usually the relation $\mathsf{M}_a + \mathsf{M}_b = 1\mathsf{M}$; see § 6.2) . Note that the formula does not involve either a, b, or x_D nor an explicit Z-coordinate throughout the scalar multiplication. See Algorithm 6 (Appendix A) for a detailed implementation.

The full coordinates (X'_1, Y'_1, Z'_1) can be recovered with $10\mathsf{M} + 3\mathsf{S}$ by evaluating

$$\begin{cases} X'_1 = 4 y_D x_D {T_D}^2 X_1 \\ Y'_1 = {x_D}^3 [T_b + 2(T_D X_1 + T_a)(X_1 + T_D) - 2X_2(X_1 - T_D)^2] \\ Z'_1 = 4 y_D {T_D}^3 \end{cases} . \tag{13}$$

See Algorithm 8 (Appendix A) for a detailed implementation.

5 In-Place vs. Out-of-Place Formulæ

Most descriptions of the elliptic-curve operations presented in the literature have claims of memory requirements and performances that assume that the finite-field operations can be performed *in-place*. That means that one source operand of the operation may be overwritten by the resulting value during the execution, *e.g.*

$$R_1 \leftarrow R_1 \circ R_2 ,$$

where $R_1 \in \mathbb{F}_p$ and $R_2 \in \mathbb{F}_p$ are variables that store the source operands and R_1 is overwritten by the resulting value after execution of an operation $\circ$. In contrast, operations that do not overwrite the input operands are referred to as *out-of-place* operations, *e.g.*

$$R_3 \leftarrow R_1 \circ R_2 \,,$$

where $R_3 \in \mathbb{F}_p$ is an additional variable that stores the result of the operation.

In general, there exist several ways to implement modular operations in software and hardware. Most implementations use multi-precision arithmetic to process the large integer operands. That means that each operand is represented as a multiple-word data structure, *i.e.* $a = (a_{t-1}, \ldots, a_1, a_0)$ and $b = (b_{t-1}, \ldots, b_1, b_0)$, where t denotes the number of words. A 160-bit addition operation, for instance, that runs on a 16-bit processor, performs therefore ten additions by loading the input operands from memory, adding the two operands, and storing the result back to the memory. A subtraction is done in the same way, performing machine word subtractions instead of additions. However, during the computation both operations process each word of the operands sequentially and can thus perform the operation *in-place* at no cost in terms of memory or computational efficiency [11].

In contrast, modular multiplication (and squaring) can be implemented in several ways. Basically, we can distinguish between *separated* and *integrated* modular multiplication [19, 18]. Separated modular multiplications perform the multiplication first and apply the reduction afterwards. In this approach, the result of the multiplication is stored in a temporary variable R_m which is then reduced in a separated step, *e.g.*

$$\begin{aligned} R_m &\leftarrow R_1 \times R_2, \\ R_1 &\leftarrow R_m \pmod{p}. \end{aligned}$$

This approach needs additional memory to store the temporary variable $R_m \in [0, 2^{2Wt})$, where W denotes the number of bits of a word (*i.e.* typically 8, 16, 32, or 64 bits).

The integrated (or interleaved) modular multiplication approach alternates between multiplication and reduction. There, partial products get reduced during the multiplication which avoids storing the double-sized result R_m and thus reduces the memory requirements significantly to the size of about the modulus $p \in [0, 2^{Wt})$ [1, 33, 34, 19, 12, 21]. However, for both multiplication types, the input operands cannot be overwritten with the resulting words because they are used not only once but multiple times throughout the algorithm. Therefore, implementations that allow *in-place* multiplications (and squarings) may need either an extra buffer to store the intermediate result $2^{Wt} \leq R_m < 2^{2Wt}$ or save the input operand to be overwritten during the computation. Formulæ for point operations in elliptic curves that involve *in-place* operations are thus very likely to require more memory in practice than claimed.

In this work, we propose *out-of-place* formulæ that use different source and destination variables to perform the modular multiplication and squaring operations. This guarantees that no additional memory is needed to perform the computation neither for software nor hardware implementations and that our formulæ will therefore meet our claims in all contexts.

