# How to Enhance the Security on the Least Significant Bit

Atsuko Miyaji * and Yiren Mo

Japan Advanced Institute of Science and Technology
miyaji@jaist.ac.jp

**Abstract.** Scalar multiplication, which computes $dP$ for a given point $P$ and a scalar $d$, is the dominant computation part of Elliptic Curve Cryptosystems (ECC). Recently, Side Channel Attacks (SCA) on scalar multiplication have become real threats. This is why secure and efficient scalar multiplication is important for ECC, and many countermeasures have been proposed so far. The Montgomery Ladder and the Regular right-to-left algorithm are the simplest and the most elegant algorithms. However, they are vulnerable to an SCA on the Least Significant Bit (LSB). In this paper, we investigate how to enhance the LSB security without spoiling the original features of simplicity. Our elegant techniques make the previous schemes secure against the SCA on LSB, while maintaining original performances.

Key words: Elliptic Curve Cryptography, Scalar Multiplication, Side Channel Attack

## 1 Introduction

The Elliptic Curve Cryptosystem (ECC), which uses a group of rational points of an elliptic curve over a finite field, was independently proposed by Miller and Koblitz in the mid 1980s. The security of ECC is based on the Elliptic Curve Discrete Logarithm Problem (ECDLP) and the scalar multiplication, which computes $dP$ for a given point $P$ and a scalar $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$, is the dominant computation part of ECC. The simplest scalar multiplication is the so-called binary algorithm. The ECC has been attracting the attention of various applications on small devices because the ECC yield security with a compact memory and little computational cost. Especially, for the use of smart card, the ECC needs to be resistant to side channel attacks on scalar multiplication, such as Simple Power Analysis (SPA)[16], Differential Power Analysis (DPA)[16], Zero-value Point Attack (ZPA)[1], Refined Power Analysis (RPA)[7], Safe-Error Attack (SEA)[25], and etc.; and must still work with a compact memory and little computational cost. To address these two issues, many scalar multiplications have been proposed so far. However, there is still room for improvement from the viewpoint of security, memory amount, and computational cost.

* This study is partly supported by Grant-in-Aid for Exploratory Research, 19650002.

Let us review some previous results on right-to-left algorithms. The regular right-to-left algorithm [12] is secure against both SPA and SEA, while it works without extra computation and just repeats both doubling and additions. This is a simple and elegant algorithm. However, it unfortunately does not work regularly for an arbitrary scalar, and needs a special treatment to force the parity of $d$, that is, $d_0$ to be an odd number. In fact, not only the right-to-left algorithm but also other right-to-left algorithms (Algorithms 1 and 3 in [11]) do not work regularly for an arbitrary scalar. Two other right-to-left algorithms (Algorithms 1' and 2 in [11]) work for an arbitrary $d$, but they are vulnerable to SEA. In a sense, those right-to-left algorithms fail to achieve the security on the Least Significant Bit (LSB), which is called *LSB security* in this paper. The typical special treatments are: computing $(d - d_0 + 1)P$ and subtracting $P$ at the end; or adding an appropriate multiple of ord($P$) to $d$. The former method needs one additional subtraction at the end, and the latter method is useful for only scalars $d \ll \mathrm{ord}(P)$. Those special treatments seem to be rather exaggerated and spoil the simplicity of regular right-to-left algorithm. The problem is just on the LSB security, but is nevertheless unavoidable to execute any $d$.

On the other hand, from the viewpoint of countermeasure to ZPA, a Random-Initial-Point algorithm (RIP), which works regularly in the right-to-left way, was proposed in [10]. It is called IIT-RIP in this paper. RIP in the left-to-right way was proposed in [19], which is called MMM-RIP in this paper. Both algorithms enhance the security by just repeating both doubling and additions, without extra computation. In a sense, both IIT-RIP and MMM-RIP are also elegant and simple. Compared with MMM-RIP resistant to both SEA and ZPA, IIT-RIP, however, is vulnerable to SEA and needs one more register of points than MMM-RIP, although it is secure against ZPA. In order to enhance the security to SEA, error detection steps are introduced in [2, 13]. These, however, need additional steps, and, thus, spoils the original simplicity.

Next, let us review some previous results on left-to-right algorithms. The Montgomery Ladder [23] works regularly in the right-to-left way with only 2 registers of points. However, it is vulnerable to SEA, and leaks LSB (See Alg. 7 in Section 2.5). The signed-digit algorithm proposed in [9] can also work regularly in the left-and-right way with only 2 registers of points in the same way as the Montgomery Ladder. The signed-digit algorithm is secure against SEA, but it is not available for even $d$. There is room for improvement for both the Montgomery Ladder and the signed-digit algorithm from the viewpoint of security and availability.

In this paper, we improve the right-to-left and left-to-right algorithms from the viewpoint of security, memory amount, computational cost, and availability. First, we improve Joye's regular right-to-left algorithm to work for an arbitrary scalar $d$, while maintaining the same memory amount as the original. In this paper, our improved algorithm is called the subtracting-doubling algorithm. Next, we improve the IIT-RIP from the viewpoint of security and memory amount. Our improved algorithm can reduce one register of points from the IIT-RIP, and security is further enhanced. Then, we improve the Montgomery

Ladder to be secure against SEA for an arbitrary scalar $d$, while maintaining the same memory amount as the original algorithm. Finally, we improve the signed-digit algorithm to work for an arbitrary scalar $d$ with only 2 registers of points in the same way as the original.

This paper consists of 6 sections. Section 2 describes some known side channel attacks and the previous right-to-left and left-to-right algorithms. Section 3 presents two new right-to-left algorithms, which improve the IIT-RIP or Joye's regular right-to-left algorithms. Section 4 presents two new left-to-right algorithms, which improve the Montgomery Ladder or the signed-digit algorithm. Section 5 compares our proposed algorithms with the previous algorithms [10, 12, 23, 9]. Section 6 concludes this paper.

## 2 Previous Results

### 2.1 Elliptic curve, coordinate system, and scalar multiplication

We can use several different coordinate systems to represent an elliptic curve. In this work, we assume that an elliptic curve $E$ is defined over $\mathbb{F}_p$ with $p > 3$, and choose the Jacobian coordinate. Then, an elliptic curve is given by $E : y^2 = x^3 + ax + b$ $(a, b \in \mathbb{F}_p)$. The Jacobian coordinate and its variants are described in [5], whose doubling and addition can be slightly improved by changing multiplication to square such as $2Z_1Z_2 = (Z_1 + Z_2)^2 - Z_1^2 - Z_2^2$. The latest addition and doubling formulae are available from [5], and the latest iterated doubling formulae are presented in [22]. The co-$Z$ addition, ZADDU, deals with points having the same $Z$-coordinate [21], where $(R, P) \leftarrow \mathrm{ZADDU}(P, Q)$ is defined as: $R \leftarrow P + Q = (X_3 : Y_3 : Z_3)$ and $P \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : Z_3)$ with $Z_3 = \lambda Z_1$ for input of $P \leftarrow (X_1 : Y_1 : Z)$ and $Q \leftarrow (X_2 : Y_2 : Z)$. The conjugate addition, which outputs $(P+Q, P-Q)$ from $P$ and $Q$ [18], is further improved to ZADDC by combining the co-$Z$ [8], where $(R, S) \leftarrow \mathrm{ZADDC}(P, Q)$ is defined as $R \leftarrow P + Q = (X_3 : Y_3 : Z_3)$ and $S \leftarrow (\overline{X_3} : \overline{Y_3} : Z_3)$ for input of $P \leftarrow (X_1 : Y_1 : Z)$ and $Q \leftarrow (X_2 : Y_2 : Z)$.

