Compact Elliptic Curve Scalar Multiplication with a Secure Generality

YAOAN JIN¹,a) ATSUKO MIYAJI¹,2,b)

Received: December 9, 2019, Accepted: June 1, 2020

Abstract: Elliptic curve cryptosystems (ECCs) are widely used because of their short key size. ECCs can ensure sufficient security with shorter keys, using less memory to reduce parameters. Hence, ECCs are typically used in IoT devices. The dominant computation of an ECC is scalar multiplication \( Q = kP \) for \( P \in \mathbb{E}(\mathbb{F}_q) \). Thus, the security and efficiency of scalar multiplication are paramount. To render secure ECCs, complete addition (CA) formulae can be employed for secure scalar multiplication algorithms. However, this requires significant memory; thus, it is not suitable for compact devices. Several types of coordinates exist for elliptic curves such as affine, Jacobian, Projective and so on. The CA formulae are not based on affine coordinates and, thus, require considerable memory. In this study, we achieve a compact ECC by focusing on affine coordinates. In fact, affine coordinates are highly advantageous in terms of memory but require many if statements for scalar multiplication owing to exceptional points. We propose two scalar multiplication algorithms with the extended affine formulae to delete some exceptional inputs for scalar multiplication. Our two algorithms reduce memory cost up to 37% or 21%. In many cases such as NIST elliptic curves, our two algorithms are the most efficient if \( \frac{2}{q} < 12 \), for the ratio of computational cost of inversion and multiplication. The experiment shows that our algorithms can compute the elliptic curve scalar multiplication correctly and efficiently.

Keywords: elliptic curve scalar multiplication, side channel attack (SCA), exception-free addition formulae

1. Introduction

Elliptic curve cryptosystems (ECCs) are widely used because of their short key size. ECCs can ensure sufficient security with shorter keys, using less memory to reduce parameters. Hence, ECCs are typically used for Internet-of-things (IoT) devices [2]. The dominant computation of ECCs is scalar multiplication \( Q = kP \) for \( P \in \mathbb{E}(\mathbb{F}_q) \). Thus, the security and efficiency of the scalar multiplication are paramount. Studies of secure elliptic curve scalar multiplication algorithms can be divided into two categories. The first research direction is to find efficient and secure scalar multiplication algorithms [3], [4], [5], [6], [7]. The second direction is to find efficient and secure coordinates with addition formulae [8], [9], [10], [11], [12]. Several types of coordinates for elliptic curves exist (such as affine, Jacobian, or Projective). Although it appears that we only need to combine scalar multiplication algorithms with coordinates, it is not simple because some scalar multiplications require branches when the addition formulae are applied to them. Branches introduce simple power analysis (SPA). For example, in the case of affine or Jacobian coordinates, both doubling and addition formulae exist for two inputs of \( P \) and \( Q \). That is, when the scalar multiplication algorithm employs addition formulae in affine or Jacobian coordinates, we need to verify whether the two input points are equal. In fact, not only the condition \( P = Q \) but also other input points such as \( O + P, P - P, \) and \( 2P = O \) become exceptional inputs. Hence, researchers have investigated complete addition (CA) formulae [8], [9], [10], which can be computed for any two input points. Further, new methods have been proposed by combining a scalar multiplication algorithm with CA formulae to protect the elliptic curve scalar multiplication from a side channel attack (SCA) [13]. CA formulae operate well to exclude such branches. However, CA formulae are not efficient from the memory and computational standpoints. Particularly, CA formulae are not based on affine coordinates and, thus, require significant memory.

In this study, we achieve a compact ECC by focusing on affine coordinates. Although affine coordinates are highly advantageous in terms of memory, they require if statements for scalar multiplication owing to exceptional points. We adopt two approaches. First, we examine a scalar multiplication with the input point and scalar \( k \) by defining three notions: generality of \( k \) (a scalar multiplication algorithm can operate on any input scalar \( k \)), secure generality (a scalar multiplication algorithm can resist SCA with generality of \( k \)), and executable coordinates (coordinates with elliptic curve addition formulae, which can be used to a scalar multiplication algorithm without introducing if statements). Subsequently, we demonstrate that Joye’s right-to-left (RL) 2-ary algorithm (Algorithm 2) [4] satisfies the secure
generality but that Joye’s double-add algorithm (Algorithm 1) [3] does not satisfy secure generality. Further, we verify coordinates that become executable. Second, we improve Joye’s RL 2-ary algorithm to reduce exceptional point inputs and the limitations of input k. We extend the affine to delete some exceptional inputs for scalar multiplication. Subsequently, we propose a new scalar multiplication, Algorithm 9, by combining our improved Joye’s RL 2-ary algorithm with affine formulae and our extended affine formulae. In this paper, combinations of affine formulae and extended affine formulae are called (extended) affine in short. We propose Algorithm 10 to enhance the efficiency of Algorithm 9 by 2-bit scanning using the affine double and quadruple formulae (DQ) [14] that can compute both 2P and 4P simultaneously with only one inversion computation. Finally, we do a theoretical analysis of our algorithms and implement them. Algorithms 9 and 10 with (extended) affine reduce memory cost by 37% and 21% compared with Algorithm 2 with CA formulae, respectively. As for computational cost, we evaluate all algorithms by estimating the number of modulo multiplication (M), modulo square (S), multiplication with parameters a and b (ma and mb), addition (A), and inversion (I).