6 Discussion

6.1 Security Analysis

The resistance to side-channel attacks and fault attacks is essential for the implementation of cryptographic applications in embedded device. The given formulæ allow the use of traditional countermeasures against such attacks without disadvantages. As described in Section 3, the Montgomery ladder is well suited to the implementation of the scalar-multiplication method since it is resistant against SPA attacks [20, 24] as well as safe-error attacks [35].

In addition, there exist several proposals to protect the Montgomery ladder against statistical attacks such as Differential Power Analysis (DPA) [20, 24]. One cheap but effective countermeasure against these attacks is the use of Randomized Projective Coordinates (RPC) as proposed by Coron [4]. In our context, this countermeasure can be implemented by randomizing the intermediate points of the Montgomery ladder since they are represented in projective-coordinate representation as seen in Section 4. This can be done in Algorithm 2 at the cost of only two multiplications by randomizing the initial coordinates of the points $\boldsymbol{R_0}$ and $\boldsymbol{R_1}$ which are represented by the triplet $\{X_1, X_2, Z\}$ such that $\mathrm{x}(\boldsymbol{R_0}) = (X_1, Z)$ and $\mathrm{x}(\boldsymbol{R_1}) = (X_2, Z)$. Then, given a random value λ, the point $\mathrm{x}(\boldsymbol{P}) = (x_P, 1)$ is randomized to $\mathrm{x}(\boldsymbol{P'}) = (\lambda x_P, \lambda)$ for the initialization of $\mathrm{x}(\boldsymbol{R_0})$ and $\mathrm{x}(\boldsymbol{R_1})$ as follows:

$$\begin{cases} \mathrm{x}(\boldsymbol{R_1}) \leftarrow (\lambda x_P, \lambda) = \mathrm{x}(\boldsymbol{P'}) \\ \mathrm{x}(\boldsymbol{R_1}) \leftarrow \mathrm{doubling}(\boldsymbol{R_1}) = \mathrm{x}(2\boldsymbol{P'}) \\ \mathrm{x}(\boldsymbol{R_0}) \leftarrow (Z x_P, Z) = \mathrm{x}(\boldsymbol{P'}) \end{cases} . \tag{14}$$

This effectively randomizes every intermediate value during scalar multiplication and makes therefore DPA attacks ineffective. Note that the doubling can computed using the differential algorithms 4, 5, and 6 to save the need for a dedicated function.

In order to thwart fault injections during the scalar multiplication [32, 31] a countermeasure that checks the resulting point can be applied. Checking that $x^3 + ax + b$ is a square may seem conceivable, unfortunately that may not detect if the point belongs to the twist curve instead of the original curve and would leave the implementation vulnerable to attacks such as the one introduced by Fouque *et al.* [8]. Another check consists in verifying that the coordinates of the resulting point satisfy the curve equation, in which case the recovery of the y-coordinate is required. However that can be done in an efficient way with projective coordinates *i.e.* $Z(Y^2 - bZ^2) = X(X^2 + aZ^2)$ [5]. This countermeasure effectively protects against fault attacks on the Montgomery ladder even when implemented with x-coordinate only formulæ [8].

Algorithm 3 shows the proposed Montgomery ladder in projective co-Z coordinate system using RPC [4] and Point-Validity Check [5]. `AddDblCoZ` denotes the implemented differential addition-and-doubling operation using Algorithms 4, 5, or 6. `RecoverFullCoordinatesCoZ` recovers the coordinates using Algorithm 7 or 8.

Algorithm 3 Montgomery ladder in projective co-Z coordinate system using RPC [4] and Point-Validity Check [5].

