Secure and Compact Elliptic Curve LR Scalar Multiplication

Yaoan Jin[1] and Atsuko Miyaji[1,2](✉)

[1] Graduate School of Engineering, Osaka University, Suita, Japan
jin@cy2sec.comm.eng.osaka-u.ac.jp, miyaji@comm.eng.osaka-u.ac.jp
[2] Japan Advanced Institute of Science and Technology, Nomi, Japan

Abstract. Elliptic curve cryptography (ECC) can ensure an equivalent security with much smaller key sizes. Elliptic curve scalar multiplication (ECSM) is a fundamental computation used in ECC. This paper focuses on ECSM resisting simple power attack and safe error attack of side-channel attack specifically. Elliptic curve complete addition (CA) formulae can achieve secure ECSM algorithms but are inefficient from memory and computational cost perspectives. Another secure ECSM, which uses (extended) affine, is more efficient for both memory and computational costs. However, it scans input scalars from right to left. In this paper, our developed scalar multiplication algorithms also use their extended affine, but scan from left to right (LR). We also prove the security of our LR ECSM algorithms and analyze them both theoretically and experimentally. Our new LR ECSM algorithms can reduce the amount of memory by 37.5% and reduce the computational time by more than 40% compared to Joye's regular 2-ary LR algorithm with CA formulae.

Keywords: Elliptic curve scalar multiplication · Side-channel attack

1 Introduction

Elliptic curve cryptography (ECC) can ensure an equivalent security with much smaller key sizes. Hence, ECC has been implemented in secure Internet-of-Things (IoT) devices [1] and various blockchain applications. Elliptic curve scalar multiplication (ECSM) is a fundamental computation used in ECC. It is therefore important to construct a secure and efficient ECSM. Studies on secure and efficient ECSM algorithms can be divided into two categories. The first direction is to find secure and efficient scalar multiplication algorithms [9–11,13,14]. The second direction is to find secure and efficient coordinates with addition formulae [3,6,7,15,17]. This paper concentrates on resisting simple power attack (SPA) and safe error attack (SEA). SPA makes use of "if statements" and SEA makes use of "dummy statements" to reveal significant bits of input scalars. Thus, secure ECSM algorithms should exclude conditional and dummy statements. Elliptic curve CA formulae [7,15,17] can achieve secure ECSM algorithms but are inefficient in terms of memory and computational costs. Another secure ECSM, which uses (extended) affine, is more efficient for both memory and computational costs [8]. However, it scans input scalars from right to left (RL).

In this paper, we propose secure and compact left-to-right (LR) ECSM algorithms based on affine coordinates. We improve Joye's LR 2-ary algorithm to exclude exceptional computations of affine formulae and extended affine formulae [8]. We propose secure scalar multiplication algorithms, Algorithm 7 and Algorithm 8 through 2-bit scanning using the affine double and quadruple algorithm (DQ-algorithm) [12]. Subsequently, along with applying the idea of a Montgomery trick [4,12], we revise three affine combination-addition formulae, which reduce the number of inversion computations to one during all computations. We combine Algorithm 7 and Algorithm 8 with affine combination-addition algorithms and modify Algorithm 8 to Algorithm 9 with our affine combination-addition algorithm (Algorithm 6).

For memory, (Algorithm 7) and (Algorithm 7 with Algorithm 2) use the least amount of memory for ten field elements, reducing that of Joye's LR with CA formulae by 37.5% and that of Joye's RL with CA formulae by 47.37%. For computational cost, we evaluate all ECSMs by estimating the number of modulo multiplication (M), modulo square (S), and inversion (I). Modulo multiplication with parameters a and b (m_a and m_b) and modulo addition (A) are omitted. In many cases, such as the National Institute of Standard and Technology (NIST) elliptic curves, we can only omit m_a and A. Then, our ECSMs of (Algorithm 7 with (extended) affine), (Algorithm 7 with (extended) affine and Algorithm 2), and (Algorithm 9 with (extended) affine and Algorithm 6) can be the most efficient during a larger interval of $\frac{I}{M} \leq \frac{26.8-54/\ell}{1+17/\ell}$ (24.93 when bit length $\ell = 256$) compared to Joye's LR with CA formulae. Experiments also show that our new LR ECSM algorithms can reduce the computational time by more than 40% compared to Joye's LR with CA formulae.

The remainder of this paper is organized as follows. Related studies are provided in Sect. 2. Our proposed algorithms are described in Sect. 3. In Sect. 4, we analyze our Algorithms 7–9 with (extended) affine and affine combination-addition algorithms (Algorithms 2, 3, 4, 5 and 6) from the theoretical and experimental perspectives. Finally, we conclude our work in Sect. 5.

2 Related Work

ECSM algorithms consist of two parts: scalar multiplication algorithms and elliptic curve addition formulae. Thus, related studies on secure and efficient ECSM algorithms can be divided into two categories: scalar multiplication algorithms [9–11,13,14] and elliptic curve coordinates with addition formulae [3,7,15,17]. We briefly introduce related studies in this section.

affine addition formula ($P \neq \pm Q$)

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2$$
$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \tag{1}$$

affine doubling formula ($2P \neq \mathcal{O}$)

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1$$
$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \tag{2}$$