2.2 Complete Addition (CA) Formulae

Izu and Takagi proposed the x-only differential addition and doubling formulae [8], which proved to be exceptional only if both input coordinates of x and y are 0 [13]. These addition formulae are applied to the Montgomery ladder in which after the computation of the x-coordinate, the y-coordinate can be recovered by the formula of Ebeid and Lambert [15].

Renes et al. proposed complete addition formulae for prime order elliptic curves [9]. Based on the theorems of Bosma and Lenstra [16], the complete addition formulae for an elliptic curve E(\mathbb{F}_p) without points of order two can be obtained. Note that E(\mathbb{F}_p) with prime order excludes the points of order two. Thus, we can use the complete addition formulae on E(\mathbb{F}_p) with prime order. The authors also mentioned that if the complete addition formulae were used in an application, their efficiency could be improved based on specific parameters and further computation. However, they remain costly.

Wronski presented a new idea to obtain complete addition formulae for an elliptic curve E_{SW}/\mathbb{F}_p in the short Weierstrass form [10]. We can expand E_{SW}/\mathbb{F}_p to E_{SW}/\mathbb{F}_{p^2} and subsequently obtain the isomorphism \phi from E_{SW}/\mathbb{F}_{p^2} to the twisted Hessian curve E_{H}/\mathbb{F}_{p^2} when both conditions of 3|\#E_{SW}/\mathbb{F}_{p^2} and q \equiv 1 (mod 3) are satisfied. Using the arithmetic on E_{H}/\mathbb{F}_{p^2}, we can compute the elliptic curve scalar multiplication more quickly. After kP was computed on the twisted Hessian curve E_{H}/\mathbb{F}_{p^2}, we can use \phi^{-1} to transform kP to a real result kP on E_{SW}/\mathbb{F}_{p^2}. Further, the addition formulae on the twisted Hessian curve E_{H}/\mathbb{F}_{p^2} may be complete when parameter a of the twisted Hessian curve E_{H}/\mathbb{F}_{p^2} : ax^3 + y^3 + 1 = dx^y is not a cube in \mathbb{F}_q.

Table 1 summarizes the addition formulae including the CA formulae, where M, S, I, and A are the costs for one field multiplication, square, inversion, and addition, respectively. Further, ma and mb are the costs for multiplication to a and b, respectively.

Assuming that S = 0.8M and ignoring the computational cost of ma, mb, and A, the computational cost of ADD + DBL in the CA formulae is 24M. Subsequently, the computational cost of ADD + DBL in affine is more efficient than that in the CA for-
Algorithm 1 Joyce’s double-add algorithm [3]

Input: \( P \in E(\mathbb{F}_p^2), k = \sum_{i=0}^{t-1} k_i 2^i \)
Output: \( kP \)
Uses: \( R[0], R[1] \)
1: \( R[0] \leftarrow O \)
2: \( R[1] \leftarrow P \)
3: for \( i = 0 \) to \( t - 1 \) do
4: \( R[1 + k_i] \leftarrow 2R[1 - k_i] + R[k_i] \)
5: end for
6: return \( R[0] \)

Algorithm 2 Joyce’s RL 2-ary algorithm [4]

Input: \( P \in E(\mathbb{F}_p^2), k = \sum_{i=0}^{t-1} k_i 2^i \)
Output: \( kP \)
Uses: \( A, R[1], R[2] \)
Initialization
1: \( R[1] \leftarrow O, R[2] \leftarrow O, A \leftarrow P \)
Main loop
2: for \( i = 0 \) to \( t - 2 \) do
3: \( R[1 + k_i] \leftarrow R[1 + k_i] + A, A \leftarrow 2A \)
4: end for
Aggregation and final correction
5: \( A \leftarrow (k_{t-1} - 1)A + R[1] + 2R[2] \)
6: \( A \leftarrow A + P \)
7: return \( A \)

Table 1

| Method          | Conditions               | ADD          | DBL          | Memory |
|-----------------|--------------------------|--------------|--------------|--------|
| x-only addition | \( x \) or \( z \)-coordinate \( \neq 0 \) | \( 8M + 2S \) | \( 5M + 3S \) | 10     |
| Complete addition | \( 2 \not\exists E(\mathbb{F}_p) \) | \( 12M + 3ma + 2mb + 23A \) | \( 12M + 3ma + 2mb + 23A \) | 15     |
| Affine          | -                        | \( 2M + S + I \) | \( 2M + 2S + I \) | 5      |
| Jacobian        | -                        | \( 11M + 5S \) | \( M + 8S \) | 8      |

3. Exceptional Inputs in Scalar Multiplication

This section analyzes two algorithms (Algorithms 1–2) with an input scalar \( k = \sum_{i=0}^{t-1} k_i 2^i \) (in binary) and an elliptic curve point \( P \) from the following three aspects: generality of \( k \), secure generality, and executable coordinates.