Let us summarize the computational cost of the formulae. Here we denote the computational cost of multiplication, square, and inversion over a definition field by $M$, $S$ and $I$. Then, the costs of the EC point addition, doubling, $k$-iterated doublings, and conjugate addition in Jacobian coordinate are $11M + 5S$, $2M + 8S$, $(3k - 1)M + (5k + 3)S$ and $12M + 6S$, respectively. Note that $k$-iterated doublings are used in right-to-left algorithms, and the conjugate addition is used in both right-to-left and left-to-right algorithms. We also give the left-to-right and the right-to-left binary algorithms.

| Addition formula (Jacobian coord.) |
|---|
| $U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$, $S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$ |
| $H = U_2 - U_1$, $I = (2H)^2$, $J = HI$, |
| $R = 2(S_2 - S_1)$, $V = U_1I$ |
| $X_3 = R^2 - J - 2V$, $Y_3 = R(V - X_3) - 2S_1J$, |
| $Z_3 = ((Z_1 + Z_2)^2 - Z_1^2 - Z_2^2)H$ |

| Doubling formula (Jacobian coord.) |
|---|
| $S = 2((X_1 + Y_1^2)^2 - X_1^2 - Y_1^4)$, |
| $M = 3X_1^2 + aZ_1^4$, |
| $X_3 = M^2 - 2S$, |
| $Y_3 = M(S - X_3) - 8Y_1^4$, |
| $Z_3 = (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2$ |

Iterated Doubling Formulae to compute $2^k P$ in Jacobian Coordinate

$Y'_0 = 2Y_0, W_0 = aZ_0^4, T_0 = Y_0'^4, S = ((X_0 + Y_0'^2)^2 - X_0^2 - T_0), M = 3X_0^2 + W_0$
$X_1 = M^2 - 2S, Y'_1 = 2M(S - X_1) - T_0, Z_1 = ((Y'_0 + Z_0)^2 - Y_0'^2 - Z_0^2)/2$
For $i = 1$ to $k - 1$ : {
$W_i = W_{i-1}T_{i-1}, T_i = Y_i'^4, S = ((X_i + Y_i'^2)^2 - X_i^2 - T_i), M = 3X_i^2 + W_i$
$X_{i+1} = M^2 - 2S, Y'_{i+1} = 2M(S - X_{i+1}) - T_i, Z_{i+1} = Y'_i Z_i$
}
$Y_k = Y'_k/2$

Conjugate Addition Formulae in Jacobian Coordinate

$P = (X_1, Y_1, Z_1), Q = (X_2, Y_2, Z_2), P + Q = (X_3, Y_3, Z_3), P - Q = (X_4, Y_4, Z_4)$
$X_3 = A^2 - (4B^3 + 8Z_2^2 X_1 B^2), Y_3 = A(Z_2^2 X_1 B^2 - X_3) - Z_2^3 Y_1 B^3, Z_3 = DB$
$X_4 = C^2 - (4B^2 + 8Z_2^2 X_1 B^2), Y_4 = C(Z_2^2 X_1 B^2 - X_4) - Z_2^3 Y_1 B^3, Z_4 = Z_3$
$A = 2(Z_1^3 Y_2 - Z_2^3 Y_1), B = Z_1^2 X_2 - Z_2^2 X_1, C = -2(Z_1^3 Y_2 + Z_2^3 Y_1), D = (Z_1 + Z_2)^2 - Z_1^2 - Z_2^2$

**Algorithm 1** Left-to-Right Binary Alg.

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i = d_{\ell-1} d_{\ell-2} \dots d_0$
**Output:** $dP$

1: $R[0] \leftarrow O, R[1] \leftarrow P$
2: **for** $i = \ell - 1$ to 0 **do**
3: $R[0] \leftarrow 2R[0]$
4: **if** $d_i = 1$ **then**
5: $R[0] \leftarrow R[0] + R[1]$
6: **end if**
7: **end for**
8: **return** $R[0]$

**Algorithm 2** Right-to-Left Binary Alg.

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i = d_{\ell-1} d_{\ell-2} \dots d_0$
**Output:** $dP$

1: $R[0] \leftarrow O, R[1] \leftarrow P$
2: **for** $i = 0$ to $\ell - 1$ **do**
3: **if** $d_i = 1$ **then**
4: $R[0] \leftarrow R[0] + R[1]$
5: **end if**
6: $R[1] \leftarrow 2R[1]$
7: **end for**
8: **return** $R[0]$

### 2.2 Side Channel Attacks

Side Channel Attacks (SCA) are a type of attack which uses information taken from the physical implementation, such as Timing Analysis Attack [15], Simple Power Analysis (SPA) [16] and Differential Power Analysis (DPA) [16]. These are explained in [4]. Here, we summarize SPA and DPA. SPA observes a suitable side channel, such as the power consumption or electromagnetic emanations, and recovers secret information from the leaked information. In DPA, an attacker not only observes but also statistically analyzes the power consumption of a cryptosystem.

In addition to these attacks, the Doubling Attack, which works only in left-to-right algorithms, is proposed in [24], called DblA in this paper. This is because left-to-right algorithms usually execute in such a way that a return value is doubled and added $P$ when $d_i = 1$, where $d = \sum_{i=0}^{\ell} 2^i d_i$ is represented by $d_{\ell-1} d_{\ell-2} \dots d_0$, and $d_i$ is the current bit. Right-to-left algorithms usually execute in such a way that $2^i P$ is added to a return value when $d_i = 1$. Here, let us explain how DblA works in left-to-right algorithms. DblA computes $dP$ and $d(2P)$. If $d_i = 0$, then the $i^{\text{th}}$-round $R[0]$ in the computation of $dP$ is the same

as the $(i-1)^{\text{th}}$-round $R[0]$ in the computation of $d(2P)$. In the $(i+1)^{\text{th}}$ round of $dP$ and $i^{\text{th}}$ round of $d(2P)$, each round executes a doubling. It is possible for an attacker to check whether result values of two doubling operations are the same.

Safe-Error Attack (SEA) timely induces a fault during the execution of an instruction [25], and, deduces whether a target instruction is dummy or not, because an induced error will be a safe-error when the corresponding operation is dummy. Let us explain SEA by taking the double-and-add always algorithm (Algorithm 3) as an example. Suppose that $R[1]$ in Step 4 for $i = i_0$ ($\ell - 1 \leq i_0 \leq 1$) is attacked. Let $(R[0], R[1], R[2]) = (a, b, c)$ in the beginning of Step $i_0$. In the case of $(d_{i_0}, d_{i_0-1}) = (0, *)$, Algorithm 3 works as Table 1, where N/A means that the value is wrong: $R[1]$ has an error in $i = i_0$, while there is no error in either $R[0]$ or $R[2]$. However, the error in $R[1]$ will disappear in Step 4 for $i = i_0 - 1$ by inputting $R[1] \leftarrow R[0] + R[2]$. This is why the error of $R[1]$ for $i = i_0$ is a safe-error. In the case of $(d_{i_0}, d_{i_0-1}) = (1, *)$, Algorithm 3 works as Table 2: $R[1]$ has an error, where the error in $R[1]$ is copied into $R[0]$ by $R[0] \leftarrow R[1]$ in Step 5 for $i = i_0$; and, finally, both $R[0]$ and $R[1]$ have errors in Step 4 for $i = i_0 - 1$. This is why the error of $R[1]$ for $i = i_0$ is a real-error. Thus, we can detect $d_{i_0} = 0$ or 1 ($\ell - 1 \geq i_0 \geq 1$) by using the fact of whether the output value is correct.