2.1 Addition Formulae and Exceptional Computations

Let $E(\mathbb{F}_p)$ be a Weierstrass elliptic curve over $\mathbb{F}_p$, $E(\mathbb{F}_p) : y^2 = x^3+ax+b$, $(a, b \in \mathbb{F}_p)$. Then affine coordinates compute addition and doubling as Eqs. 1 and 2. A point at infinity cannot be represented clearly by the affine coordinates. Thus, $\mathcal{O} + P$, $P + Q = \mathcal{O}$ and $2P = \mathcal{O}$, which cannot be computed correctly by the affine addition formulae, are so-called exceptional computations of affine addition formulae. In addition, $P + P$ becomes an exceptional computation of the affine addition formula. In summary, $\mathcal{O} + P$, $P + Q = \mathcal{O}$ and $P + P$ are exceptional computations of the affine addition formula and $2P = \mathcal{O}$ is the exceptional computation of the affine doubling formula. Similarly, $\mathcal{O} + P$ and $P + P$ are exceptional computations of Jacobian and projective addition formula. To reduce exceptional computations from affine coordinates, extended affine coordinates assign $(0, 0)$ as the point at infinity for elliptic curves without point $(0, 0)$, such as prime order elliptic curves [8]. Using the extended affine addition formulae, $P + Q = \mathcal{O}$ and $2P = \mathcal{O}$ can be computed as $(0, 0)$, which is exactly the point at infinity. Both $\mathcal{O} + P$ and $P + P$ are still exceptional computations of the extended affine addition formula. We use (extended) affine to indicate the mixed use of the original affine addition and the extended affine addition.

The complete addition (CA) formulae of an elliptic curve [15] can be used to compute the addition of any elliptic curve point pair, and thus, can be employed to secure scalar multiplication algorithms without introducing conditional statements to process exceptional computations. However, CA formulae are inefficient in terms of memory and computational costs. In addition, note that they only work for prime order elliptic curves.

Table 1 summarizes the computational cost of the elliptic curve addition formulae, where M, S, I, and A are the computational costs for one field multiplication, square, inversion, and addition, respectively. Further, m_a and m_b are the computational costs for one field multiplication with parameters a and b, respectively. Assuming that $S = 0.8M$, and ignoring the computational costs of m_a, m_b, and A, the computational cost of one elliptic curve addition (ADD) and one elliptic curve doubling (DBL) using the CA formulae is $24M$ in total. Subsequently, the computational cost of the ADD and DBL using affine addition formulae is more efficient than those using the CA formulae, Jacobian addition formulae, or projective addition formulae when $I < 8.8M$, $I < 8M$ or $I < 9.1M$, respectively. If we employ these addition formulae on NIST elliptic curves, where $a = -3$ and the computational cost m_b cannot be ignored, the computational cost of ADD and DBL using the CA formulae is $26.4M$ in total. The computational cost of ADD and DBL using affine addition formulae is more efficient when $I < 10M$.

2.2 Scalar Multiplication Algorithms

SCA has several attack methods to reveal significant bits of input scalars: a simple power analysis (SPA), which makes use of conditional statements applied during an algorithm depending on the data being processed; a differential power

Table 1. Computational cost of elliptic curve addition formulae

Addition formulae	Conditions	ADD $(P+Q)$	DBL $(2P)$	Memory
CA formulae [15]	$2 \nmid \#E(\mathbb{F}_p)$	$12M + 3m_a + 2m_b + 23A$	$12M + 3m_a + 2m_b + 23A$	15
Affine	–	$2M + S + I$	$2M + 2S + I$	5
Extended affine [8]	$(0,0) \notin E(\mathbb{F}_p)$	$6M + S + I$	$4M + 4S + I$	7
Projective	–	$12M + 2S$	$7M + 5S$	7
Jacobian	–	$11M + 5S$	$M + 8S$	8

analysis (DPA), which uses the correlation between the power consumption and specific key-dependent bits; a timing attack, which uses the relation between the implementation time and the bits of the scalars; and a safe error attack (SEA), which uses dummy statements [2,5]. Therefore, to resist these attacks, we need to eliminate conditional statements in the ECSM for the SPA, the relation between the implementation time and the input scalars for the timing attack, and dummy statements for the SEA. In addition, the power consumption should be changed at each new execution for the DPA. Note that countermeasure to timing attack is taken by padding '0's in front of the input scalars to make certain that almost the same execution time can be easy employed in our algorithms [16]. In this paper, we focus on the SPA and SEA.

Regarding secure ECSM resisting SPA and SEA, three properties, namely, the *generality of k*, *secure generality*, and *executable coordinates* are defined in [8]. In their paper, the authors evaluated secure ECSM focusing on RL scalar multiplication algorithms. We can evaluate Joye's regular 2-ary LR algorithm (Algorithm 1) in the same way as shown in Theorem 1.

Algorithm 1. Joye's regular 2-ary LR algorithm [10]

```
Input: $P \in E(\mathbb{F}_p)$, $k = \sum_{i=0}^{\ell-1} k_i 2^i$
Output: $kP$
Uses: $A$, $R[1]$, $R[2]$
Initialization
1: $R[1] \leftarrow P$, $R[2] \leftarrow 2P$
2: $A \leftarrow (k_{\ell-1} - 1)P$
Main Loop
3: for $i = \ell - 2$ to 0 do
4:     $A \leftarrow 2A + R[1 + k_i]$
5: end for
Final Correction
6: $A \leftarrow A + R[1]$
7: return A
```

Theorem 1. *Joye's regular 2-ary LR algorithm satisfies the* generality of k *and the* secure generality*. Coordinates with CA formulae are its* executable coordinates*. Affine and Jacobian coordinates are not* executable coordinates *of this algorithm.*

New (two-bit) 2-ary RL scalar multiplication algorithms by improving Joye's regular 2-ary RL algorithm to make (extended) affine be *executable coordinates* for it are proposed in [8]. In fact, Joye's regular 2-ary LR algorithm uses two fewer memories than Joye's regular 2-ary RL algorithm. We would like to employ the same idea to improve Joye's regular LR algorithm to use the (extended) affine as *executable coordinates.*