3.1 Generality of \( k \)

We define the generality of \( k \) as follows. The scalar multiplication should compute \( kP \) for \( k \in \mathbb{Z}/N\mathbb{Z} \), where \( k \in [0,1] \) and \( N \) is the order of \( P \). Subsequently, it includes a case where the MSB of \( k \) is zero (\( k_{t-1} = 0 \)). We say that a scalar multiplication satisfies the generality of \( k \) if it can operate for any \( k \in \mathbb{Z}/N\mathbb{Z} \) with \( k_{t-1} = 0 \) or \( k_{t-1} = 1 \). Let us investigate whether Algorithms 1–2 satisfy the generality of input scalar \( k \). The Joye’s double-add algorithm (Algorithm 1) can operate for any input scalar \( k \in \mathbb{Z}/N\mathbb{Z} \) with \( k_{t-1} = 0 \) or \( k_{t-1} = 1 \). It is obvious that Algorithm 1 can compute \( kP \) correctly when \( k_{t-1} = 1 \). Algorithm 1 scans the scalar from right and reads “0”s at the end if \( k_{t-1} = 0 \). The “0”s read at the end do not change the value saved in \( R[0] \), which is the final correct computation result. In summary, Algorithm 1 can compute \( kP \) correctly with any input scalar \( k \in \mathbb{Z}/N\mathbb{Z} \) with \( k_{t-1} = 0 \) or \( k_{t-1} = 1 \).

Joye’s RL \( m \)-ary algorithm satisfies generality of \( k \), implying that it can compute \( kP \) for any input \( k \in \mathbb{Z}/N\mathbb{Z} \) with \( k_{t-1} = 0 \) or \( k_{t-1} = 1 \). The proof is given in Appendix A.1. We herein focus on the case of \( m = 2 \), which is described by Algorithm 2.

3.2 Secure Generality

We define the notion of the secure generality added to the generality of \( k \) as follows. If a scalar multiplication can compute \( kP \) regularly without dummy operations satisfying generality of \( k \) for \( k \in \mathbb{Z}/N\mathbb{Z} \) with \( k_{t-1} = 0 \) or \( k_{t-1} = 1 \), where \( N \) is the order of \( P \), then we say that such an algorithm satisfies the secure generality.

Algorithm 2 executes the same computations of addition and doubling without any dummy operation for every bit of scalar. It is regular without dummy operations for any \( k \); thus, it satisfies the secure generality. Algorithm 1 also executes the same computations of addition and doubling without any dummy operations until the final input bit \( k_{t-1} \) of a scalar \( k \). Its final step in the main loop becomes a dummy operation when processing \( k_{t-1} = 0 \). In detail, Algorithm 1 reads “0”s at the end if \( k_{t-1} = 0 \). Subsequently, the computation \( R[1] \leftarrow 2R[1] + R[0] \) becomes a dummy operation. Thus, we can know whether the scalar begins with “0”s by inserting safe-error to \( R[1 - k_i] \leftarrow 2R[1 - k_i] + R[k_i] \).

If the result does not change, then the MSB of the scalar is “0”. Thus, Algorithm 1 does not satisfy the secure generality.

3.3 Executable Coordinates

Let us define the notion of coordinates in a scalar multiplication algorithm. If the coordinates can be executed for an algorithm for \( k \in \mathbb{Z}/N\mathbb{Z} \) without exceptional inputs, we say that coordinates are executable coordinates for the algorithm, where \( N \) is the order of \( P \) in \( E(\mathbb{F}_p) \). This notion is important because even if a scalar multiplication algorithm satisfies secure generality, we must choose executable coordinates.

Let us investigate the executable coordinates in Algorithm 1. Algorithm 1 requires addition or doubling formulae with \( O \). This is why neither affine nor Jacobian coordinates are executable. Let us investigate Algorithm 2. Algorithm 2 contains exceptional inputs \( k \), \( R[1] \) and \( R[2] \) are initialized as \( O \), and \( A \) is initialized as \( P \) in Step 1. In the main loop, \( O + P \) appears independent of \( k \) in Step 3. It is obvious that \( O + P, P + P, \) and \( -P + P \) are computed when \( k = 1, 2, 0 \) in the aggregation and final correction, respectively. In summary, Algorithm 2 has to compute addition with \( O \) independent to \( k, P + P \) if \( k = 2 \), and \( P - P \) if \( k = 0 \). Neither the affine nor Jacobian coordinates can compute all of \( O + P, P + P, \) and \( -P + P \). Meanwhile, CA formulae [9] are executable coordinates. As shown in Section 2, we must sacrifice computational
and memory cost if we use CA formulae.

We herein focus on Algorithm 2, as it satisfies the secure generality of \( k \). Specifically, we improve it by adapting it to affine coordinates that requires a small memory. Jacobian coordinates are also executable for our new Algorithms 9–10.

4. Extended Affine Addition Formulae

Affine formulae are advantageous because of less memory usage. The computational cost, however, depends on the ratio of inversion cost to multiplication cost.

The detailed algorithms are shown in Algorithms 4 and 5. It is noteworthy that both Algorithms 4 and 5 can retain the value of the input point of \( P \), which can be used continually as the next input. Affine formulae have exceptional points. \( O \) cannot be represented explicitly, while it is described as a point at infinity. Thus, affine formulae cannot compute \( O + P = O \) and \( P - P = O \), or \( 2P = O \). The addition formula cannot compute \( P + P \), which can only be computed by the doubling formula. When implementing affine formulae, branches are required to avoid such exceptional points. We want to fully utilize affine formulae because they reduce memory. Scalar multiplication algorithms should satisfy the secure generality in Section 3; thus, they are suitable for any \( k \in \mathbb{Z}/N\mathbb{Z} \), which includes a special case of \( k = 0 \). Algorithm 2 satisfies the secure generality but the affine coordinates are not executable on them.