Input: $\boldsymbol{P} \in E(\mathbb{F}_q)$ and $k = (k_{n-1}, \ldots, k_0)_2 \in \mathbb{N}$, with $k_{n-1} \neq 0$
Output: $\boldsymbol{Q} = k\boldsymbol{P}$

```
1: $\{X_1, X_2, Z\} \leftarrow$ AddDblCoZ($\{0, \lambda x_P, \lambda\}$)
2: $X_1 \leftarrow x_P \cdot Z$
3: for $i = n-2$ downto 0 do
4:     $b \leftarrow k_i$;
5:     $\{X_{2-b}, X_{1+b}, Z\} \leftarrow$ AddDblCoZ($\{X_{2-b}, X_{1+b}, Z\}$)
6: end for
7: $\{X, Y, Z\} \leftarrow$ RecoverFullCoordinatesCoZ($\{X_1, X_2, Z\}$)
8: $Z(Y^2 - bZ^2) \overset{?}{=} X(X^2 + aZ^2)$
9: return $\{X, Y, Z\}$
```

6.2 Performance Analysis

We now compare our formulæ with existing differential addition-and-doubling formulæ. Comparing one formula with another is not straightforward because the complexity ratio of the field-arithmetic operations involved may vary according to the underlying implementation as well as the usage context. Hence we provide the global complexity of each algorithm along with figures corresponding to some assumptions that are made based on on-the-field experience and previous works.

Thus, we first adopt the common assumption that the squaring operation is faster than a multiplication with the weighting $1\mathsf{S} = 0.8\mathsf{M}$ [11, 6]. The cost of additions and subtractions are usually neglected when evaluating the complexity of the formulæ. However, according to several previous works [23, 26, 9] it may be relevant to take these operations into account since in practice they have reported ratios from $1\mathsf{add} = 0.1\mathsf{M}$ up to $1\mathsf{add} = 0.3\mathsf{M}$. Hence, in the following we will consider both cases, by first considering additions negligible and then the worst case where $1\mathsf{add} = 0.3\mathsf{M}$. Besides, a special case can also be made for the multiplications involving the curve parameters and especially the parameter a because several standardized curves have a set to -3. In any case, this assumption can be applied without loss of generality because a curve isomorphism can be used to reduce a to a small relative integer [13, §§ A.9.5 and A.10.4] (see also [3]). Subsequently, we will assume that a multiplication with a takes 2 additions, *i.e.* $1\mathsf{M}_a = 2\mathsf{add}$. Rescaling a curve to reduce the value of a also modifies the value b in a way that it is unlikely in the general case to have both a and b small. Therefore a multiplication with b (or any fixed pre-computed multiple *e.g.* $4b$ denoted M_{4b}) is considered as a regular modular multiplication of cost $\mathsf{M}_b = 1\mathsf{M}$.

In the following, we compare different low-memory scalar multiplication formulæ first sorted by performances in Table 1 and then sorted by memory requirements in Table 2.

Table 1 shows the efficiency of the formulæ we proposed in Section 4. Algorithm 5 is more efficient than any previous works found in literature since

Table 1. Complexity of scalar multiplications per bit of scalar.

Method	Costs[⋆]	M/bit[⋆⋆]	M/bit[⋆⋆⋆]
Algorithm 6	$\mathbf{10M + 5S + 13add}$	**14**	**17.9**
Algorithm 5	$\mathbf{9M + 5S + 1M_{a} + 1M_{4b} + 14add}$	**14**	**18.8**
Izu *et al.* [14]	$10M + 4S + 2M_a + 1M_b + 18add$	14.2	20.8
Goundar *et al.* [10]	$8M + 7S + 3M_a + 1M_b + 18add$	14.6	21.8
Algorithm 4	$\mathbf{11M + 4S + 1M_{a} + 1M_{4b} + 14add}$	**15.2**	**20.0**
Fischer *et al.* [7]	$10M + 5S + 2M_a + 2M_b + 14add$	16	21.4

⋆ The explicit formulæ are given in Appendix A.

⋆⋆ $M_b = 1M$; $S = 0.8M$; $1M_a \simeq 0$; $1add \simeq 0$ (negligible)

⋆⋆⋆ $M_b = 1M$; $S = 0.8M$; $1M_a = 2add$; $1add = 0.3M$

$1S \geq 0.5M$ [26, §§ 14.18]. In practice, Algorithm 6 is also more efficient because rescaling general curves implies at best $1M_a + 1M_b \geq 1M$. The performance improvement is significant (up to 14 % less multiplications per bit) when adopting the usual assumption that $1S \geq 0.8M$ (cf [11, 6]) and $1M_b = 1M$. One can also remark that the benefits of our approach increases when the squaring is performed using the multiplication instead of a dedicated implementation (for program or hardware saving) as well as when the secondary operations such as additions and subtractions are not negligible (as observed in practice).

