Generalized MMM-algorithm Secure against SPA, DPA, and RPA

Atsuko Miyaji *

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

Abstract. In the execution on a smart card, elliptic curve cryptosystems have to be secure against side channel attacks such as the simple power analysis (SPA), the differential power analysis (DPA), and the refined power analysis (RPA), and so on. MMM-algorithm proposed by Mamiya, Miyaji, and Morimoto is a scalar multiplication algorithm secure against SPA, DPA, and RPA, which can decrease the computational complexity by increasing the size of a pre-computed table. However, it provides only 4 different cases of pre-computed tables. From the practical point of view, a wider range of time-memory tradeoffs is usually desired. This paper generalizes MMM-algorithm to improve the flexibility of tables as well as the computational complexity. Our improved algorithm is secure, efficient and flexible for the storage size.

keywords elliptic curve, DPA, RPA, SPA

1 Introduction

Elliptic Curve Cryptosystems: The elliptic curve cryptosystem (ECC) chosen appropriately can offer efficient public key cryptosystems [1]. Thus, elliptic curve cryptosystems have been desired in various application such as a smart card, whose memory storage and CPU power are very limited. The efficiency of elliptic curve cryptosystems depends on the implementation of scalar multiplication kP for a secret key k and an elliptic-curve point P.

Side Channel Attacks on ECC: Side channel attacks monitor power consumption and even exploit the leakage information related to power consumption to reveal bits of a secret key k although k is hidden inside a smart card [1]. There are two types of power analysis, the simple power analysis (SPA) and the differential power analysis (DPA). SPA makes use of such an instruction performed during a scalar multiplication that depends on the data being processed. DPA uses correlation between power consumption and specific key-dependent bits. The refined power analysis (RPA) over ECC is one of DPA, which exploits a special point with a zero value such as $(0, y)$ or $(x, 0)$ and reveals a secret key

* This study is partly supported by Grant-in-Aid for Scientific Research (B), 17300002. and Yazaki Memorial Foundation for Science and Technology.

[9]. RPA is also called a Goubin-type attack. Not all elliptic curves are vulnerable against RPA, but some curves in [18] are vulnerable against these attacks. There exist countermeasures against SPA, DPA, and RPA in [3, 16, 14]. This paper revisits the algorithm proposed by Mamiya, Miyaji, and Morimoto [14], which is called MMM-algorithm[1], here.

Overview of MMM-algorithm: MMM-algorithm uses a random initial point (RIP) R, computes $kP + R$ by dividing a scalar k into $h \times 1$ blocks with a table of pre-computed points based on the blocks, subtracts R from a result, and, finally, gets kP, where $kP + R$ is computed from left to right without any branch instruction dependent on the data being processed. A random initial point at each execution of kP makes it impossible for an attacker to control a point P as he needs since any point or register used in addition formulae is different at each execution. Thus, MMM-algorithm is not only secure against SPA, DPA, and RPA but also efficient scalar multiplication with a precomputed table. However, it provides only 4 available tables of 9, 15, 27, or 51 field elements[2]. Note that MMM-algorithm with a table of 51 field elements is the most efficient in a 160-bit elliptic curve even if more memory space is allowed to use. From the practical point of view, the memory space allowed to use or the time complexity required for cryptographic functions depends on each individual application. Thus, in some application, MMM-algorithm might not be the best.

Our Contribution: In this paper, we generalize MMM-algorithm by dividing a scalar k into $h \times v$ blocks and optimize the computation method of $kP + R$ to improve flexibility of tables as well as computational complexity, while being secure against SPA, DPA, and RPA. It is called Generalized MMM-algorithm in this paper, GMMM-algorithm in short. We also analyze the computational complexity of GMMM-algorithm theoretically. Furthermore, we explore each best coordinate between affine, (modified) Jacobian, mixed coordinate, etc [6] for each division $h \times v$ of GMMM-algorithm according as the ratio of I/M, where I/M represents the ratio of complexity of modular inversion against modular multiplication. As a result, even in the same division as MMM-algorithm, our optimization on coordinates can reduce the computational and memory complexity since such optimization was not investigated in MMM-algorithm [14]. In facts, GMMM-algorithm with a table of 7 or 38 field elements can reduce the computational complexity of MMM-algorithm with a table of 9 or 51 field elements by 19% or 13.2% over for the range of I/M between 4 and 11, respectively; and GMMM-algorithm with a table of only 19 field elements can work faster than MMM-algorithm with a table of 51 field elements under the above range of I/M. Thus, GMMM-algorithm is significantly efficient and flexible even when the storage available is very small or rather large.

This paper is organized as follows. Section 2 summarizes the known facts on elliptic curves and also reviews MMM-algorithm. Section 3 presents our GMMM-algorithm and analyzes the computational complexity theoretically. Section 4 compares our results with the previous results.