**Table 1.** Safe-Error

| $(i$, Step) | Instruction | Value |
|---|---|---|
| $(i_0, 3)$ | $R[0] \leftarrow 2R[0]$ | $R[0] = 2a$ |
| $(i_0, 4)$ | $R[1] \leftarrow R[0] + R[2]$ | $R[1] = N/A$ |
| $(i_0, 5)$ | $R[0] \leftarrow R[0]$ | $R[0] = 2a$ |
| $(i_0 - 1, 3)$ | $R[0] \leftarrow 2R[0]$ | $R[0] = 4a$ |
| $(i_0 - 1, 4)$ | $R[1] \leftarrow R[0] + R[2]$ | $R[1] = 4a + c$ |
| $(i_0 - 1, 5)$ | $R[*] \leftarrow R[1]$ | $R[0] = 4a$ or $4a + c$ |

**Table 2.** Real-Error

| $(i$, Step) | Instruction | Value |
|---|---|---|
| $(i_0, 3)$ | $R[0] \leftarrow 2R[0]$ | $R[0] = 2a$ |
| $(i_0, 4)$ | $R[1] \leftarrow R[0] + R[2]$ | $R[1] = N/A$ |
| $(i_0, 5)$ | $R[0] \leftarrow R[1]$ | $R[0] = N/A$ |
| $(i_0 - 1, 3)$ | $R[0] \leftarrow 2R[0]$ | $R[0] = N/A$ |
| $(i_0 - 1, 4)$ | $R[1] \leftarrow R[0] + R[2]$ | $R[1] = N/A$ |
| $(i_0 - 1, 5)$ | $R[*] \leftarrow R[1]$ | $R[0] = N/A$ |

### 2.3 Highly Regular Right-to-Left Scalar Multiplication Algorithm

Joye proposed a highly regular powering ladder [12], whose idea is to use a representation of $d - 1$ instead of $d$. The representation of $d - 1$ for the binary expansion of $d = \sum_{i=0}^{\ell-1} d_i 2^i$ ($d_{\ell-1} = 1$) is given as follows: $d - 1 = \sum_{i=0}^{\ell-2}(d_i + 1)2^i$. This follows easily by regarding $-1$ as $\underbrace{\bar{1}11\ldots11}_{l}$. Algorithm 4 shows his regular right-to-left scalar multiplication algorithm.

**Algorithm 3** DBL-and-ADD always alg. (L-R)[6]

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$
**Output:** $dP$

1: $R[0] \leftarrow P, R[2] \leftarrow P$
2: **for** $i = \ell - 1$ to $0$ **do**
3: $\quad R[0] \leftarrow 2R[0]$
4: $\quad R[1] \leftarrow R[0] + R[2]$
5: $\quad R[0] \leftarrow R[d_i]$
6: **end for**
7: **return** $R[0]$

**Algorithm 4** Regular Right-to-Left alg. [12]

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 1)$
**Output:** $dP$

1: $R[1] \leftarrow d_0 P, R[2] \leftarrow P, R[0] \leftarrow R[2]$
2: **for** $i = 1$ to $\ell - 2$ **do**
3: $\quad R[0] \leftarrow 2R[0]$
4: $\quad R[1 + d_i] \leftarrow R[1 + d_i] + R[0]$
5: **end for**
6: $R[0] \leftarrow R[1] + 2R[2]$
7: **return** $R[0]$

We should note that if $d_0 = 0$, this algorithm induces an addition to $O$ in the first $i \geq 1$ with $d_i = 0$: $R[1] \leftarrow O + R[0]$ in Step 4, since $R[1] = O$ until the $i$. Thus, Algorithm 4 itself can securely execute only for an odd $d$. In order to enhance the LSB security of $d$, some *special treatment*, as described in [12, 11], is needed to force the parity of $d$ to 1 such as: computing $(d - d_0 + 1)P$ and subtracting $P$ at the end; or adding an appropriate multiple of ord($P$) to $d$. The former method needs one additional subtraction at the end, and the latter method is useful for only scalars $d \ll \text{ord}(P)$. Those special treatments seem to be rather exaggerated and spoil the simplicity of Algorithm 4. The problem exists only in the LSB, but it is nevertheless unavoidable to execute any $d$. We'll propose a simple method to enhance the LSB security, which does not need any additional computation, and works for any ord($P$).

Here, we investigate the LSB security of previous right-to-left scalar multiplication algorithms in [12, 11]. Not only Algorithms 4, but also other right-to-left scalar multiplication algorithms (Algorithms 1 and 3 in [11]) have an initial value of $O$ in spite of $d_0 = 0$ or 1. As a result, these algorithms also need a *special treatment* in order to work on an arbitrary $d$. Two other right-to-left algorithms (Algorithms 1′ and 2 in [11]) work for both even and odd $d$, since they have no initial value with $O$. They are, however, vulnerable to SEA in the last step: if $k_0 = 1$, then an error induced on $R[b] \leftarrow R[b] - P$ will be a safe error, because a return value is $R[0]$, and the error is in $R[1]$. Therefore, those previous right-to-left algorithms in [12, 11] leak LSB of the scalar, need a special treatment for an arbitrary $d$, or are vulnerable to SEA.

### 2.4 Left-to-Right and Right-to-Left RIP Algorithms

There are several countermeasures against DPA attacks, such as the Randomized Projective coordinate method (RPC)[6], the Randomized Curve method (RC) [14], the Exponent Splitting method (ES)[3] and the Random Initial Point method (RIP)[19, 10]. Both RPC and RC are vulnerable to both the Refined Power Analysis (RPA) and the Zero-value Point Attack (ZPA). ES and RIP are resistant to both RPA and ZPA. There are two algorithms of RIP. Algorithm 5

is the left-to-right RIP [19], called MMM-RIP in this paper, although it is called BRIP in the original paper. Algorithm 6 is the right-to-left RIP algorithm [10], called IIT-RIP, although it is called ADA and RIP in the original paper.

**Algorithm 5** MMM-RIP[19, 20]

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 1)$
**Output:** $dP$

1: $R \leftarrow randompoint()$
2: $R[0] \leftarrow R$; $R[1] \leftarrow -R[0]$
3: $R[2] \leftarrow P - R[0]$
4: **for** $i = \ell - 1$ to 0 **do**
5: $\quad R[0] \leftarrow 2R[0] + R[1 + d_i]$
6: **end for**
7: $R[0] \leftarrow R[0] + R[1]$
8: **return** $R[0]$

**Algorithm 6** IIT-RIP[10]

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 1)$
**Output:** $dP$

1: $R = randompoint()$
2: $R[0] \leftarrow R$; $R[2] \leftarrow P$; $R[3] \leftarrow R[0]$
3: **for** $i = 0$ up to $\ell - 1$ **do**
4: $\quad R[1] \leftarrow R[0] + R[2]$
5: $\quad R[2] \leftarrow 2R[2]$; $R[0] \leftarrow R[d_i]$
6: **end for**
7: $R[0] \leftarrow R[0] - R[3]$
8: **return** $R[0]$

Let us investigate differences between Algorithms 5 and 6 from the viewpoint of security, computational cost and memory amount. As for security, Algorithm 6 is vulnerable to SEA: an error in step 4 will be a safe error when $d_i = 0$. It, however, is secure against SPA, DPA, RPA, ZPA, and DblA described in Section 2.2. On the other hand, Algorithm 5 is secure against SEA, SPA, DPA, RPA, ZPA, and DblA. As for the computational cost, we assume the Jacobian coordinate. Algorithm 6 can use the iterated doubling formulae presented in Section 2.1 in the same way as other right-to-left algorithms, which can reduce the computational cost of each $2^i P$. The computational cost for $\ell$ doublings, $2\ell M + 8\ell S$, is reduced to $(3\ell - 1)M + (5\ell + 3)S$ in total. However, it needs to keep an intermediate point $(X_i, Y_i', Z_i)$ for the next computation, as well as outputs $(X_i, Y_i, Z_i)$ in each round $1 \leqslant i \leqslant \ell - 1$. On the other hand, Algorithm 5 cannot use the iterated doubling formulae but the doubling add algorithm [17], which can compute directly both double and add with a cost of $14M + 9S$. The doubling add algorithm reduces the computational cost of ordinary computations of double and add ($13M + 13S$) by $4S - M$, but increases the memory amount by 2 more registers. So, there is no difference in the computational cost between Algorithms 5 and 6, under the ordinary addition formulae without increasing memory amount. As for the memory amount, Algorithm 5 needs 3-point registers, while Algorithm 6 needs one more register to keep $R$ until Step 8, and thus, it needs 4-point registers in total.