Thus, we extend the affine formulae in such a way that they can compute exceptional computations of \( P - P = O \) and \( 2P = O \). The corresponding operations are shown in Algorithms 6 and 7, which can compute \( P - P = O \) and \( 2P = O \) when \( E(F_p) \) does not include a point \((0, 0)\). For example, \( E(F_p) \) without two-torsion points satisfies the condition, including the prime order elliptic curve on the Weierstrass form. Importantly, both Algorithms 6 and 7 retain the value of the input point of \( P \) similar to Algorithms 4 and 5. Let us explain our idea of the extended affine formulae. The inversion of \( a (\mod p) \) can be computed by the extended Euclidean algorithm (Algorithm 3), \( Ecd(a, p) \), or Fermat’s little theorem, \( Fermat(a, p) = a^{p-2} (\mod p) \). Interestingly, Algorithm 3 outputs 0 with inputs \( a = 0 \) and any \( p \); Fermat’s little theorem computes \( 0^{p-2} = 0 \) with inputs \( a = 0 \) and any \( p \); that is, both are executable for a special input of “0”. Therefore, Algorithms 6 and 7 compute \((x_2 - x_1)^{-1}(2y_1)^{-1}\) in the beginning by extended Euclidean algorithm or Fermat’s little theorem and execute the remaining parts. Subsequently, the results for ordinary inputs \( P \) and \( Q \) are the same as those of Algorithms 4 and 5, respectively. Furthermore, the results for the exceptional inputs of \( P = P - P \) and \( 2P = O \) can be given as \((0, 0)\), which is assumed as \( O = (0, 0) \).

The extended affine addition formula is transformed from the original affine addition formula by extracting the factor of \((x_2 - x_1)^{-1}\). The computational cost of Algorithm 6 is \( 6M + S + I \) and its memory cost is seven field elements. The extended affine doubling formula is transformed from the original affine doubling formula by extracting \((2y_1)^{-1}\). The computational cost of Algorithm 7 is \( 4M + 4S + I \) and its memory cost is also seven field elements.

Remark 1 Neither Algorithm 4 nor 5 can output \( P = P - P = (0, 0) \) or \( 2P = (0, 0) \), even if an inversion of \( x_2 - x_1 \) or \( 2y_1 \) is computed by the Euclidean algorithm or Fermat’s little theorem.

\[\text{Algorithm 3} \text{ Extend Euclidean Algorithm}\]
\[
\text{Input: } x_0, x_1, x_2, y_0, y_1, y_2, a, p, r, q
\]
\[
\text{Output: } a^{-1} \text{ mod } p
\]
\[
1. \quad x_0 = 1, y_0 = 0, x_1 = 0, y_1 = 1
2. \quad \text{while } p \neq 0 \text{ do}
3. \quad \quad \quad r = a \mod p, q = a \div p
4. \quad \quad \quad x_2 = x_0 - q \cdot x_1, y_2 = y_0 - q \cdot y_1
5. \quad \quad \quad a = p, p = r
6. \quad \quad \quad x_0 = x_1, x_1 = x_2
7. \quad \quad \quad y_0 = y_1, y_1 = y_2
8. \quad \quad \quad \text{end while}
9. \quad \quad \quad \text{return } x_0
\]

\[\text{Algorithm 4} \text{ Affine addition formula}\]
\[
\text{Input: } P = (x_1, y_1) \text{ and } Q = (x_2, y_2)
\]
\[
\text{Output: } P + Q
\]
\[
1. \quad t_0 \leftarrow (x_2 - x_1)^{-1}
2. \quad y_2 \leftarrow y_2 - y_1
3. \quad t_0 \leftarrow t_0y_2
4. \quad y_2 \leftarrow t_0^2 - x_1 - x_2
5. \quad x_2 \leftarrow (x_1 - y_2)0 - y_1
6. \quad \text{return } (x_1, y_1), (y_2, x_2)
\]

\[\text{Algorithm 5} \text{ Affine doubling formula}\]
\[
\text{Input: } P = (x_1, y_1)
\]
\[
\text{Output: } 2P
\]
\[
1. \quad t_0 \leftarrow 3x_1^2 + a, t_1 \leftarrow (2y_1)^{-1}
2. \quad y_2 \leftarrow y_2 - y_1
3. \quad t_0 \leftarrow t_0t_2
4. \quad t_1 \leftarrow t_1^2 - 2x_1
5. \quad t_2 \leftarrow (x_1 - t_1)t_0 - y_1
6. \quad \text{return } (x_1, y_1), (t_1, t_2)
\]

\[\text{Algorithm 6} \text{ Extended affine addition}\]
\[
\text{Input: } P = (x_1, y_1) \text{ and } Q = (x_2, y_2)
\]
\[
\text{Output: } P + Q
\]
\[
1. \quad t_0 \leftarrow (x_2 - x_1)^{-1}
2. \quad y_2 \leftarrow y_2 - y_1
3. \quad t_3 \leftarrow t_3 - x_3
4. \quad t_1 \leftarrow (x_3 + 2x_1)x_3
5. \quad x_2 \leftarrow y_1x_3
6. \quad t_2 \leftarrow (q_3t_0 - t_0^2)t_0
7. \quad t_1 \leftarrow ((x_1 - t_2)y_2 - x_2)t_0
8. \quad \text{return } (x_1, y_1), (t_2, t_1)
\]