Table 2 lists the memory requirements of the scalar multiplication methods. For constrained devices where the elliptic-curve parameters x_D, a, b or $4b$ are hard-coded or stored in read-only memory, Algorithm 4 provides the lowest memory requirements. It allows to implement the scalar multiplication with only 7 working registers combined with the memory gain offered by the implementation of *out-of-place* field operations as described in Section 5.

In a context where the curve parameters cannot be set during the design-time of the device or if they can not be processed directly from the read-only memory as it is in the case with most cryptographic accelerators, Algorithm 6 becomes

Table 2. Memory requirements of scalar multiplications.

Method	Working registers	In-place[⋆] memory	Constants	Total
Algorithm 4	**7 reg.**	-	$\boldsymbol{\{x_D, a, 4b\}}$	**10 reg.**
Izu *et al.* [14]	7 reg.	+1 reg.	$\{x_D, a, b\}$	11 reg.
Goundar *et al.* [10]	7 reg.	+1 reg.	$\{x_D, a, b\}$	11 reg.
Algorithm 5	**8 reg.**	-	$\boldsymbol{\{x_D, a, 4b\}}$	**11 reg.**
Fischer *et al.* [7]	8 reg.	+1 reg.	$\{x_D, a, 4b\}$	12 reg.
Algorithm 6	**10 reg.**	-	-	**10 reg.**

⋆ In-place operations require additional memory to perform multiple-precision arithmetic operations (see Section 5).

equivalent in terms of memory requirement to Algorithm 4 while being faster as shown in Table 1.

7 Conclusion

In this paper, we presented new formulæ for fast and memory-wise scalar multiplication on elliptic curves over prime fields. The proposed formulæ use *out-of-place* operations, namely the source and destination variables of finite-field multiplications are always different. This guarantees that neither additional memory is needed nor additional operations have to be executed to perform multiple-precision arithmetic operations in both software or hardware implementations. Furthermore, the given formulæ outperform existing solutions by using a co-Z coordinate representation. The formulæ can be applied on general elliptic curves and allow the integration of conventional countermeasures against implementation attacks. They can be efficiently applied in low-resource implementations of RFIDs, smart cards, and other embedded systems.

Acknowledgements. The work has been supported in part by the Austrian Government funded project *PIT* established by the research program FIT-IT Trust in IT Systems under the project number 825743.

References

1. G. R. Blakely. A computer algorithm for calculating the product ab modulo m. *IEEE Transactions on Computers*, 32(5):497 – 500, 1983.
2. E. Brier and M. Joye. Weierstraß elliptic curves and side-channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography (PKC 2002)*, volume 2274 of *Lecture Notes in Computer Science*, pages 335–345. Springer-Verlag, 2002.
3. E. Brier and M. Joye. Fast point multiplication on elliptic curves through isogenies. In M. Fossorier, T. Høholdt, and A. Poli, editors, *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes*, volume 2643 of *Lecture Notes in Computer Science*, pages 43–50. Springer-Verlag, 2003.
4. J.-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems (CHES '99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer-Verlag, 1999.
5. N. M. Ebeid and R. Lambert. Securing the elliptic curve Montgomery ladder against fault attacks. In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography (FDTC 2009)*, pages 46–50. IEEE Computer Society, 2009.
6. Explicit-formulas database (EFD). `http://www.hyperelliptic.org/EFD/`.
7. W. Fischer, C. Giraud, E. W. Knudsen, and J.-P. Seifert. Parallel scalar multiplication on general elliptic curves over $\mathbb{F}_p$ hedged against non-differential side-channel attacks. Cryptology ePrint Archive, Report 2002/007, 2002.
8. P.-A. Fouque, R. Lercier, D. Réal, and F. Valette. Fault attack on elliptic curve Montgomery ladder implementation. In L. Breveglieri et al., editors, *Fault Diagnosis and Tolerance in Cryptography (FDTC 2008)*, pages 92–98. IEEE Computer Society, 2008.