In summary, Algorithm 5 can execute with a smaller memory amount and is secure against SEA, SPA, DPA, RPA, ZPA, and DblA, while Algorithm 6 needs more registers and is not secure against SEA. Section 3 will present an elegant technique to improve Algorithm 6.

### 2.5 Highly Regular Left-to-Right Scalar Multiplication Montgomery Ladder

The Montgomery Ladder [23] is described in Algorithm 7. In order to reduce the computational cost, the co-$Z$ coordinate can be applied in Algorithm 8, where $(R, P) \leftarrow \text{DBLU}(P)$ in Step 1 is defined as: $R \leftarrow 2P = (X_2 : Y_2 : Z_2)$ and $P \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : \lambda)$ with $\lambda = Z_2$ [9]; and $\text{ZACAU}(R[d_i], R[1-d_i])$, that is a combination of ZADDC and ZADDU, and can work in $9M + 7S$ with an extra register of $C = (X_1 - X_2)^2$ in addition to two points $R[d_i] = (X_1, Y_1, Z)$ and $R[1-d_i] = (X_2, Y_2, Z)$.

The Montgomery Ladder can work regularly in the left-and-right way with only 2 registers of points. However, we notice that an operation on $R[1]$ of for-loop for the last round, i.e. $i = 0$, becomes a dummy operation because both $R[0]$ and $R[1]$ are executed in the last round, but only $R[0]$ is returned at Step 6. This is why the Montgomery Ladder is vulnerable to SEA, and leaks LSB. One possible countermeasure is to check the coherency between $R[0]$ and $R[1]$ to detect some fault attack. However, it is rather exaggerated and spoils the simplicity of the Montgomery Ladder. We will present a simple method to enhance the LSB security in Section 4.1.

**Algorithm 7** Montgomery Ladder[23]

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$
**Output:** $dP$
1: $R[0] \leftarrow P; R[1] \leftarrow 2P$
2: **for** $i = \ell - 2$ to 0 **do**
3: $\quad R[1-d_i] \leftarrow R[0] + R[1]$
4: $\quad R[d_i] \leftarrow 2R[d_i]$
5: **end for**
6: **return** $R[0]$

**Algorithm 8** Montgomery Ladder (co-$Z$)[8]

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$
**Output:** $dP$
1: $(R[1], R[0]) \leftarrow \text{DBLU}(P)$
2: **for** $i = \ell - 2$ to 0 **do**
3: $\quad (R[d_i], R[1-d_i]) \leftarrow \text{ZACAU}(R[d_i], R[1-d_i])$
4: **end for**
5: **return** $R[0]$

**Signed-digit Algorithm**

Signed-digit algorithms, both left-to-right and right-to-left, are proposed in [9] by using the fact that any $w$-bit binary expansion $00\cdots01$ is equal to a $w$-bit signed-digit expansion $1\bar{1}\cdots\bar{1}\bar{1}$. Here $\bar{1}$ means $-1$. In fact, any odd binary-expansion number $d = \sum_{i=0}^{\ell-1} d_i 2^i$ $(d_{\ell-1}, d_0 = 1)$ can be written in a non-zero form, called ZSD expansion, as $d = \sum_{i=0}^{\ell-1} \delta_i 2^i$, where $\delta_i = (-1)^{1+d_{i+1}} (0 \leqslant i \leqslant \ell - 2)$ and $\delta_{\ell-1} = 1$. Here, we focus on only the left-to-right algorithm, which is presented in Algorithm 9. Remarkably, the ZSD expansion can be obtained on the fly, as we will see in Algorithm 9. In order to reduce the computational cost, the co-$Z$ coordinate can be applied in Algorithm 10, where $(R, P) \leftarrow \text{TPLU}(P)$ in Step 1 is defined as: $R \leftarrow 3P = (X_3 : Y_3 : Z_3)$ and $P \leftarrow (\lambda^2 X_1 : \lambda^3 Y_1 : \lambda)$ with $\lambda = Z_3$ [9]; and $\text{ZDAU}(R[0], (-1)^{1+d_i} R[1])$ is a direct computation of ZADDU and ZADDC, which can work in $9M + 7S$.

Note that, in the same way as the Montgomery Ladder, the signed-digit algorithm can work regularly in the left-and-right way with only 2 registers of points and is secure against SEA. However, it works for only odd $d$. Section 4.2

will present an elegant method to let the signed-digit algorithm work for any $d$.

**Algorithm 9** Signed-digit Alg. [9]

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i$ $(d_0 = 1)$
**Output:** $dP$
1: $R[0] \leftarrow P; R[1] \leftarrow P$
2: **for** $i = \ell - 1$ to 1 **do**
3: $\quad R[0] \leftarrow 2R[0] + (-1)^{1+d_i} R[1]$
4: **end for**
5: **return** $R[0]$

**Algorithm 10** Signed-digit Alg. (co-Z)[9]

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i$ $(d_0 = 1, d \geq 3)$
**Output:** $dP$
1: $(R[0], R[1]) \leftarrow \mathrm{TPLU}(P)$
2: **for** $i = \ell - 2$ to 1 **do**
3: $\quad (R[0], R[1]) \leftarrow \mathrm{ZDAU}(R[0], (-1)^{1+d_i} R[1])$
4: $\quad R[1] \leftarrow (-1)^{1+d_i} R[1]$
5: **end for**
6: **return** $R[0]$

## 3 Enhance the LSB Security of Right-to-Left Algorithms

First, we improve Algorithm 4 to Algorithm 11, which works for any scalar $d$ and is resistant to SEA and SPA, while maintaining the performance of Algorithm 4. Next, we improve Algorithm 6 from the point of view of security and memory amount, which is presented in Algorithm 12.

### 3.1 Subtract-doubling algorithm

Let us explain Algorithm 1 in detail. To enhance the LSB security, Algorithm 11 transforms an $\ell$-bit binary-expansion $d = \sum_{i=0}^{\ell-1} d_i 2^i$ into an $\ell$-bit $\{\bar{1}, \bar{2}\}$-expansion with MSB equal to $3 = d_{\ell-1}+2$ by changing $d$ to $d+2$ *on the fly*, and, then computes $(d + 2)P - 2P$ by regarding 2 as $2\bar{2}\bar{2}\ldots\bar{2}\bar{2}$ for $\bar{2} = -2$. The idea is an extension of Joye's algorithm that regards $-1$ as $\bar{1}11\ldots11$ for $\bar{1} = -1$. Thus, Algorithm 11 naturally changes $d_0$ to "$-2$" and "$-1$", and repeats subtraction and doubling. This is why Algorithm 11 is called the *subtract-doubling* algorithm. To further enhance the security of $d_1$ (next to LSB), Algorithm 11 treats $d_1$ separately from a for-loop. Theorem 1 proves the correctness of Algorithm 11 and also shows that the final subtraction of $2P$ is executed naturally by setting $R[2] = d_0P - 2P$ in Step 1.