2.3 Inversion-Reduction Combination-Addition Formulae

We can compute any two or more inversions of the field elements using only a single inversion by applying the Montgomery trick. The computational cost of nI becomes $3(n-1)M+I$, which is more efficient when $I > 3M$. Using this method, Eisentrager et al. proposed an affine doubling and addition algorithm (DA-algorithm), computing $2P+Q$ as $P+Q+P$ with $P(x_1,y_1)$, $Q(x_2,y_2)$ using the following formulae [4]:

$$x_3 = \lambda_1^2 - x_1 - x_2, y_3 = \lambda_1(x_1 - x_3) - y_1, \lambda_1 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right) \tag{3}$$

$$x_4 = (\lambda_2 - \lambda_1)(\lambda_2 + \lambda_1) + x_2, y_4 = \lambda_2(x_1 - x_4) - y_1, \lambda_2 = -\lambda_1 - \frac{2y_1}{x_3 - x_1} \tag{4}$$

DA-algorithm computes an inversion of $(x_2 - x_1)^3(x_3 - x_1) = (x_2 - x_1)(y_2 - y_1)^2 - (2x_1 + x_2)(x_2 - x_1)^3$ first and then computes inversions of $(x_2 - x_1)$ and $(x_3 - x_1)$. The result (x_4, y_4) can be computed without computing (x_3, y_3) completely. The computational cost is $9M + 2S + I$.

Le and Nguyen proposed an affine double and quadruple algorithm (DQ-algorithm) [12]. Their algorithm can compute both $2P$ and $4P$ simultaneously with only one inversion computation. The computational cost is $8M + 8S + I$. Its memory use is improved to 11 field elements, as described in [8].

3 Secure and Efficient LR-ECSM Algorithms

Algorithm 1 satisfies the *generality of k* and the *secure generality*, and uses two fewer memories than Joye's regular 2-ary RL algorithm. The affine addition formulae save memory and are efficient depending on the ratio of inversion and multiplication costs but are not the *executive coordinates* of Algorithm 1. In the case of Joye's regular 2-ary RL algorithm, accelerated version with (extended) affine is proposed in [8]. However, the authors failed to apply them to Algorithm 1. With the advantages of Algorithm 1 and affine coordinates, in this section, we describe the improvement of Algorithm 1 to adapt (extended) affine.

First, we revise affine combination-addition formulae using the Montgomery trick, which are used in our new LR ECSM to enhance the efficiency. We then propose our new LR scalar multiplication algorithms with (extended) affine and prove their security.

3.1 Affine Combination-Addition Formulae

In this section, using the Montgomery trick, we revise several affine combination-addition formulae, which are used in our LR ECSM Algorithms 7, 8 and 9. Table 2 shows our affine combination-addition formulae together with the previous formulae. When using the Montgomery trick to reduce the inversion cost, inverses needed to be computed depend on each other. Thus, it is not straightforward to apply the Montgomery trick. First, we improve DQ-algorithm [8,12] to optimize the use of memory as ten field elements in Algorithm 3, which saves one memory from [8].

Algorithm 2. DA-algorithm
Memory: 4+3=7 field elements.
Computational cost: $9M + 2S + I$.

```
Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$
Output: $2P + Q$
1: $y_2 = y_2 - y_1$, $t_0 = y_2^2$, $t_1 = 2x_1 + x_2$
2: $x_2 = x_2 - x_1$, $t_2 = x_2^2$, $t_1 = t_1 t_2$
3: $t_0 = t_0 - t_1$, $t_1 = t_0 x_2$, $t_1 = t_1^{-1}$
4: $t_0 = t_0 t_1 y_2$, $t_1 = -2t_1 x_2 t_2 y_1 - t_0$, $t_2 = t_1 + t_0$
5: $t_0 = t_1 - t_0$, $t_0 = t_0 t_2 + x_2 + x_1$, $x_2 = x_2 + x_1$
6: $x_1 = (x_1 - t_0)t_1 - y_1$, $y_2 = y_2 + y_1$, $y_1 = x_1$, $x_1 = t_0$
7: return $(x_1, y_1)$
```

Algorithm 3. DQ-algorithm
Memory: 6+4=10 field elements.
Computational cost: $8M + 8S + I$.

```
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: $t_1 = 3((t_1 + x_1)^2 - t_0 - t_2)$, $t_0 = 3t_0 + a$, $t_3 = t_0^2$
3: $t_1 = (t_1 - t_3)t_0$, $t_2 = 2t_2$, $t_1 = t_1 - t_2$
4: $t_3 = 2t_1 y_1$, $t_3 = t_3^{-1}$, $t_0 = t_0 t_1 t_3$
5: $x_2 = t_0^2 - 2x_1$, $y_2 = (x_1 - x_2)t_0 - y_1$, $t_3 = t_2 t_3$
6: $t_0 = (3x_2^2 + a)t_3$, $x_3 = t_0^2 - 2x_2$, $y_3 = (x_2 - x_3)t_0 - y_2$
7: return $(x_2, y_2)$, $(x_3, y_3)$
```

Algorithm 4. Double-add
Memory: 4+4=8 field elements.
Computational cost: $9M + 5S + I$.

```
Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$
Output: $2P + Q$
1: $t_0 = (2y_1)^2$, $x_2 = x_2 + 2x_1$, $t_1 = -t_0 x_2$
2: $x_2 = x_2 - 2x_1$, $t_2 = 3x_1^2 + a$, $t_3 = t_2^2$
3: $t_1 = t_1 + t_3$, $t_3 = 2t_1 y_1$, $t_3 = (t_3)^{-1}$
4: $t_2 = t_2 t_3 t_1$, $t_3 = 2t_3 y_1 t_0$, $t_1 = t_2^2 - 2x_1$
5: $t_0 = (x_1 - t_1)t_2 - y_1$, $t_0 = (t_0 - y_2)t_3$
6: $x_1 = t_0^2 - x_2 - t_1$, $y_1 = (x_2 - x_1)t_0 - y_2$
7: return $(x_1, y_1)$
```