\[\text{Algorithm 7} \text{ Extended affine doubling}\]
\[
\text{Input: } P = (x_1, y_1)
\]
\[
\text{Output: } 2P
\]
\[
1. \quad t_0 \leftarrow 3x_1^2 + a, t_1 \leftarrow (2y_1)^{-1}
2. \quad t_4 \leftarrow y_1^2 + t_2 \leftarrow 8x_1t_4
3. \quad t_3 \leftarrow t_3 - t_2\cdot t_2 \leftarrow t_3^2
4. \quad t_1 \leftarrow t_1t_2, x_1 \leftarrow x_1 - t_3
5. \quad t_0 \leftarrow t_0t_4, t_2 \leftarrow 2t_4
6. \quad t_0 \leftarrow (t_0 - t_4)t_1
7. \quad x_1 \leftarrow x_1 + t_1
8. \quad \text{return } (x_1, y_1), (t_1, t_0)
\]
Theorem 1  Let $E(F_p)$ be $y^2 = x^3 + ax + b, b \neq 0 \pmod{p}$, meaning that point $(0, 0)$ is not on $E(F_p)$. $P$ and $Q$ are points on $E(F_p)$. By setting $(0, 0)$ as $O$, the extended addition formula can compute the addition of $P$ and $Q$ correctly if $P \neq Q$ ($P \neq O$, $Q \neq O$), $P + Q = O$, and $O + O$. The extended doubling formula can compute the doubling of $P$ correctly for any point on $E(F_p)$.

Proof 1  Firstly we prove that Algorithm 6 can compute $P + Q = O$ and $O + O = O$ correctly. When computing $P + Q = O$ (for example $P = (x_1, y_1) = (x, y)$ and $Q = (x_2, y_2) = (x, -y)$), the inversion of zero $((x_2 - x_1) = (x - x) = 0)$ has to be computed. As we stated by the extended Euclidean algorithm, by using Fermat’s little theorem, we obtain zero for the inversion of zero. This demonstrates that by our Algorithm 6, we can compute $P + Q = O$.

$$x_3 = 0, \ y_3 = 0.$$  

This implies $P + Q = O \equiv (0, 0)$. Further, we regard $(0, 0)$ as $O$. Subsequently, our variant of affine addition formula computes $P + Q = O$ correctly. Further, it is clear that $O + O = O(0, 0)$ can be computed correctly.

We should emphasize that extracting the factor of $(x_2 - x_1)^{-1}$ does not affect the addition of other points because the factor $(x_2 - x_1)^{-1}$ becomes zero only when computing $P + Q = O$ and $O + O = O$, and in the other situation, extracting the factor of $(x_2 - x_1)^{-1}$ is always safe.

Secondly we prove that Algorithm 7 can compute $2P = O$ correctly where $P$ is the point of order two. When computing $2P = O$, where the two-torsion point $P = (x_1, y_1) = (x, 0)$ is of zero $y$-coordinate, the inversion of zero $2y_1 = 0$ has to be computed. Subsequently, we can compute $2P = (0, 0)$ by our extended affine doubling formula. Further, we regard $(0, 0)$ as $O$, implying that our variant of the affine doubling formula can compute $2P = O$ correctly when the point $(0, 0)$ is not on $E(F_p)$.

Further, extracting the factor of $(2p)^{-1}$ does not affect the doubling of other points. The $y$-coordinate of $P$ becomes zero only when $2P = O$. The variant of the affine doubling formula is exception-free, implying that it can compute the doubling of all points on $E(F_p)$, which does not include the point $(0, 0)$.

Importantly, the original affine addition formulae cannot compute $P + P, P + Q = O, P + O,$ and $2P = O$, while our extended affine addition formulae can compute $P + Q = O$ and $2P = O$ correctly. The Jacobian and Projective addition formulae can compute $P + Q = O$ and $2P = O$ correctly. Thus, both coordinates become “executable coordinates” in our Algorithms 9–10, where extended affine coordinates are “executable coordinates”. This implies that if our algorithms perform well on the extended affine to compute elliptic curve scalar multiplications, our approach can be easily extended to the Jacobian addition formulae or Projective addition formulae.

5. Secure and Efficient RL Elliptic Curve Scalar Multiplication

We improve Algorithm 2 and propose Algorithms 9 and 10, which avoid exceptional inputs such as $P + P, P + O, P - P,$ and $2P = O$ with a two-torsion point $P$. Then, we combine Algorithms 9 and 10 with (extended) affine to secure elliptic curve scalar multiplication algorithms.

We also enhance the efficiency of our method by two-bit scanning using the affine double and quadruple formulae (DQ-formula) [14], which can compute both $2P$ and $4P$ simultaneously with only one inversion computation, denoted by $[2P, 4P] \leftarrow DQ(P)$. Thus, the computational cost of obtaining both $2P$ and $4P$ in affine coordinates is $t([2P, 4P] \leftarrow P) = 8M + 8S + I$. We revise the details of operations in Algorithm 8 to optimize the use of memory. In fact, the necessary memory in the DQ-formula is improved to 10 field elements.

First, we improve Algorithm 2 to obtain the new 2-ary RL algorithm 9 and combine it with two-bit scanning to obtain the new two-bit 2-ary RL algorithm 10. For Algorithm 10, we adjust the length of $|k|$ to be odd by padding “0” in front of input scalar $|k|$. Thus, two-bit scanning can operate well for even or odd length of $|k|$. Both Algorithms 9 and 10 assume that $k \in \mathbb{Z}/N\mathbb{Z}$ is in $k \in [-\frac{N}{2}, \frac{N}{2})$. Actually, $k$ is determined by modulo $N$, thus, this is a natural setting. This technique ensures that our algorithms exclude exceptional points exactly, as shown in Theorem 2. Then, $k$ is represented by $k = (-1)^h \sum_{i=1}^{\ell} k_i 2^i (k_i \in \{0, 1\})$, where $k_i$ is the sign bit and $0 \leq |k| \leq \frac{N}{2}$.