9. C. Giraud and V. Verneuil. Atomicity improvement for elliptic curve scalar multiplication. In D. Gollmann, J.-L. Lanet, and J. Iguchi-Cartigny, editors, *Smart Card Research and Advanced Application (CARDIS 2010)*, volume 6035 of *Lecture Notes in Computer Science*, pages 80–101. Springer-Verlag, 2010.
10. R. R. Goundar, M. Joye, and A. Miyaji. Co-Z addition formulæ and binary ladders. In S. Mangard and F.-X. Standaert, editors, *Cryptographic Hardware and Embedded Systems – CHES 2010*, volume 2523 of *Lecture Notes in Computer Science*, pages 65–79. Springer-Verlag, 2010.
11. D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography.* Springer, 2004.
12. D. Hein, J. Wolkerstorfer, and N. Felber. ECC is ready for RFID – A proof in silicon. In *4th Workshop on RFID Security 2008 (RFIDsec 08)*, July 9–11, 2008.
13. IEEE Std 1363-2000. IEEE Standard Specifications for Public-Key Cryptography. IEEE Computer Society, Aug. 2000.
14. T. Izu, B. Möller, and T. Takagi. Improved elliptic curve multiplication methods reistant against side-channel attacks. In A. Menezes and P. Sarkar, editors, *Progress in Cryptology – INDOCRYPT 2002*, volume 2551 of *Lecture Notes in Computer Science*, pages 296–313. Springer-Verlag, 2002.
15. T. Izu and T. Takagi. A fast parallel elliptic curve multiplication resistant against side channel attacks. In D. Naccache and P. Paillier, editors, *Public Key Cryptography (PKC 2002)*, volume 2274 of *Lecture Notes in Computer Science*, pages 280–296. Springer-Verlag, 2002.
16. M. Joye and S.-M. Yen. The Montgomery powering ladder. In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, 2003.
17. N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.
18. Ç. K. Koç. RSA Hardware Implementation. Technical report, RSA Laboratories, RSA Data Security, Inc. 100 Marine Parkway, Suite 500 Redwood City, CA 94065-1031, 1995.
19. Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16:26–33, 1996.
20. P. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – CRYPTO '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.
21. Y. K. Lee, K. Sakiyama, L. Batina, and I. Verbauwhede. Elliptic-curve-based security processor for RFID. *IEEE Transactions on Computers*, 57(11):1514–1527, 2008.
22. Y. K. Lee and I. Verbauwhede. A compact architecture for Montgomery elliptic curve scalar multiplication processor. In S. Kim et al., editors, *Information Security Applications (WISA 2007)*, volume 4867 of *Lecture Notes in Computer Science*, pages 115–127. Springer-Verlag, 2007.
23. C. H. Lim and H. S. Hwang. Fast implementation of elliptic curve arithmetic in GF(p^n). In H. Imai and Y. Zheng, editors, *Public Key Cryptography (PKC 2000)*, volume 1751 of *Lecture Notes in Computer Science*, pages 405–421. Springer-Verlag, 2000.
24. S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks – Revealing the Secrets of Smartcards.* Springer, 2007.

25. N. Meloni. New point addition formulæ for ECC applications. In C. Carlet and B. Sunar, editors, *Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer-Verlag, 2007.
26. A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.
27. V. S. Miller. Use of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology – CRYPTO '85*, volume 218 of *Lecture Notes in Computer Science*, pages 417–426. Springer-Verlag, 1986.
28. P. L. Montgomery. Speeding up the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
29. National Institute of Standards and Technology. FIPS 186-3 – Digital Signature Standard (DSS). `http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf`, June 2009.
30. R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
31. J.-M. Schmidt. *Implementation Attacks – Manipulating Devices to Reveal Their Secrets*. PhD thesis, Graz University of Technology, 2009.
32. S. P. Skorobogatov. *Semi-Invasive Attacks – A New Approach to Hardware Security Analysis*. PhD thesis, University of Cambridge, 2005. Available online at `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-630.pdf`.
33. K. R. Sloan. Comments on "A computer algorithm for calculating the product AB modulo M". *IEEE Transactions on Computers*, 34:290–292, 1985.
34. J. Wolkerstorfer. Dual-field arithmetic unit for GF(p) and GF(2^m). In B. S. Kaliski Jr., Ç. K. Koç, and C. Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2002*, volume 2523 of *Lecture Notes in Computer Science*, pages 500–514. Springer-Verlag, 2003.
35. S.-M. Yen and M. Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.