**Theorem 1.** *Algorithm 11 computes $dP$ correctly.*

**Proof:** The initial values of $(R[0], R[1], R[2])$ in Step 1 are: $(R[0], R[1], R[2]) = (2P, -P, d_0P - 2P)$. Thus, the final subtraction of $2P$, that is, $(d+2)P - 2P$ is implemented in the beginning. From the simple discussion, the values of $R[0], R[1], R[2]$ right after the for-loop satisfies the equations: $R[0] = 2^{\ell-1}P$, and $2R[1] + R[2] = \sum_{i=0}^{\ell-2}(d_i - 2)2^iP - 2P$. Thus, $dP$ is correctly returned as follow:

$$R[0] + 2(R[0] + R[1]) + R[2] = (d_{\ell-1} + 2)2^{\ell-1}P + \sum_{i=0}^{\ell-2}(d_i - 2)2^iP - 2P = dP. \blacksquare$$

As for security, Algorithm 11 works in a highly-regular right-to-left way, and executes the same operations in each iteration of for-loop without any dummy operation. This is why Algorithm 11 is resistant to SPA, DblA, and SEA.

### 3.2 Modified IIT-RIP

Algorithm 6 uses a register of $R[3]$ to store a random initial point $R$ which is used only in Steps 2 and 8. Our algorithm 12 can execute without this register (See in Steps 2 and 10 of algorithm 12). Let us explain in detail. Algorithm 12 embeds a random initial point $2R$ into Algorithm 4 elegantly, where Algorithm 4 computes $(d-1)P+P$ by setting $(R[0], R[1], R[2]) = (P, d_0P, P)$ in the beginning, repeating doubling and addition, and finally returning $R[1] + 2R[2]$, which includes the final addition to $P$ in $(d-1)P+P$ implicitly. We apply this idea to compute $((d-1)P+2R)+(P-2R)$ as follows: set $(R[0], R[1], R[2]) = (P, d_0P+2R, P-R)$ in the beginning, repeat doubling and addition, and finally return $R[1] + 2R[2]$, which includes the addition to $P-2R$ in $((d-1)P+2R)+(P-2R)$ implicitly. Furthermore, the register $R[0]$ is well re-used in the initialization, which avoids increasing one more register. By using these elegant ideas, no extra register is needed to store the random initial point. Note that the conjugate addition in Section 2 can be applied to Step 3, which can reduce the computational cost. The correctness of Algorithm 12 will be shown in Theorem 2.

As for security, Algorithm 12 works in a highly-regular right-to-left way, executes the same operations in each iteration of for-loop without any dummy operation, and applies the RIP countermeasure at the same time. This is why Algorithm 12 is resistant to SPA, DblA, SEA, RPA, ZPA and DPA.

**Algorithm 11** Subtract-Doubling Alg.

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 3)$
**Output:** $dP$

1: $R[0] \leftarrow 2P$; $R[1] \leftarrow -P$
2: $R[2] \leftarrow (-1)^{d_0+1} R[d_0]$
3: $R[1 + d_1 \& \overline{d_0}] \leftarrow (-1)^{d_1 \& \overline{d_0}} R[1] - R[0]$
4: $R[0] \leftarrow 2R[0]$
5: $R[2] \leftarrow (-1)^{d_1 \& \overline{d_0}} R[1 + (-1)^{d_1 \& \overline{d_0}}]$
6: **for** $i = 2$ to $\ell - 2$ **do**
7: $\quad R[d_i + 1] \leftarrow R[d_i + 1] - R[0]$
8: $\quad R[0] \leftarrow 2R[0]$
9: **end for**
10: $R[0] \leftarrow R[0] + 2(R[0] + R[1]) + R[2]$
11: **return** $R[0]$

($\overline{d_0}$ means the complement of $d_0$.)

**Algorithm 12** Modified IIT-RIP

**Input:** $P$ and $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 1)$
**Output:** $dP$

1: $R = RandomPoint()$
2: $R[0] \leftarrow R$; $R[2] \leftarrow P$
3: $R[1] \leftarrow R[2] + R[0]$
4: $R[2] \leftarrow R[2] - R[0]$
5: $R[1] \leftarrow R[1] + (-1)^{1+d_0} R[1 - (-1)^{1+d_0}]$
6: $R[0] \leftarrow R[0] + R[2]$
7: **for** $i = 1$ to $\ell - 2$ **do**
8: $\quad R[0] \leftarrow 2R[0]$
9: $\quad R[1 + d_i] \leftarrow R[1 + d_i] + R[0]$
10: **end for**
11: $R[0] \leftarrow R[1] + 2R[2]$
12: **return** $R[0]$

**Theorem 2.** *Algorithm 12 computes dP correctly.*

**Proof:**

Values of $(R[0], R[1], R[2])$ before Step 6 are[1]: $(R[0], R[1], R[2]) = (P, d_0P + 2R, P - R)$. From the simple discussion, the values of $(R[1], R[2])$ right after the

[1] To avoid an addition to $O$, Algorithm 12 does not compute $R[1] = d_0P + 2R$ directly but sets $R[1] = P + 2R$ or $2R$ for an odd or even $d$, respectively.

for-loop are: $R[2] = P - R + \sum_{i=1}^{\ell-2} d_i 2^i P$ and $R[1] = d_0 P + 2R + \sum_{i=1}^{\ell-2} \overline{d_i} 2^i P$, where $\overline{d_i}$ means the complement of $d_i$. Thus, $dP$ is correctly returned as follows:

$$2R[2] + R[1] = \sum_{i=0}^{\ell-2} (d_i + 1)2^i P + P = (d-1)P + P = dP. \blacksquare$$

## 4 Enhance the LSB Security of Left-to-Right Algorithms

First, we improve the Montgomery Ladder (Algorithms 7 and 8) such that it is resistant to SEA. It is called the Modified Montgomery Ladder (Algorithms 13 and 14). We also improve the signed-digit algorithm such that it is available for any scalar. It is called the extended signed-digit algorithm (Algorithms 15 and 16).

### 4.1 Modified Montgomery Ladder

An operation on $R[1]$ in $i = 0$ of the for-loop in Algorithm 7 is a dummy operation, mentioned in Section 2.5. Let us explain steps in $i = 0$ of the for-loop and Step 6 of Algorithm 7 in detail. A returned value $R[0]$ in Step 6 can be represented by using $(r_0, r_1) = (R[0], R[1])$ in $i = 1$ of the for-loop:

$$R[0] = \begin{cases} 2r_0 & \text{if } d_0 = 0, \\ r_0 + r_1 & \text{if } d_0 = 1. \end{cases}$$

We modify steps in $i = 0$ of the for-loop to use both registers by changing to: compute $R[d_0] = 2r_0 + r_1$ and $R[d_0] = R[d_0] - R[1 - d_0]$, and return $R[d_0]$. Then, the returned value is the same as Algorithm 7, seen below:

$$R[d_0] = \begin{cases} 2r_0 & \text{if } d_0 = 0, \\ r_0 + r_1 & \text{if } d_0 = 1. \end{cases}$$

Our Algorithm 13 actually uses both two registers until Step 8. Thus, our algorithm is resistant to SEA. Furthermore, Algorithm 13 executes the same operations in each iteration of the for-loop, and, thus is resistant to SPA in the same way as Algorithm 7. As for the memory amount, it uses registers of 2 points, which is the same as in the case of Algorithm 7.