Algorithms 9 and 10 consist of three parts: initialization, main loop, and final correction. Compared with Algorithm 2, we change the initialization of $R[.]$ to avoid the exceptional initialization of $O$ and the exceptional computation $O + P$ in the main loop. The initialization of $R[.]$ causes $R[0] + 2R[1] = P$ to be

Algorithm 8  Affine double and quadruple formulae

Input: $P(x_1, y_1)$
Output: $2P, 4P$

1: $t_0 = x_1^2, t_1 = 2y_1^2, t_2 = t_1^2$
2: $t_1 = 3(t_0 + x_1^2) - t_0 - t_2, t_0 = 3t_0 + a, t_1 = t_0^2$
3: $t_1 = (t_1 - t_0)y_0, t_2 = 2t_1 - t_1 - t_2$
4: $t_2 = 2nt_1, t_3 = t_1, t_3 = t_3 + nt_1$
5: $x_0 = x_1^2 - 2x_1y_0 + (x_1 - x_0)y_0, t_1 = t_1(t_1)$
6: $t_0 = (3x_1^2 + a)t_3, t_3 = t_0 - 2x_1y_1 = (x_1 - x_0)y_0 - y_2$
7: return $(x_1, y_2), (x_1, y_1)$

Algorithm 9  New 2-ary RL algorithm

Input: $P \in E(F_p), k \in [-\frac{N}{2}, \frac{N}{2}), k = (-1)^h \sum_{i=1}^{\ell} k_i 2^i, k_i \in \{0, 1\}$
Output: $kP$
Uses: $A, R[0], R[1]$

Initialization
1: $R[0] \leftarrow -P$
2: $R[1] \leftarrow P$
3: $A \leftarrow 2P$
4: $R[k_0] \leftarrow R[k_3] + A$

Main loop
5: for $i = 1$ to $1 - 1$
do
6: $R[k_0] \leftarrow R[k_0] + A$
7: $A \leftarrow 2A$
8: end for

Final correction
9: $R[k_0] \leftarrow R[k_0] - P$
10: $A \leftarrow -A + R[0] + 2R[1]$
11: $A = (-1)^h \times A$
12: return $A$
Algorithm 10 New two-bit 2-ary RL algorithm

Input: \( P \in E(\mathbb{F}_p), k \in [-\frac{N}{2}, \frac{N}{2}) \), \( k = (-1)^{i_2}k_{2}, k_1 \in \{0,1\} \)

Output: \( kP \)

Uses: \( A, A[1], R[0], R[1] \)

Initialization
1: \( R[0] = -P \)
2: \( R[1] = P \)
3: \( \{A, A[1]\} \leftarrow DQ(P) = (2P, 4P) \)
4: \( {R[k]} \leftarrow R[k_0] + A \)

Main loop
5: for \( i = 1 \) to \( i-1 \) do
6: \( R[k] \leftarrow R[k] + A \)
7: \( \{A, A[1]\} \leftarrow DQ(A[1]) \)
8: \( i = i + 2 \)
9: end for

Final correction
10: \( R[k_0] \leftarrow R[k_0] - P \)
11: \( A \leftarrow -A + R[0] + 2R[1] \)
12: \( A = (-1)^{i} \times A \)
13: return \( A \)

Algorithm 10 is an elliptic curve without two-torsion points. Let \( P \in E(\mathbb{F}_p), P \neq O \) be an elliptic curve point, whose order is \( N \neq 3 \). Then, Algorithms 9 and 10 using (extended) affine can compute \( kP \) correctly for any input \( k \in [-\frac{N}{2}, \frac{N}{2}] \) without introducing conditional statements.

Proof 2 We prove that all three parts exclude the exceptional computations of affine coordinates, which are additions of \( P \) and \( P \) and doubling of \( 2P \) or \( 2P \). The doubling of \( 2P = O \) does not appear in the algorithms because of the assumption that \( E(\mathbb{F}_p) \) is without two-torsion points. Thus, we only focus on exceptional additions.

In the initialization, \( R[0] \) and \( R[1] \) initialized as \( (P_x, -P_y) \) and \( (P_x, P_y) \) are “odd” scalar points such as \( (2t + 1)P, t \in \mathbb{Z} \). A initialized as \( (2P_x, 2P_y) \) is an “even” scalar point such as \( (2t)P, t \in \mathbb{Z} \). It is obvious that \( R[0] \leftarrow -P + 2P \) or \( R[1] \leftarrow P + 2P \) in Step 4 can be computed correctly by the original affine addition formula if \( N \neq 3 \).

In the main loop, it is noteworthy that 1) \( A \neq O \) because of \( E(\mathbb{F}_p) \) without two-torsion points and \( A \) is always updated as an “even” scalar point until \( 2P \) or \( 2P \) is without two-torsion points. Thus, we only focus on exceptional computations. Without introducing conditional statements.