A Appendix

In the following, the explicit formulæ for differential addition-and-doubling and full projective coordinate recovery in co-Z coordinates are given. Algorithm 4 gives the formulæ for differential addition-and-doubling using $11\mathsf{M}+4\mathsf{S}+14\mathsf{add}+1\mathsf{M}_a+1\mathsf{M}_{4b}$ and $7+\{x_D,a,4b\}$ registers. Algorithm 5 show the formulæ using $9\mathsf{M}+5\mathsf{S}+14\mathsf{add}+1\mathsf{M}_a+1\mathsf{M}_{4b}$ and $8+\{x_D,a,4b\}$ registers. Algorithm 6 gives the formulae using $10\mathsf{M}+5\mathsf{S}+13\mathsf{add}$ and 10 registers (without involving curve parameters). The recovery of the full projective coordinates is given in Algorithm 7 which uses $8\mathsf{M}+2\mathsf{S}+8\mathsf{add}+1\mathsf{M}_a+1\mathsf{M}_{4b}$ and $7+\{x_D,y_D,a,4b\}$ registers. All given formulæ provide the *out-of-place* property so that input operands are not overwritten by output operands of squaring and multiplication operations. Furthermore, the elliptic curve parameter b is always used in a quadruple representation so that it can be pre-computed and pre-stored as $4b$. In addition, all formulae update the input variables with the resulting values using the same memory location. This avoids memory copies or pointer manipulations in hardware or software implementations. Finite field operations are denoted by $\times$ for multiplication, $\cdot^2$ for squaring, $+$ for addition, and $-$ for subtraction.

Algorithm 4 Out-of-place differential addition-and-doubling in projective co-Z coordinate system using $11\mathsf{M} + 4\mathsf{S} + 14\mathsf{add} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$ and $7 + \{x_D, a, 4b\}$ registers.

```
Require: $X_1$, $X_2$, $Z$, $x_D$, $a$, $4b$
Ensure: $X_1$, $X_2$, $Z$
1:   1. $R_1 \leftarrow X_1 \times X_2$
     2. $R_3 \leftarrow Z^2$
     3. $R_4 \leftarrow Z \times R_3$
     4. $R_2 \leftarrow a \times R_3$
     5. $R_1 \leftarrow R_1 + R_2$
     6. $X_1 \leftarrow X_1 + X_2$
     7. $R_3 \leftarrow X_1 \times R_1$
     8. $X_1 \leftarrow X_1 - X_2$
     9. $X_1 \leftarrow X_1 - X_2$
     10. $R_1 \leftarrow 4b \times R_4$
     11. $R_4 \leftarrow {X_1}^2$
     12. $X_1 \leftarrow R_4 \times Z$
     13. $R_3 \leftarrow R_3 + R_3$
     14. $R_3 \leftarrow R_3 + R_1$
     15. $Z \leftarrow X_2 \times R_4$
     16. $R_4 \leftarrow R_1 \times X_2$
     17. $R_1 \leftarrow {X_2}^2$
     18. $R_2 \leftarrow R_1 + R_2$
     19. $R_1 \leftarrow R_1 + R_1$
     20. $X_2 \leftarrow x_D \times X_1$
     21. $R_3 \leftarrow R_3 - X_2$
     22. $X_2 \leftarrow R_1 \times R_2$
     23. $X_2 \leftarrow X_2 + X_2$
     24. $R_2 \leftarrow R_2 - R_1$
     25. $R_1 \leftarrow R_4 + R_4$
     26. $R_4 \leftarrow X_2 + R_4$
     27. $X_2 \leftarrow {R_2}^2$
     28. $R_1 \leftarrow X_2 - R_1$
     29. $X_2 \leftarrow R_1 \times Z$
     30. $Z \leftarrow X_1 \times R_4$
     31. $X_1 \leftarrow R_3 \times R_4$
2: return $(X_1, X_2, Z)$
```