As for the computational cost, Algorithm 13 has the same for-loop as Algorithm 7. The difference exists only in steps for $i = 0$, where it is in the for-loop in Algorithm 7, while it is out of the for-loop in Algorithm 13. For a further reduction of the computational cost, the co-$Z$ coordinate can be applied in the same way as Algorithm 7, which is described in Algorithm 14. The difference is that ZACAU in $i = 0$ of the for-loop in Algorithm 8 is changed to ZDAU and ZADDU in Steps 5 and 6 in Algorithm 14.

**Algorithm 13** Modified Montgomery Ladder

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$
**Output:** $dP$

1: $R[0] \leftarrow P; R[1] \leftarrow 2P$
2: **for** $i = \ell - 2$ to 1 **do**
3: $\quad R[1 - d_i] \leftarrow R[0] + R[1]$
4: $\quad R[d_i] \leftarrow 2R[d_i]$
5: **end for**
6: $R[d_0] \leftarrow 2R[0] + R[1]$
7: $R[d_0] \leftarrow R[d_0] - R[1 - d_0]$
8: **return** $R[d_0]$

**Algorithm 14** Modified Montgomery Ladder (co-Z)

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i (d > 0)$
**Output:** $dP$

1: $(R[1], R[0]) \leftarrow \mathrm{DBLU}(P)$
2: **for** $i = \ell - 2$ to 1 **do**
3: $\quad (R[d_i], R[1 - d_i]) \leftarrow \mathrm{ZACAU}(R[d_i], R[1 - d_i])$
4: **end for**
5: $(R[d_0], R[1 - d_0]) \leftarrow \mathrm{ZDAU}(R[0], R[1])$
6: $(R[d_0], R[1 - d_0]) \leftarrow \mathrm{ZADDU}(-R[1 - d_0], R[d_0])$
7: **return** $R[d_0]$

### 4.2 Extended Signed-digit Algorithm

Algorithm 9 is only available for an odd scalar, as mentioned in Section 2.5, while Algorithm 7 can work for any $d$ although it reveals LSB. Both algorithms 7 and 9 have an important similarity such that both work with two registers of points. We will change the last steps of Algorithm 9 in the same way as Algorithm 7 to work for any $d$.

Let us compare these two algorithms. Let $(R[0]_i, R[1]_i)$ be values of $(R[0], R[1])$ at the end of the for-loop for $1 < i < \ell - 2$. Then, by using the feature that $R[1] - R[0] = P$ holds in Algorithm 7, the next equations hold.

$$R[1 - d_i]_i = R[0]_{i+1} + R[1]_{i+1} = 2R[0]_{i+1} + P = 2R[1]_{i+1} - P \text{ (Step 3, Alg. 7)}, \quad (1)$$
$$R[d_i]_i = 2R[d_i]_{i+1} = R[1 - d_i]_i + (-1)^{1+d_i} P \text{(Step 4, Alg. 7)}, \quad (2)$$

where Eq. (1) is represented by using $d_{i+1}$ as follows:

$$R[1 - d_i]_i = 2R[1 - d_{i+1}]_{i+1} + (-1)^{1+d_{i+1}} P \text{ (Step 3, Algorithm 7)}. \quad (3)$$

This is easily derived from: $R[1 - d_i]_i = 2R[0]_{i+1} + P = 2R[1 - d_{i+1}]_{i+1} + (-1)^{1+d_{i+1}} P$ if $d_{i+1} = 1$, and $R[1 - d_i]_i = 2R[1]_{i+1} - P = 2R[1 - d_{i+1}]_{i+1} + (-1)^{1+d_{i+1}} P$ if $d_{i+1} = 0$. On the other hand,

$$R[0]_i = 2R[0]_{i+1} + (-1)^{1+d_i} P \qquad \text{(Step 3, Algorithm 9)}. \quad (4)$$

Then, the following theorem holds.

**Theorem 3.** *Let* $(R[0]_i, R[1]_i)$ *be values of* $(R[0], R[1])$ *at the end of the for-loop for* $1 < i < \ell - 2$ *in each Algorithms 7 and 9. Then, for the same scalar d and* $\ell - 2 > i > 1$,

$$R[1 - d_i]_i (Alg.\ 7) = R[0]_{i+1} (Alg.\ 9).$$

**Proof:** The statement follows by induction on $i$. When $i = \ell - 2$,

$$R[1 - d_{\ell-2}]_{\ell-2} = 2R[1 - d_{\ell-1}]_{\ell-1} + (-1)^{1+d_{\ell-1}} P = 2R[0] + P = 3P \text{(Alg. 7)},$$
$$R[0]_{\ell-1} = 2R[0]_{\ell} + (-1)^{1+d_{\ell-1}} P = 2P + P = 3P \text{(Alg. 9)},$$

follows. Assume that $R[1-d_i]_i$(Alg. 7) = $R[0]_{i+1}$(Alg. 9) holds for $i$. Then,

$$R[1-d_{i-1}]_{i-1} = 2R[1-d_i]_i + (-1)^{1+d_i}P(\text{Alg. 7}),$$
$$R[0]_i = 2R[0]_{i+1} + (-1)^{1+d_i}P(\text{Alg. 9}),$$

holds for $i-1$, and, thus statements follows. ∎
A simple example between Algorithms 7 and 9 in Table 3 makes Theorem 3 more clear, where underlined points show the relation.

**Table 3.** Transit of $(R[0], R[1])$ in Alg. 7 and 9 ($d = 45 = (101101)_2$)

| Algorithm | Initial Value | | $i=5$ | 4 | 3 | 2 | 1 | 0 | Return |
|---|---|---|---|---|---|---|---|---|---|
| Alg. 7 | $R[0]$ | $P$ | - | $2P$ | $\underline{5P}$ | $\underline{11P}$ | $22P$ | $45P$ | $45P$ |
| | $R[1]$ | $2P$ | - | $\underline{3P}$ | $6P$ | $12P$ | $\underline{23P}$ | $46P$ | |
| Alg. 9 | $R[0]$ | $P$ | $\underline{3P}$ | $\underline{5P}$ | $\underline{11P}$ | $\underline{23P}$ | $45P$ | - | $45P$ |
| | $R[1]$ | $P$ | $P$ | $P$ | $P$ | $P$ | $P$ | - | |

Our algorithm is presented in Algorithm 15: for-loop is the same as that of Algorithm 9; and Step 6 changes $(R[0], R[1])$ to those in $i = 1$ of for-loop of Algorithm 7; and Steps 7 and 8 are the same as Steps 6 and 7 in Algorithm 13 in order to be secure against SEA on LSB. Tables 4, 5, 6, and 7 describe all patterns in Algorithm 15.