Algorithm 5. Two-Continuous Adds
Memory: 6+4=10 field elements.
Computational cost: $9M + 4S + I$.

```
Input: $P = (x_1, y_1)$, $Q = (x_2, y_2)$, $R = (x_3, y_3)$
Output: $P + Q + R$
1: $y_2 = y_2 - y_1$, $t_0 = y_2^2$, $t_1 = x_2 - x_1$
2: $t_2 = t_1^2$, $t_3 = x_1 + x_2 + x_3$, $t_3 = t_0 - t_3 t_2$
3: $t_0 = t_1 t_3$, $t_0 = (t_0)^{-1}$, $y_2 = y_2 t_0 t_3$
4: $t_3 = y_2^2 - x_1 - x_2$, $x_2 = (x_1 - t_3)y_2 - y_1 - y_3$, $t_0 = t_0 t_1 t_2 x_2$, $x_2 = t_0^2 - x_3 - t_3$
5: $y_2 = (x_3 - x_2)t_0 - y_3$
6: return $(x_2, y_2)$
```

DA, the computation of $2P + Q$, is a basic computation formulae in the main loop of Algorithm 1. DA-algorithm is not described in detail, and it is thus unclear how much memory is required [4]. We specify the DA-algorithm to optimize the use of memory as seven field elements in Algorithm 2. DA-algorithm in [4] has exceptional inputs of $P + Q = \mathcal{O}$ and $P = Q$, where P and Q have the same x-coordinate.

If an exceptional input $P+Q=\mathcal{O}$ or $P=Q$ is computed using Algorithm 2, $-P$ is its output. That's DA$(P,Q)\rightarrow -P$, where $P+Q=\mathcal{O}$ or $P=Q$. If we make use of Algorithm 2 twice with exceptional input $P+Q=\mathcal{O}$ (for example $P=(x,y)$ and $Q=-P=(x,-y)$), then Algorithm 2 outputs $P\leftarrow \text{DA}(P,Q)=-P$ first, whereas $2P+Q=P$. Next, input the updated P and the original Q into the algorithm again, then Algorithm 2 outputs $P\leftarrow \text{DA}(-P,Q)=P$. Thus, Algorithm 2 outputs DA(DA(P,Q), Q), where $P+Q=\mathcal{O}$, correctly. This is why we can not directly use Algorithm 2 in our Algorithm 7. In Algorithms 2 and 3, inversions are computed in the same way as in the extended affine formulae [8], which means an inversion of zero is computed as zero.

Algorithm 6. Quadruple-Add
Memory: 4+6=10 field elements.
Computational cost: $18M+14S+I$.

```
Input: $P(x_1,y_1)$, $Q(x_2,y_2)$
Output: $4P+Q$
1: $t_0=x_1^2,\ t_1=2y_1^2,\ t_2=t_1^2,\ t_3=(t_1+x_1)^2-t_0-t_2,\ t_0=3t_0+a,\ t_4=t_0^2,\ t_4=t_4-2t_3$
2: $t_3=t_3-t_4,\ t_3=t_0t_3,\ t_3=t_3-2t_2,\ t_1=2t_1x_2+2t_4,\ t_5=t_3^2,\ t_1=4t_1t_5,\ t_5=4t_2a$
3: $t_4=3t_4^2,\ t_4=t_4+t_5,\ t_4=t_4^2,\ t_4=t_4-t_1,\ t_5=2t_3t_4y_1,\ t_5=(t_5)^{-1},\ t_1=2t_2t_4t_5$
4: $t_5=t_3t_5,\ t_0=t_0t_4t_5,\ t_2=t_2^2,\ t_5=32t_2t_5y_1,\ t_2=t_0^2-2x_1,\ x_1=(x_1-t_2)t_0-y_1$
5: $y_1=3t_2^2+a,\ t_1=t_1y_1,\ y_1=t_1^2-2t_2,\ t_2=(t_2-y_1)t_1-x_1,\ x_1=4x_1^2,\ t_5=t_5x_1$
6: $t_2=t_2-y_2,\ t_2=t_2t_5,\ x_1=t_2^2-x_2-y_1,\ y_1=(x_2-x_1)t_2-y_2$
7: return $(x_1,y_1)$
```

Table 2. Comparison of affine combination-addition algorithms

Affine combination-addition algorithms	Ordinary	Computational cost	Memory
DA-algorithm Algorithm 2 $(2P+Q)$ [4]	$4M+3S+2I$	$9M+2S+I$	7
DQ-algorithm Algorithm 3 $(2P,4P)$ [12]	$4M+4S+2I$	$8M+8S+I$	10
Double-Add Algorithm 4 $(2P+Q)$	$4M+3S+2I$	$9M+5S+I$	8
Two-Continuous Adds Algorithm 5 $(P+Q+R)$	$4M+2S+2I$	$9M+4S+I$	10
Quadruple-Add Algorithm 6 $(4P+Q)$	$6M+5S+3I$	$18M+14S+I$	10