6. Efficiency and Memory Analysis

6.1 Theoretical Analysis

We analyze the computational and memory cost of Algorithms 9 and 10 with (extended) affine and Algorithm 2 with CA formulae, which is shown in Table 2. The memory cost counts the number of \( \mathbb{F}_p \) elements, including the memory used in the addition formulae. As for computational cost, we evaluate all algorithms by estimating the number of modulo multiplication \((M)\), modulo square \((S)\), multiplication with parameters \(a \) and \( b \) (\(ma\) and \(mb\)), addition \((A)\), and inversion \((I)\). The total computational cost of Algorithm 2 with CA formulae is \((l + 1)24M\) if we ignore the computational cost of \(ma\), \(mb\), and \(A\). Assuming the

| Algorithm | Computational cost | Memory |
|-----------|--------------------|--------|
| Alg. 2 + CA \(9\) | \((l + 1)(24M + 6ma + 4mb + 46A)\) | 19 |
| Alg. 9 + (extended) affine | \((6.4l + 16M + (2l + 4)I)\) | 12 |
| Alg. 10 + (extended) affine | \((10l + 23.2)M + (\frac{22l}{2})I\) | 15 |

© 2020 Information Processing Society of Japan
ratio of $S = 0.8M$. Algorithms 9 and 10 with (extended) affine are more efficient than Algorithm 2 with CA formulae if $\frac{1}{7} < 8.8$ and $\frac{1}{8} < 7.9$, respectively. Algorithm 10 is more efficient than Algorithm 9 if $\frac{1}{7} < 7.2$. In summary, when omitting the computational cost of $ma$, $mb$, and $A$, Algorithm 10 is the most efficient if $7.2 < \frac{1}{7} < 7.9$, Algorithm 9 is the most efficient if $\frac{1}{7} < 7.2$, and Algorithm 2 is the most efficient if $\frac{1}{7} > 7.9$. In many cases, such as NIST elliptic curves, we can only omit the computational cost of $ma$ and $A$. Then, Algorithm 10 is the most efficient if $7.2 < \frac{1}{7} < 7.9$, Algorithm 9 is the most efficient if $\frac{1}{7} < 7.2$, and Algorithm 2 is the most efficient if $\frac{1}{7} > 7.9$. Depending on whether ignoring computational costs of $ma$ and $mb$ or not, we summarize the most efficient algorithms depending on the $\frac{1}{7}$ in Table 3.

As for the memory cost, Algorithms 9 and 10 can reduce that of Algorithm 2 with CA formulae by 37% and 21%, respectively.

6.2 Experimental Results

We have implemented Algorithms 9 and 10 with (extended) affine and Algorithm 2 with CA formulae on NIST P-224, P-256, and P-384, which are shown in Table 4. We randomly generate $10^5$ test scalars during the interval of $[-\frac{N}{2}, \frac{N}{2}]$, where $N$ is the order of the point $P$. The experimental platform uses C programming language with GNU MP 6.1.2 and Intel (R) Core(TM) i7-8650U CPU @ 1.90 GHz 2.11 GHz personal computer with 16.0 GB RAM 64-bit; the operating system is Windows 10.

Table 5 shows the average scalar multiplication time of Algorithms 9 and 10 with (extended) affine and Algorithm 2 with CA formulae. Table 5 shows that Algorithms 9 and 10 reduce the computational time of Algorithm 2 by 28.39% and 29.62%, 20.04% and 25.28%, and 6.53% and 14.72%, over NIST P-224, P-256, and P-384, respectively.

As we have already established, the efficiency of our algorithms depends on the ratio $\frac{1}{7}$. Our Algorithm 10 is the most efficient in our experiment, although the ratio $\frac{1}{7}$ in the GNU MP library is approximately between 4 and 7 in Table 6. Function calls and the number of loops may cost time. Algorithm 10 has fewer function calls and loops, which saves time. Consequently, Algorithm 10 with (extended) affine may be the most efficient over all NIST elliptic curves regardless of $\frac{1}{7}$.

7. Conclusion

We have proposed two new secure and compact RL elliptic curve scalar multiplication Algorithms 9 and 10 with (extended) affine coordinates. Our algorithms have generality of $k$ and secure generality and can except exceptional computations of $O + P$, $P - P = O$, and $P + P$. Our extended affine coordinates can compute $P - P = O$ and $2P = O$ by introducing a point $(0,0)$ as $O$ when an elliptic curve $E(\mathbb{F}_p) \ni (0,0)$. From the theoretical point of view, our results can be summarized as follows. When omitting the computational cost of $ma$, $mb$, and $A$, Algorithm 10 with (extended) affine is the most efficient if $7.2 < \frac{1}{7} < 7.9$, Algorithm 9 with (extended) affine is the most efficient if $\frac{1}{7} < 7.2$, and Algorithm 2 with CA formulae is the most efficient if $\frac{1}{7} > 7.9$. In many cases, such as for NIST elliptic curves, we can only omit the computational cost of $ma$ and $A$. In this case, Algorithm 10 with (extended) affine is the most efficient if $7.2 < \frac{1}{7} < 7.9$, Algorithm 9 with (extended) affine is the most efficient if $\frac{1}{7} < 7.2$, and Algorithm 2 with CA formulae is the most efficient if $\frac{1}{7} > 7.9$. Algorithms 9 and 10 with (extended) affine can reduce the memory of Algorithm 2 with CA formulae by 37% and 21%, respectively.

Acknowledgments This work is partially supported by Project for Establishing a Nationwide Practical Education Network for IT Human Resources Development, Education Network for Practical Information Technologies, and Innovation Platform for Society 5.0 at MEXT.