Algorithm 5 Out-of-place differential addition-and-doubling in projective co-Z coordinate system using $9\mathsf{M} + 5\mathsf{S} + 14\mathsf{add} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$ and $8 + \{x_D, a, 4b\}$ registers.

```
Require: $X_1$, $X_2$, $Z$, $x_D$, $a$, $4b$
Ensure: $X_1$, $X_2$, $Z$
1:   1. $R_2 \leftarrow Z^2$
     2. $R_3 \leftarrow a \times R_2$
     3. $R_1 \leftarrow Z \times R_2$
     4. $R_2 \leftarrow 4b \times R_1$
     5. $R_1 \leftarrow {X_2}^2$
     6. $R_5 \leftarrow R_1 - R_3$
     7. $R_4 \leftarrow {R_5}^2$
     8. $R_1 \leftarrow R_1 + R_3$
     9. $R_5 \leftarrow X_2 \times R_1$
     10. $R_5 \leftarrow R_5 + R_5$
     11. $R_5 \leftarrow R_5 + R_5$
     12. $R_5 \leftarrow R_5 + R_2$
     13. $R_1 \leftarrow R_1 + R_3$
     14. $R_3 \leftarrow {X_1}^2$
     15. $R_1 \leftarrow R_1 + R_3$
     16. $X_1 \leftarrow X_1 - X_2$
     17. $X_2 \leftarrow X_2 + X_2$
     18. $R_3 \leftarrow X_2 \times R_2$
     19. $R_4 \leftarrow R_4 - R_3$
     20. $R_3 \leftarrow {X_1}^2$
     21. $R_1 \leftarrow R_1 - R_3$
     22. $X_1 \leftarrow X_1 + X_2$
     23. $X_2 \leftarrow X_1 \times R_1$
     24. $X_2 \leftarrow X_2 + R_2$
     25. $R_2 \leftarrow Z \times R_3$
     26. $Z \leftarrow x_D \times R_2$
     27. $X_2 \leftarrow X_2 - Z$
     28. $X_1 \leftarrow R_5 \times X_2$
     29. $X_2 \leftarrow R_3 \times R_4$
     30. $Z \leftarrow R_2 \times R_5$
2: return $(X_1, X_2, Z)$
```

Algorithm 6 Out-of-place differential addition-and-doubling in projective co-Z coordinate system using $10\mathsf{M} + 5\mathsf{S} + 13\mathsf{add}$ and 10 registers.

```
Require: $X_1$, $X_2$, $T_D = x_D Z$, $T_a = aZ^2$, $T_b = 4bZ^3$
Ensure: $X_1$, $X_2$, $T_D$, $T_a$, $T_b$
1:  1. $R_2 \leftarrow X_1 - X_2$
    2. $R_1 \leftarrow {R_2}^2$
    3. $R_2 \leftarrow {X_2}^2$
    4. $R_3 \leftarrow R_2 - T_a$
    5. $R_4 \leftarrow R_3^2$
    6. $R_5 \leftarrow X_2 + X_2$
    7. $R_3 \leftarrow R_5 \times T_b$
    8. $R_4 \leftarrow R_4 - R_3$
    9. $R_5 \leftarrow R_5 + R_5$
    10. $R_2 \leftarrow R_2 + T_a$
    11. $R_3 \leftarrow R_5 \times R_2$
    12. $R_3 \leftarrow R_3 + T_b$
    13. $R_5 \leftarrow X_1 + X_2$
    14. $R_2 \leftarrow R_2 + T_a$
    15. $R_2 \leftarrow R_2 - R_1$
    16. $X_2 \leftarrow {X_1}^2$
    17. $R_2 \leftarrow R_2 + X_2$
    18. $X_2 \leftarrow R_5 \times R_2$
    19. $X_2 \leftarrow X_2 + T_b$
    20. $X_1 \leftarrow R_3 \times X_2$
    21. $X_2' \leftarrow R_1 \times R_4$
    22. $R_2 \leftarrow R_1 \times R_3$
    23. $R_3 \leftarrow R_2 \times T_b$
    24. $R_4 \leftarrow {R_2}^2$
    25. $R_1 \leftarrow T_D \times R_2$
    26. $R_2 \leftarrow T_a \times R_4$
    27. $T_b \leftarrow R_3 \times R_4$
    28. $X_1 \leftarrow X_1 - R_1$
    29. $T_D \leftarrow R_1$
    30. $T_a \leftarrow R_2$
2:  return $(X_1, X_2, T_D, T_a, T_b)$
```