Our double-add algorithm (Algorithm 4) computes $2P$ followed by $2P+Q$ instead of first computing $P+Q$, and then $(P+Q)+P$ in Algorithm 2. Algorithm 4 can correctly compute $2P+Q$ if $2P\neq\mathcal{O}$, $2P\neq Q$, $2P+Q\neq\mathcal{O}$ and $Q\neq\mathcal{O}$. Note that Algorithm 4 can compute $2P+Q$ correctly even when $P+Q=\mathcal{O}$ or $P=Q$, which cannot be computed correctly by Algorithm 2. Algorithm 4 can be used in Algorithm 7 without exceptional inputs. The memory usage is eight field elements and the computational cost is $9M+5S+I$. Next, Algorithm 5 combines two continuous affine additions into a single unit, which can compute $P+Q+R$ correctly if $P+Q\neq\mathcal{O}$, $P+Q+R\neq\mathcal{O}$, $P\neq Q$, $P+Q\neq R$ and $P,Q,R\neq\mathcal{O}$, used in Algorithm 8. The memory usage is ten field elements and the computational cost is $9M+4S+I$. Finally, our affine quadruple and addition algorithm (Algorithm 6) combines the computations in the main loop of our extended new two-bit 2-ary LR algorithm (Algorithm 9), $4P+Q$. In general, to obtain $4P+Q$, we need to compute $P \leftarrow 2P$, $P \leftarrow 2P$, and $P \leftarrow P+Q$, which cost $3I$. Algorithm 6 can compute $4P+Q$ correctly if $4P+Q \neq \mathcal{O}$, $4P \neq Q$ and $P, 2P, 4P, Q \neq \mathcal{O}$. The memory usage is ten field elements and the computational cost is $18M+14S+I$.

3.2 Secure and Efficient LR Scalar Multiplication

We improve Algorithm 1 to a new 2-ary LR algorithm (Algorithm 7) and a new two-bit 2-ary LR algorithm (Algorithm 8). We then combine Algorithms 7 and 8 with affine combination-addition algorithms to reduce the inversion computations.

Algorithm 8 uses two-bit scanning, which is different from Algorithm 7. Algorithm 8 adjusts the length of the input scalars k including the sign bit to be even by padding '0's after the sign bit of input scalars[1]. Therefore, two-bit scanning can operate well for both even and odd lengths of input scalars k.

Algorithm 7. New 2-ary LR

```
Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, sign bit $k_l \in \{0,1\}$
Output: $kP$
Uses: A, and R[0], R[1]
Initialization
 1: $A \leftarrow 2P$
 2: $R[0] \leftarrow -2P$
 3: $R[1] \leftarrow -P$
Main loop
 4: for $i = \ell - 1$ to 1 do
 5:     $A \leftarrow 2A + R[k_i]$
 6: end for
Final correction
 7: $A \leftarrow 2A + R[0]$
 8: $A \leftarrow A + R[k_0]$
 9: $A \leftarrow (-1)^{k_\ell} \times A$
10: return A
```

Algorithm 8. New two-bit 2-ary LR

```
Input: $P \in E(\mathbb{F}_p)$, $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, sign bit $k_\ell \in \{0,1\}$
Output: $kP$
Uses: A and R[0], R[1], R[2], R[3]
Initialization
 1: $R[1] \leftarrow -P$, $\{R[0], R[2]\} \leftarrow DQ(R[1])$
 2: $R[3] \leftarrow R[0]$, $A \leftarrow -R[0]$
Main loop
 3: for $i = \ell - 1$ to 1 do
 4:     $A \leftarrow DQ(A)[1] = 4A$
 5:     $A \leftarrow A + R[k_i + 2] + R[k_{i-1}]$, $i = i - 2$
 6: end for
Final correction
 7: $A \leftarrow 2A + R[0]$
 8: $A \leftarrow A + R[k_0]$
 9: $A \leftarrow (-1)^{k_\ell} \times A$
10: return A
```

Both Algorithms 7 and 8 assume that $k \in \mathbb{Z}/N\mathbb{Z}$ is in $k \in [-\frac{N}{2}, \frac{N}{2}]$, which ensures that our algorithms exclude exceptional computations as shown in Theorem 2. Then, k is represented by $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$ ($k_i \in \{0,1\}$), where $k_\ell \in \{0,1\}$ is the sign bit and $0 \leq |k| \leq \frac{N}{2}$. Algorithms 7 and 8 consist of three parts: initialization, a main loop, and a final correction. Compared with Algorithm 1, we change the initialization of $R[.]$ and A to avoid the exceptional initialization of $A \leftarrow \mathcal{O}$ when $k_{\ell-1} = 1$ and the exceptional computations of $2\mathcal{O}+P$, $2\mathcal{O}+2P$, and $-2P+2P$ in the main loop. Our initializations of $R[.]$ and A cause $4P$, or $3P$, to be added to the final result when $k_0 = 0$, or $k_0 = 1$, respectively. We correct this in Steps 7 and 8 of the final correction of Algorithm 7 and Algorithm 8, and thus, avoid the exceptional computation, $A \leftarrow A + R[1]$, in the original final correction of Algorithm 1. Remark that the extended affine is used only once in Step 8 of Algorithm 7 and Algorithm 8. Actually, the extended

[1] The sign bit is '0' at the beginning of k when k is positive, or '1' at the beginning of k when k is negative.

affine is only necessary for $k = 0$. If $k = 0$ is excluded from the input, then only the original affine can work well.

As for further reduction of inversion computations, Algorithms 2 or 4 can be applied in Steps 5 and 7 of Algorithm 7, respectively. Although Algorithm 2 cannot be directly employed in Algorithm 7 when $k_{\ell-1} = 0$, twice use of of Algorithm 2 to exceptional inputs $A + R[k_i] = \mathcal{O}$ outputs a correct result $A \leftarrow 2P$. Thus, in Algorithm 7, using Algorithm 2, we can make sure it can compute correctly by adjusting the number of '0's from $k_{\ell-1}$ until the first bit '1' on the left to be even, by padding '0's after the sign bit of k. Algorithm 3 is applied in Step 4 of Algorithm 8. Algorithm 5 is applied in Step 5 of Algorithm 8.