**Algorithm 15** Extended Signed-digit Alg.

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i$ with $d > 1$
**Output:** $dP$

1: $R[1] \leftarrow P; R[0] \leftarrow P$
2: **for** $i = \ell - 1$ to 2 **do**
3: $\quad R[0] \leftarrow 2R[0] + (-1)^{1+d_i}R[1]$
4: **end for**
▷ Finalization
5: $b = \overline{d_1 \oplus d_0}$
6: $R[1] = R[0] + (-1)^{1+d_1}R[1]$
7: $R[b] = 2R[1-d_1] + R[d_1]$
8: $R[b] = R[b] - R[1-b]$
9: **return** $R[b]$

**Algorithm 16** Extended Signed-digit Alg. (co-Z)

**Input:** $P$, $d = \sum_{i=0}^{\ell-1} d_i 2^i$ with $d_0 = 1$ and $d \geq 3$
**Output:** $dP$

1: $(R[0], R[1]) \leftarrow \text{TPLU}(P)$
2: **for** $i = \ell - 2$ to 2 **do**
3: $\quad (R[0], R[1]) \leftarrow \text{ZDAU}(R[0], (-1)^{1+d_i}R[1])$
4: $\quad R[1] \leftarrow (-1)^{1+d_i}R[1]$
5: **end for**
▷ Finalization
6: $b = \overline{d_1 \oplus d_0}$
7: $(R[1], R[0]) \leftarrow \text{ZADDU}(R[0], (-1)^{1+d_1}R[1])$
8: $(R[b], R[1-b]) \leftarrow \text{ZDAU}(R[1-d_1], R[d_1])$
9: $(R[b], R[1-b]) \leftarrow \text{ZADDU}(R[b], -R[1-b])$
10: **return** $R[b]$

Thus, our Algorithm 15 can work for any $d$. As for security, Algorithm 15 executes the same operations in each iteration of the for-loop, and is thus secure against SPA. Algorithm 15 is also secure against SEA without revealing LSB using the same idea as in Algorithm 13. As for the memory amount, it uses registers of 2 points, which is the same as that of Algorithm 9. As for the computational cost, Algorithm 15 has the same for-loop as Algorithm 9. The differences exist only in steps for $i = 1$, where it is in the for-loop in Algorithm 9,

**Table 4.** Transit of $(R[0], R[1])$ in Alg. 15 ($d = 44 = (101100)_2$)

| Initial Value | | For-loop ($5 \geqslant i \geqslant 2$) | | | | Finalization | | | Return |
|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | Step 6 | Step 7 | Step 8 | |
| $R[0]$ | $P$ | $3P$ | $5P$ | $11P$ | $23P$ | $23P$ | $23P$ | $23P$ | |
| $R[1]$ | $P$ | $P$ | $P$ | $P$ | $P$ | $22P$ | $67P$ | $44P$ | $44P$ |

**Table 5.** Transit of $(R[0], R[1])$ in Alg. 15 ($d = 45 = (101101)_2$)

| Initial Value | | For-loop ($5 \geqslant i \geqslant 2$) | | | | Finalization | | | Return |
|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | Step 6 | Step 7 | Step 8 | |
| $R[0]$ | $P$ | $3P$ | $5P$ | $11P$ | $23P$ | $23P$ | $67P$ | $45P$ | $45P$ |
| $R[1]$ | $P$ | $P$ | $P$ | $P$ | $P$ | $22P$ | $22P$ | $22P$ | |

**Table 6.** Transit of $(R[0], R[1])$ in Alg. 15 ($d = 46 = (101110)_2$)

| Initial Value | | For-loop ($5 \geqslant i \geqslant 2$) | | | | Finalization | | | Return |
|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | Step 6 | Step 7 | Step 8 | |
| $R[0]$ | $P$ | $3P$ | $5P$ | $11P$ | $23P$ | $23P$ | $70P$ | $46P$ | $46P$ |
| $R[1]$ | $P$ | $P$ | $P$ | $P$ | $P$ | $24P$ | $24P$ | $24P$ | |

**Table 7.** Transit of $(R[0], R[1])$ in Alg. 15 ($d = 47 = (101111)_2$)

| Initial Value | | For-loop ($5 \geqslant i \geqslant 2$) | | | | Finalization | | | Return |
|---|---|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | Step 6 | Step 7 | Step 8 | |
| $R[0]$ | $P$ | $3P$ | $5P$ | $11P$ | $23P$ | $23P$ | $23P$ | $23P$ | |
| $R[1]$ | $P$ | $P$ | $P$ | $P$ | $P$ | $24P$ | $70P$ | $47P$ | $47P$ |

while it is out of the for-loop in Algorithm 15. For a further reduction of the computational cost, the co-$Z$ coordinate can be applied in the same way as Algorithm 9, which is described in Algorithm 16. The differences are that ZDAU in $i = 1$ of the for-loop in Algorithm 16 is changed to ZADDU, ZDAU and ZADDU in Steps 7 to 9 in Algorithm 16.

## 5 Comparison

Table 8 shows comparisons of security, which omit the security with the co-$Z$ coordinate because they are the same as that without the co-$Z$ coordinate. Table 9 shows comparisons of computational cost and memory amount. Let $\ell$ represent the length of a scalar $d$. The computational cost assumes Jacobian coordinates described in Section 2.1. The memory amount is described in two ways: the first indicates the number of points necessary to implement each algorithm; and the other indicates the precise number of registers necessary to implement, which includes registers of points.

Let us compare the right-to-left algorithms: our Algorithms 11 (resp. 12) with Algorithms 4 (resp. 6) from the viewpoint of security, computational cost, and memory amount. Compared with Algorithm 4, our Algorithm 11 enhances

the LSB security for SPA, while maintaining the performances such as computational cost per bit and memory amount. Algorithm 4 needs a special treatment in order to be secure against SPA for an arbitrary scalar. On the other hand, Algorithm 12 also enhances security for SEA, although Algorithm 6 is vulnerable to SEA. In addition, Algorithm 12 can work with 3 points of $R[0]$, $R[1]$, $R[2]$, and thus, it can reduce memory amount. To reduce the computational cost, all right-to-left algorithms, Algorithms 6, 4, 11, and 12, can use iterated doubling formulae, although this technique increases memory amount.

Let us compare left-to-right algorithms from the viewpoint of security, computational cost, and memory amount. Compared with Algorithm 7, Algorithm 13 enhances the LSB security for SEA, while maintaining the performances such as computational cost per bit and memory amount. Algorithm 7 leaks LSB by SEA. On the other hand, Algorithm 15 enhances the availability of Algorithm 9 that works only for odd $d$. Algorithm 15 is secure against SEA, SPA, and DblA in the same way as Algorithm 9, and also keeps the performances such as computational cost per bit and memory amount.

## 6 Conclusion

We have revisited regular right-to-left and left-to-right algorithms, IIT-RIP (Algorithm 6), Joye's regular right-to-left algorithm (Algorithm 4), the Montgomery Ladder (Algorithm 7), and the signed-digit algorithm (Algorithm 9), and we modified them to Algorithm 12, Algorithm 11, Algorithm 13, and Algorithm 15, respectively. Those modified algorithms enhance each LSB security using elegant techniques while maintaining performances such as memory amount and computational cost per bit.

## References


1. Toru Akishita and Tsuyoshi Takagi. Zero-value point attacks on elliptic curve cryptosystem. In Colin Boyd and Wenbo Mao, editors, *Proceedings of Information Security, 6th International Conference, ISC 2003*, volume 2851 of *Lecture Notes in Computer Science*, pages 218–233. Springer, 2003.
2. Yoo-Jin Baek. Regular $2^w$-ary right-to-left exponentiation algorithm with very efficient DPA and FA countermeasures. *International Journal of Information Security*, 9:363–370, 2010.
3. Mathieu Ciet and Marc Joye. (Virtually) Free randomization techniques for elliptic curve cryptography. In Sihan Qing, Dieter Gollmann, and Jianying Zhou, editors, *Proceedings of Information and Communications Security, 5th International Conference, ICICS 2003*, volume 2836 of *Lecture Notes in Computer Science*, pages 348–359. Springer, 2003.
4. Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography (Discrete Mathematics and Its Applications)*. Chapman and Hall/CRC, July 2005.
5. Henri Cohen, Atsuko Miyaji, and Takatoshi Ono. Efficient elliptic curve exponentiation using mixed coordinates. In Kazuo Ohta and Dingyi Pei, editors, *Proceedings of Advances in Cryptology - ASIACRYPT '98, International Conference on the Theory and*