References

[1] Jin, Y. and Miyaji, A.: Secure and Compact Elliptic Curve Cryptosys-
tems, Australasian Conference on Information Security and Privacy, Vol.11547, pp.639–650, Springer (2019).
[2] Afreen, R. and Mehrotra, S.: A review on elliptic curve cryptography for embedded systems, arXiv preprint arXiv:1107.3631 (2011).
[3] Joyce, M.: Highly regular right-to-left algorithms for scalar multipli-
cation, International Workshop on Cryptographic Hardware and Em-
Appendix

A.1 Proof of Generality of $k$ of Joye’s Regular RL m-ary Algorithm

Algorithm 11: Joye’s RL m-ary algorithm [4]

Input: $P \in E(F_p), k = \sum_{i=0}^{n-1} k_i m^i$

Output: $kP$

Uses: $A$ and $R\{1\}, \ldots, R\{m\}$

Initialization
1: for $i = 1$ to $m$
2: $R[i] \leftarrow O$
3: end for
4: $A \leftarrow P$

Main loop
5: for $i = 0$ to $1 - 2d$
6: $R[1 + k_i] \leftarrow R[1 + k_i] + A$
7: $A \leftarrow mA$
8: end for

Aggregation and final correction
9: $A \leftarrow (k_{i-1} - 1)A + \sum_{i=0}^{n-1} (m + i - 2) R[i]$
10: $A \leftarrow A + P$
11: return $A$

Theorem 3: Joye’s regular RL m-ary algorithm. Algorithm 11, can correctly compute $kP$ with any input scalar $k \in [0, \ldots, m - 1]^d$ and $P \in E(F_p)$.

Proof:
Let $k = \sum_{i=0}^{n-1} a_i m^i$ be an m-ary representation of scalar with $a_i \in [0, m - 1]$

Case 1: The MSB of $k$ is not zero, $a_{i-1} \in [1, m - 1]$. Joye’s regular RL m-ary algorithm computing $kP$ correctly is described in Ref. [4].

Case 2: The MSB of $k$ is zero, $a_{i-1} = 0$, and $k \neq 0$. Assuming that the length of “0”的 before the first bit $a_0$ ($a_0 \neq 0$) is $N$, the length of the rest part is $n = 1 - N$. From Algorithm 11, we can see the values will be updated as:

| $R[1]$ | $R[a_i + 1]$ | $A$ |
|-------|--------------|-----|
| $O$   | $O$          | $P$ |

Reading $a_i$









Because of a series of “0”s in front of $a_i$, in the aggregation of Algorithm 11, $(A' + mA' + \cdots + m^{N-1} A') (m - 1) + m^N A' = -A'$ will be computed. When the MSB of $k'$ is not zero, Algorithm 11 will compute $(a_{i-1} - 1) A'_{m}$ for $a_{i-1}$ in Step 9. When the MSB of $k$ is zero, $a_{i-1} = 0$, and $k \neq 0$, Algorithm 11 will compute $(m + a_1 - 1) \frac{A} {m}$ for the first nonzero bit $a_i$ in Step 9. Then $(m + a_1 - 1) \frac{A} {m} - (a_{i-1} - 1) \frac{A} {m} = A'$ when $a_i = a_{i-1}'$ will be added to the correct result because of $a_i$. And finally $-A' + A' = O$ will influence the result because of the first nonzero bit $a_i$ and “0”s in front of $a_i$. This indicates that there’s no influence to the final result when the MSB of scalar $k$ is ‘0’.
Case 3: $k = 0, (P + mP + \cdots + m^{l-2}P)(m - 1) - m^{l-1}P + P = O$. Thus, Joye’s regular RL m-ary algorithm is of generality of $k$.

Yaoan Jin received his bachelor’s degree of computer science from Shanghai Jiao Tong University in 2017 and his master’s degree of electrical and electronic information engineering from Osaka University in 2019. He received the 85th CSEC Outstanding Research Award in 2019. His research work is around Elliptic Curves. He is also interested in secure protocols, FPGA programming, machine learning and try to find out association studies between them.

Atsuko Miyaji received her B.Sc., M.Sc., and Dr.Sci. degrees in mathematics from Osaka University, in 1988, 1990, and 1997 respectively. She joined Panasonic Co., LTD. from 1990 to 1998 and engaged in research and development for secure communication. She was an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) in 1998. She joined the computer science department of the University of California, Davis from 2002 to 2003. She has been a professor at Japan Advanced Institute of Science and Technology (JAIST) since 2007. She has been a professor at Graduate School of Engineering, Osaka University since 2015. Her research interests include the application of number theory into cryptography and information security. She received Young Paper Award of SCIS ’93 in 1993, Notable Invention Award of the Science and Technology Agency in 1997, the IPSJ Sakai Special Researcher Award in 2002, the Standardization Contribution Award in 2003, the AWARD for the contribution to CULTURE of SECURITY in 2007, the Director-General of Industrial Science and Technology Policy and Environment Bureau Award in 2007, DoCoMo Mobile Science Awards in 2008, Advanced Data Mining and Applications (ADMA 2010) Best Paper Award, Prizes for Science and Technology, the Commendation for Science and Technology by the Minister of Education, Culture, Sports, Science and Technology, International Conference on Applications and Technologies in Information Security (ATIS 2016) Best Paper Award, the 16th IEEE Trustcom 2017 Best Paper Award, IEICE milestone certification in 2017, and the 14th Asia Joint Conference on Information Security (AsiaCIS 2019) Best Paper Award. She is a member of the International Association for Cryptologic Research, the Institute of Electrical and Electronics Engineers, the Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and the Mathematical Society of Japan.