Our Algorithms 7 and 8 satisfy the *generality of k* as well as the *secure generality*, and the affine coordinates are *executable coordinates* for them. Algorithms 7 and 8 have no conditional or dummy statements. (Extended) affine and affine combination-addition algorithms can be employed without introducing conditional statements, which will be given in the final paper. Thus, Algorithms 7 and 8 with (extended) affine and affine combination-addition are secure ECSM algorithms.

Algorithm 9. Extended New two-bit 2-ary LR algorithm

```
Input: $P \in E(\mathbb{F}_p)$
    $k \in [-\frac{N}{2}, \frac{N}{2}]$, $k = (-1)^{k_\ell} \sum_{i=0}^{\ell-1} k_i 2^i$, sign bit $k_\ell \in \{0, 1\}$
Output: $kP$
Uses: A and R[0], R[1], R[2], R[3]
Initialization
1: $\{R[2], A\} \leftarrow DQ(P)$, $R[1] \leftarrow -(R[2] + P)$, $R[0] \leftarrow -(R[2] + A)$
2: $R[3] = P \leftarrow -(P + A)$, $R[2] \leftarrow -R[2]$
Main loop
3: for $i = \ell - 1$ to 1 do
4:     $A \leftarrow QA(A, R[2k_i + k_{i-1}])$ (Algorithm 6), $i = i - 2$
5: end for
Final correction
6: $R[1] \leftarrow R[1] - R[2]$, $R[0] \leftarrow R[3] - R[1]$
7: $A \leftarrow 2A + R[0]$, $A \leftarrow A + R[k_0]$, $A \leftarrow (-1)^{k_\ell} \times A$
8: return A
```

Theorem 2. *Let $E(\mathbb{F}_p)$ be an elliptic curve without two-torsion points. Let $P \in E(\mathbb{F}_p)$, $P \neq \mathcal{O}$ be an elliptic curve point, whose order is $N > 4$. Then, Algorithms 7 and 8 using (extended) affine and affine combination-addition formulae can compute kP correctly without exceptional computations for any input $k \in [-\frac{N}{2}, \frac{N}{2}]$.*

Algorithm 7 combined with Algorithm 2 (or Algorithm 4) can reduce the inversions from two-times to one-time in the main loop. By contrast, Algorithm 8 computes two inversions in the main loop. To reduce inversions to one-time in the main loop of Algorithm 8, we propose an extended new two-bit 2-ary LR algorithm (Algorithm 9), where our Quadruple-Add algorithm (Algorithm 6) is used in the main loop of Algorithm 9.

In Algorithm 9, we initialize $R[0] = -6P$, $R[1] = -5P$, $R[2] = -4P$, $R[3] = -3P$, $A = 2P$, which can be computed without exceptional computations if $N > 6$. Because of initialization, the condition of $N > 4$ is changed to $N > 6$. We compute $R[0]$ and $R[1]$ back to $-2P$ and $-P$ in Step 6 of Algorithm 9 to what they were in Algorithm 8 to make sure the remaining part of the final correction of Algorithm 9 can be computed correctly. Algorithm 9 has the *generality of* k and the *secure generality* and avoids all exceptional computations of the affine formulae when $k \in [-\frac{N}{2}, \frac{N}{2}]$, similar to Algorithm 8. The proof of Theorems 2 can be easy extended to Algorithm 9.

4 Efficiency and Memory Analysis

4.1 Theoretical Analysis

Table 3. Computational cost and memory cost analysis

	Computational cost	Memory
Joye's RL + CA [10,15]	$(\ell+1)(24M + 6m_a + 4m_b + 46A)$	19
2-ary RL + (extended) affine [8]	$(6.4\ell + 16)M + (2\ell + 4)I$	12
two-bit 2-ary RL + (extended) affine [8]	$(10\ell + 23.2)M + (\frac{3\ell+9}{2})I$	15
Algorithm 1 + CA [10,15]	$\ell(24M + 6m_a + 4m_b + 46A)$	16
Algorithm 7 + (extended) affine	$(6.4\ell + 10.4)M + (2\ell + 2)I$	10
Algorithm 7 + (extended) affine + Algorithm 2	$(10.6\ell + 10.4)M + (\ell + 2)I$	10
Algorithm 7 + (extended) affine + Algorithm 4	$(13\ell + 10.4)M + (\ell + 2)I$	11
Algorithm 8 + (extended) affine	$(10\ell + 17.6)M + (\frac{3\ell+5}{2})I$	13
Algorithm 8 + (extended) affine + Algorithm 5	$(13.3\ell + 18.5)M + (\ell + 2)I$	13
Algorithm 9 + (extended) affine + Algorithm 6	$(14.6\ell + 27)M + (\frac{\ell+17}{2})I$	17

Table 4. The most efficient algorithm with the conditions of $r = \frac{I}{M}$ ($m_a = m_b = A = 0$, ℓ is bit length of k)

Algorithm	Condition	Memory
Algorithm 7 + (extended) affine	$r < 4.2$	10
Algorithm 7 + (extended) affine + Algorithm 2	$4.2 \le r \le \frac{8+33.2/\ell}{1-13/\ell}$	10
Algorithm 9 + (extended) affine + Algorithm 6	$\frac{8+33.2/\ell}{1-13/\ell} \le r \le \frac{18.8-54/\ell}{1+17/\ell}$	17
Algorithm 1 + CA [10,15]	$r > \frac{18.8-54/\ell}{1+17/\ell}$	16

We analyzed the computational and memory costs of Algorithms 7–9 with (extended) affine and affine combination-addition algorithms in comparison with

Table 5. NIST elliptic curves($y^2 = x^3 - 3x + c$)

P-224	$c =$ 18958286285566608000408668544493926415504680968679321075787234672564
P-256	$c =$ 41058363725152142129326129780047268409114441015993725554835256314039467401291
P-384	$c =$ 27580193559959705877849011840389048093056905856361568521428707301988689241309 860865136260764883745107765439761230575

the Algorithm 1 with CA formulae, Joye's regular 2-ary RL algorithm with CA formulae, and two RL algorithms [8], the results of which are shown in Table 3.