Algorithm 7 Out-of-place (X, Y, Z)-recovery in projective co-Z coordinate system using $8\mathsf{M} + 2\mathsf{S} + 8\mathsf{add} + 1\mathsf{M}_a + 1\mathsf{M}_{4b}$ and $7 + \{x_D, y_D, a, 4b\}$ registers.

```
Require: $X_1$, $X_2$, $Z$, $x_D$, $y_D$, $a$, $4b$
Ensure: $X_1$, $X_2$, $Z$
1:  1. $R_1 \leftarrow x_D \times Z$
    2. $R_2 \leftarrow X_1 - R_1$
    3. $R_3 \leftarrow {R_2}^2$
    4. $R_4 \leftarrow R_3 \times X_2$
    5. $R_2 \leftarrow R_1 \times X_1$
    6. $R_1 \leftarrow X_1 + R_1$
    7. $X_2 \leftarrow Z^2$
    8. $R_3 \leftarrow a \times X_2$
    9. $R_2 \leftarrow R_2 + R_3$
    10. $R_3 \leftarrow R_2 \times R_1$
    11. $R_3 \leftarrow R_3 - R_4$
    12. $R_3 \leftarrow R_3 + R_3$
    13. $R_1 \leftarrow y_D + y_D$
    14. $R_1 \leftarrow R_1 + R_1$
    15. $R_2 \leftarrow R_1 \times X_1$
    16. $X_1 \leftarrow R_2 \times X_2$
    17. $R_2 \leftarrow X_2 \times Z$
    18. $Z \leftarrow R_2 \times R_1$
    19. $R_4 \leftarrow 4b \times R_2$
    20. $X_2 \leftarrow R_4 + R_3$
2:  return $(X_1, X_2, Z)$
```

Algorithm 8 Out-of-place (X, Y, Z)-recovery in projective co-Z coordinate system using $10\mathsf{M} + 3\mathsf{S} + 8\mathsf{add}$ and $9 + \{x_D, y_D, a, 4b\}$ registers.

```
Require: $X_1$, $X_2$, $T_D = x_D Z$, $T_a = aZ^2$, $T_b = 4bZ^3$, $x_D$, $y_D$
Ensure: $X_1$, $X_2$, $Z$
1:  1. $R_1 \leftarrow T_D \times X_1$
    2. $R_2 \leftarrow R_1 + T_a$
    3. $R_3 \leftarrow X_1 + T_D$
    4. $R_4 \leftarrow R_2 \times R_3$
    5. $R_3 \leftarrow X_1 - T_D$
    6. $R_2 \leftarrow R_3^2$
    7. $R_3 \leftarrow R_2 \times X_2$
    8. $R_4 \leftarrow R_4 - R3$
    9. $R_4 \leftarrow R_4 + R_4$
    10. $R_4 \leftarrow R_4 + T_b$
    11. $R_2 \leftarrow T_D^2$
    12. $R_3 \leftarrow X_1 \times R_2$
    13. $R_1 \leftarrow x_D \times R_3$
    14. $R_3 \leftarrow y_D + y_D$
    15. $R_3 \leftarrow R_3 + R_3$
    16. $X_1 \leftarrow R_3 \times R_1$
    17. $R_1 \leftarrow R_2 \times T_D$
    18. $Z \leftarrow R_3 \times R_1$
    19. $R_2 \leftarrow x_D^2$
    20. $R_3 \leftarrow R_2 \times x_D$
    21. $X_2 \leftarrow R_3 \times R_4$
2:  return $(X_1, X_2, Z)$
```