**Table 8.** Comparisons of security

| Alg. | SEA | SPA | DPA | RPA | ZPA | DblA |
|---|---|---|---|---|---|---|
| *Right-to-left algorithms:* | | | | | | |
| Alg. 4 [12] | S | LW | W | W | W | S |
| Alg. 11 | S | S | W | W | W | S |
| Alg. 6 [10] | W | S | S | S | S | S |
| Alg. 12 | S | S | S | S | S | S |
| *Left-to-right algorithms:* | | | | | | |
| Alg. 7 [23] | LW | S | W | W | W | S |
| Alg. 13 | S | S | W | W | W | S |
| Alg. 9 [9] | S | S | W | W | W | S |
| Alg. 15 | S | S | W | W | W | S |

S: Secure, W: Weak, LW: LSB Weak

**Table 9.** Comparisons of computational cost and memory amount

| Alg. | Computational cost(per bit) | Memory amount(# regs.) | Memory amount(# points.) | Work for $\forall d$ |
|---|---|---|---|---|
| *Right-to-left algorithms:* | | | | |
| Alg. 4 [12] | $13M + 13S$ | 14 | $R[0], R[1], R[2]$ | odd $d$ |
| Alg. 11 | $13M + 13S$ | 14 | $R[0], R[1], R[2]$ | $\forall d$ |
| Alg. 6 [10] | $13M + 13S$ | 17 | $R[0], R[1], R[2], R[3]$ | $\forall d$ |
| Alg. 12 | $13M + 13S$ | 14 | $R[0], R[1], R[2]$ | $\forall d$ |
| *Left-to-right algorithms:* | | | | |
| Alg. 7 [23] | $13M + 13S$ | 11 | $R[0], R[1]$ | $\forall d$ |
| Alg. 8 (co-Z) [8] | $9M + 7S$ | 8 | $R[0], R[1]$ | $\forall d$ |
| Alg. 13 | $13M + 13S$ | 11 | $R[0], R[1]$ | $\forall d$ |
| Alg. 14 (co-Z) | $9M + 7S$ | 8 | $R[0], R[1]$ | $\forall d$ |
| Alg. 9 [9] | $13M + 13S$ | 11 | $R[0], R[1]$ | odd $d$ |
| Alg. 10 (co-Z) [9] | $9M + 7S$ | 8 | $R[0], R[1]$ | odd $d$ |
| Alg. 15 | $13M + 13S$ | 11 | $R[0], R[1]$ | $\forall d$ |
| Alg. 16 (co-Z) | $9M + 7S$ | 8 | $R[0], R[1]$ | $\forall d$ |

*Applications of Cryptology and Information Security*, volume 1514 of *Lecture Notes in Computer Science*, pages 51–65. Springer, 1998.

6. Jean-Sébastien Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In Çetin Kaya Koç and Christof Paar, editors, *Proceedings of Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99*, volume 1717 of *Lecture Notes in Computer Science*, pages 292–302. Springer, 1999.
7. Louis Goubin. A refined power-analysis attack on elliptic curve cryptosystems. In Yvo Desmedt, editor, *Proceedings of Public Key Cryptography - PKC 2003, 6th International Workshop on Theory and Practice in Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 199–210. Springer, 2003.
8. Raveen R. Goundar, Marc Joye, and Atsuko Miyaji. Co-Z addition formulæ and binary ladders on elliptic curves - (extended abstract). In Stefan Mangard and François-Xavier Standaert, editors, *Proceedings of Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop*, volume 6225 of *Lecture Notes in*

*Computer Science*, pages 65–79. Springer, 2010.
9. Raveen R. Goundar, Marc Joye, Atsuko Miyaji, Matthieu Rivain, and Alexandre Venelli. Scalar multiplication on Weierstraß elliptic curves from co-Z arithmetic. *Journal of Cryptographic Engineering*, 1(2):161–176, 2011.
10. Kouichi Itoh, Tetsuya Izu, and Masahiko Takenaka. Efficient countermeasures against power analysis for elliptic curve cryptosystems. In Jean-Jacques Quisquater, Pierre Paradinas, Yves Deswarte, and Anas Abou El Kalam, editors, *Proceedings of Smart Card Research and Advanced Applications VI, IFIP 18th World Computer Congress, TC8/WG8.8 & TC11/WG11.2 Sixth International Conference on Smart Card Research and Advanced Applications (CARDIS)*, pages 99–114. Kluwer, 2004.
11. Marc Joye. Highly regular right-to-left algorithms for scalar multiplication. In Pascal Paillier and Ingrid Verbauwhede, editors, *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop*, volume 4727 of *Lecture Notes in Computer Science*, pages 135–147. Springer, 2007.
12. Marc Joye. Highly regular $m$-ary powering ladders. In Michael J. Jacobson Jr., Vincent Rijmen, and Reihaneh Safavi-Naini, editors, *Selected Areas in Cryptography, 16th Annual International Workshop, SAC 2009*, volume 5867 of *Lecture Notes in Computer Science*, pages 350–363. Springer, 2009.
13. Marc Joye and Mohamed Karroumi. Memory-efficient fault countermeasures. In Emmanuel Prouff, editor, *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011*, volume 7079 of *Lecture Notes in Computer Science*, pages 84–101. Springer, 2011.
14. Marc Joye and Christophe Tymen. Protections against differential analysis for elliptic curve cryptography. In Çetin Kaya Koç, David Naccache, and Christof Paar, editors, *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2001, Third International Workshop*, volume 2162 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 2001.
15. Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Proceedings of Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996.
16. Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Proceedings of Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
17. Patrick Longa. ECC point arithmetic formulae (EPAF). `http://patricklonga.bravehost.com/jacobian.html`.
18. Patrick Longa and Catherine H. Gebotys. Novel precomputation schemes for elliptic curve cryptosystems. In Michel Abdalla, David Pointcheval, Pierre-Alain Fouque, and Damien Vergnaud, editors, *Proceedings of Applied Cryptography and Network Security, 7th International Conference, ACNS 2009*, volume 5536 of *Lecture Notes in Computer Science*, pages 71–88. Springer, 2009.
19. Hideyo Mamiya, Atsuko Miyaji, and Hiroaki Morimoto. Efficient countermeasures against RPA, DPA, and SPA. In Marc Joye and Jean-Jacques Quisquater, editors, *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2004, 6th International Workshop Cambridge*, volume 3156 of *Lecture Notes in Computer Science*, pages 343–356. Springer, 2004.
20. Hideyo Mamiya, Atsuko Miyaji, and Hiroaki Morimoto. Secure elliptic curve exponentiation against RPA, ZRA, DPA, and SPA. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 89-A(8):2207–2215, 2006.

21. Nicolas Meloni. New point addition formulae for ECC applications. In Claude Carlet and Berk Sunar, editors, *Proceedings of Arithmetic of Finite Fields, First International Workshop,WAIFI 2007*, volume 4547 of *Lecture Notes in Computer Science*, pages 189–201. Springer, 2007.
22. Atsuko Miyaji. Generalized scalar multiplication secure against SPA, DPA, and RPA. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, 91-A(10):2833–2842, 2008.
23. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987.
24. Katsuyuki Okeya and Kouichi Sakurai. On insecurity of the side channel attack countermeasure using addition-subtraction chains under distinguishability between addition and doubling. In Lynn Margaret Batten and Jennifer Seberry, editors, *Proceedings of Information Security and Privacy, 7th Australian Conference, ACISP 2002*, volume 2384 of *Lecture Notes in Computer Science*, pages 420–435. Springer, 2002.
25. Sung-Ming Yen and Marc Joye. Checking before output may not be enough against fault-based cryptanalysis. *IEEE Transactions on Computers*, 49(9):967–970, 2000.