The memory cost considers the number of $\mathbb{F}_p$ elements, including the memory used in the scalar multiplication algorithms. For the computational cost, we evaluated all algorithms by estimating the number of modulo multiplications (M), modulo squaring (S), multiplications with parameters a and b (m_a and m_b), additions (A), and inversions (I). We assume that $S = 0.8M$, and that ℓ is the length of the input scalar k. Let us describe the ratio of inversion cost to the multiplication cost by r, i.e., $I = rM$.

The total computational cost of Algorithm 1 with CA formulae is $24\ell M$, and that of Joye's RL with CA formulae is $(\ell + 1)24M$, if we ignore the computational costs of m_a, m_b, and A. Therefore, Algorithm 1 with CA formulae is more efficient than Joye's regular 2-ary RL algorithm with CA formulae and uses less memory. Both the memory and computational costs of Algorithm 7 with Algorithm 2 are less than those of Algorithm 7 with Algorithm 4. However, as stated earlier, Algorithm 2 can be applied to Algorithm 7 only when the number of '0's between the sign bit and the first bit '1' on the left is even. Packaging all computations of the main loop as a single computation unit reduces the inversion computations, and we can see that Algorithm 9 with (extended) affine and Algorithm 6 has a computational cost of $(14.6\ell + 27)M + (\frac{\ell+17}{2})I$, which is the best when $\frac{8+33.2/\ell}{1-13/\ell} \leq r \leq \frac{18.8-54/\ell}{1+17/\ell}$. Therefore, if the ratio r is approximately 11, then Algorithm 9 with (extended) affine and Algorithm 6 is the most efficient approach. Its memory usage is costly but less than that of Joye's RL with CA formulae. Table 4 shows the most efficient ECSM algorithm with the ratio $r = \frac{I}{M}$. Note that the conditions do not change according to the size of the scalar ℓ. In numerous cases, such as the NIST elliptic curves in Table 5, we can only assume that $m_a = A = 0$. The interval of r where our algorithms are more efficient is larger.

Regarding the memory cost, Algorithm 7 uses the least amount of memory of ten field elements, which reduces that of Algorithm 1 with CA formulae by 37.5%.

4.2 Experimental Analysis

We implemented all algorithms listed in Table 3 on NIST P-224, P-256, and P-384. Table 5 shows their comparison. We randomly generated 2×10^5 test scalars during the interval of $[-\frac{N}{2}, \frac{N}{2}]$, where N is the order of point P used to measure the average scalar multiplication time of the algorithms. The experimental platform uses C programming language with GUN MP 6.1.2, which is a multiple precision arithmetic library, 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. We turn off Intel turbo boost, which is Intel's technique that automatically raises certain of its processors' operating frequency, and thus performance, when demanding tasks are running to make sure our computer works at 2.11 GHz.

Table 6. Average computation time for one scalar multiplication (milliseconds)

	P-224	P-256	P-384	Memory
Joye's RL + CA [10,15]	4.02373	4.593395	7.68237	19
2-ary RL + (extended) affine [8]	2.87742	3.552715	7.733155	12
Two-bit 2-ary RL [8] + (extended) affine	2.56329	3.049545	6.113945	15
Algorithm 1 + CA [10,15]	3.945075	4.591625	7.7481	16
Algorithm 7 + (extended) affine	2.8306	3.804565	7.6338	10
Algorithm 7 + (extended) affine + Algorithm 2	2.15022	2.554765	4.695785	10
Algorithm 7 + (extended) affine + Algorithm 4	2.408305	2.962435	6.042845	11
Algorithm 8 + (extended) affine	2.53023	3.259545	6.23321	13
Algorithm 8 + (extended) affine + Algorithm 5	2.40751	2.698705	5.55319	13
Algorithm 9 + (extended) affine + Algorithm 6	1.904045	2.684335	4.92462	17

Table 7. Time of fundamental computations of GUN MP (milliseconds)

	M	S	I	$\frac{I}{M}$
224 bits	0.00138518	0.00129926	0.00486555	4.08232
256 bits	0.00130389	0.00129878	0.00548586	4.56714
384 bits	0.0014351	0.00141946	0.00766026	6.22689

Table 6 shows the average scalar multiplication time. Table 6 shows that Algorithm 7 with Algorithm 2 is the most efficient for NIST P-256 and P-384, which reduces the computation time of Joye's RL with CA by 46.56% for P-224, 44.38% for P-256, and 38.88% for P-384, and the computation time of Algorithm 1 with CA by 45.5% for P-224, 44.36% for P-256, and 39.39% for P-384, and the computation time of two-bit 2-ary RL [8] by 16.11% for P-224, 16.22% for P-256, and 23.2% for P-384. Algorithm 7 with Algorithm 2 uses the least amount of memory of ten field elements. Algorithm 9 with (extended) affine and Algorithm 6 is the most efficient for NIST P-224.

As we previously discussed, the efficiency of our algorithms depends on the ratio $r = \frac{I}{M}$. Algorithm 7 with (extended) affine and Algorithm 2 is the most efficient when applied to P-256 and P-384 during our experiments. The ratio $r = \frac{I}{M}$ in the GUN MP library is 4.56714 and 6.22689, respectively, as shown in Table 7. These implementation results reflect the theoretic analysis in Table 4.

Algorithm 9 with (extended) affine and Algorithm 6 is the most efficient when applied to P-224, where the ratio $\frac{I}{M}$ is 4.08232. By contrast, Algorithm 7 with (extended) affine is the most efficient, as indicated in Table 4. For the implementation time, both function calls and the number of loops in an algorithm cost time according to the compiler. Algorithm 9 has much fewer function calls and loops than the other algorithms, which may save time.

5 Conclusion

We improved the affine combination-addition formulae of double-add (DA), double-quadruple (DQ), two-adds (TA), and quadruple-add (QA) in terms of the memory or computational cost. We also proposed three new secure LR scalar multiplication Algorithms 7, 8 and 9, and we proved that our new LR ECSM algorithms satisfy the *generality of* k and the *secure generality*; in addition, they can exclude exceptional computations of $\mathcal{O} + P$, $P + Q = \mathcal{O}$, and $P + P$, which means the affine coordinates are *executable coordinates* for them.

We analyzed our LR scalar multiplication algorithms with (extended) affine and affine combination-addition formulae from the theoretical perspective. In many cases, such as with NIST elliptic curves, we can only omit the computational cost of m_a and A. In this case, our algorithms of Algorithm 7 with (extended) affine, Algorithm 7 with (extended) affine and Algorithm 2, and Algorithm 9 with (extended) affine and Algorithm 6 are the most efficient when $\frac{I}{M} \leq \frac{26.8-54/\ell}{1+17/\ell}$ (24.93 at $\ell = 256$) compared to Algorithm 1 with CA formulae.

We also analyzed the algorithms from an experimental perspective. Algorithm 7 with (extended) affine and Algorithm 2 achieves a high efficiency. Algorithm 7 with (extended) affine and Algorithm 2 uses the least memory of ten field elements, which reduces the memory requirements of Algorithm 1 with CA formulae by 37.5%.

Acknowledgements. This work is partially supported by CREST (JPMJCR1404) at Japan Science and Technology Agency, enPiT (Education Network for Practical Information Technologies) at MEXT, and Innovation Platform for Society 5.0 at MEXT.

References

1. Afreen, R., Mehrotra, S.: A review on elliptic curve cryptography for embedded systems. arXiv preprint arXiv:1107.3631 (2011)
2. Ciet, M., Joye, M.: (Virtually) free randomization techniques for elliptic curve cryptography. In: Qing, S., Gollmann, D., Zhou, J. (eds.) ICICS 2003. LNCS, vol. 2836, pp. 348–359. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39927-8_32
3. Cohen, H., Miyaji, A., Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 51–65. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49649-1_6

4. Eisenträger, K., Lauter, K., Montgomery, P.L.: Fast elliptic curve arithmetic and improved weil pairing evaluation. In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 343–354. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36563-X_24
5. Fouque, P.-A., Guilley, S., Murdica, C., Naccache, D.: Safe-errors on SPA protected implementations with the atomicity technique. In: Ryan, P.Y.A., Naccache, D., Quisquater, J.-J. (eds.) The New Codebreakers. LNCS, vol. 9100, pp. 479–493. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49301-4_30
6. Goundar, R.R., Joye, M., Miyaji, A., Rivain, M., Venelli, A.: Scalar multiplication on weierstraß elliptic curves from Co-Z arithmetic. J. Cryptogr. Eng. **1**(2), 161 (2011)
7. Izu, T., Takagi, T.: A fast parallel elliptic curve multiplication resistant against side channel attacks. In: Naccache, D., Paillier, P. (eds.) PKC 2002. LNCS, vol. 2274, pp. 280–296. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45664-3_20
8. Jin, Y., Miyaji, A.: Secure and compact elliptic curve cryptosystems. In: Jang-Jaccard, J., Guo, F. (eds.) ACISP 2019. LNCS, vol. 11547, pp. 639–650. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-21548-4_36
9. Joye, M.: Highly regular right-to-left algorithms for scalar multiplication. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 135–147. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74735-2_10
10. Joye, M.: Highly regular m-ary powering ladrs. In: Jacobson, M.J., Rijmen, V., Safavi-Naini, R. (eds.) SAC 2009. LNCS, vol. 5867, pp. 350–363. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-05445-7_22
11. Joye, M., Yen, S.-M.: The montgomery powering ladder. In: Kaliski, B.S., Koç, K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 291–302. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36400-5_22
12. Le, D.P., Nguyen, B.P.: Fast point quadrupling on elliptic curves. In: Proceedings of the Third Symposium on Information and Communication Technology, pp. 218–222. ACM (2012)
13. Mamiya, H., Miyaji, A., Morimoto, H.: Efficient countermeasures against RPA, DPA, and SPA. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 343–356. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-28632-5_25
14. Miyaji, A., Mo, Y.: How to enhance the security on the least significant bit. In: Pieprzyk, J., Sadeghi, A.-R., Manulis, M. (eds.) CANS 2012. LNCS, vol. 7712, pp. 263–279. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-35404-5_20
15. Renes, J., Costello, C., Batina, L.: Complete addition formulas for prime order elliptic curves. In: Fischlin, M., Coron, J.-S. (eds.) EUROCRYPT 2016. LNCS, vol. 9665, pp. 403–428. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49890-3_16
16. Susella, R., Montrasio, S.: A compact and exception-free ladder for all short weierstrass elliptic curves. In: Lemke-Rust, K., Tunstall, M. (eds.) CARDIS 2016. LNCS, vol. 10146, pp. 156–173. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-54669-8_10
17. Wronski, M.: Faster point scalar multiplication on short weierstrass elliptic curves over FP using twisted hessian curves over FP2. J. Telecommun. Inf. Technol. (2016)