[1] In their paper, MMM-algorithm is called BRIP or EBRIP.

[2] More precisely, it uses 3, 5, 9, 17 elliptic-curve points in Jacobian coordinates.

2 Preliminaries

This section summarizes some facts of elliptic curves such as coordinate systems and side channel attacks against elliptic curves, which refers to [6, 1].

2.1 Elliptic Curve

Let $\mathbb{F}_p$ be a finite field, where $p > 3$ is a prime. The Weierstrass form of an elliptic curve over $\mathbb{F}_p$ in affine coordinates is described as

$$E/\mathbb{F}_p : y^2 = x^3 + ax + b \quad (a, b \in \mathbb{F}_p, 4a^3 + 27b^2 \neq 0).$$

The set of all points $P = (x, y) \in \mathbb{F}_p \times \mathbb{F}_p$ satisfying E with the point at infinity $\mathcal{O}$, denoted by $E(\mathbb{F}_p)$, forms an abelian group. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on $E(\mathbb{F}_p)$ and $P_3 = P_1 + P_2 = (x_3, y_3)$ be the sum. Then the addition formula Add (resp. doubling Dbl) in affine coordinate can be described by three modules of $\mathsf{Add_p}(P_1, P_2)$, $\mathsf{Add_I}(\alpha)$, and $\mathsf{Add_{NI}}(P_1, P_2, \lambda)$ (resp. $\mathsf{Dbl_p}(P_1)$, $\mathsf{Dbl_I}(\alpha)$ and $\mathsf{Dbl_{NI}}(P_1, \lambda)$) as in [15]. Each module means preparation for 1 inversion, computation of 1 inversion and computation without inversion, respectively. Then the addition formulae are given as follows.

$\mathsf{Add}(P_1, P_2)$ $(P_1 \neq \pm P_2)$		$\mathsf{Dbl}(P_1)$	
$\mathsf{Add_p}(P_1, P_2)$:	$\alpha = x_2 - x_1$	$\mathsf{Dbl_p}(P_1)$:	$\alpha = 2y_1$
$\mathsf{Add_I}(\alpha)$:	$\lambda = \frac{1}{\alpha}$	$\mathsf{Dbl_I}(\alpha)$:	$\lambda = \frac{1}{\alpha}$
$\mathsf{Add_{NI}}(P_1, P_2, \lambda)$:	$\gamma = (y_2 - y_1)\lambda$	$\mathsf{Dbl_{NI}}(P_1, \lambda)$:	$\gamma = (3x_1^2 + a)\lambda$
	$x_3 = \gamma^2 - x_1 - x_2$		$x_3 = \gamma^2 - 2x_1$
	$y_3 = \gamma(x_1 - x_3) - y_1$		$y_3 = \gamma(x_1 - x_3) - y_1$

Let us denote the computational complexity of an addition (resp. a doubling) by $t(\mathcal{A}+\mathcal{A})$ (resp. $t(2\mathcal{A})$) in affine coordinate and multiplication (resp. inversion, resp. squaring) in $\mathbb{F}_p$ by M (resp. I, resp. S), where $\mathcal{A}$ means affine coordinates. For simplicity, it is usual to neglect addition, subtraction, and multiplication by small constant in $\mathbb{F}_p$ to discuss the computation amount. Then we see that $t(\mathcal{A} + \mathcal{A}) = I + 2M + S$ and $t(2\mathcal{A}) = I + 2M + 2S$. For the sake of convenience, let us denote the computational complexity of $\mathsf{Add_p}$ and $\mathsf{Add_{NI}}$ (resp. $\mathsf{Dbl_p}$ and $\mathsf{Dbl_{NI}}$) by $t(A + A)_{nI}$ (resp. $t(2A)_{nI}$), which represents the total computational complexity of an addition (resp. doubling) without inversion.

Both addition and doubling formulae in affine coordinate need one inversion over $\mathbb{F}_p$, which is more expensive than multiplication over $\mathbb{F}_p$. Affine coordinate is transformed into Jacobian coordinate, where the inversion is free. We set $x = X/Z^2$ and $y = Y/Z^3$, giving the equation

$$E_{\mathcal{J}} : Y^2 = X^3 + aXZ^4 + bZ^6.$$

Then, two points (X, Y, Z) and (r^2X, r^3Y, rZ) for some $r \in \mathbb{F}_p^*$ are recognized as the same point. The point at infinity is transformed to $(1, 1, 0)$. The doubling and addition formulae in Jacobian coordinate are represented in [6]. The computation time of addition (resp. doubling) in Jacobian coordinate are $t(\mathcal{J}+\mathcal{J}) = 12M+4S$

(resp. $t(2\mathcal{J}) = 4M + 6S$), where $\mathcal{J}$ means Jacobian coordinate. Regarding the iterated doubling, that is the computation of $2^w P$, the iterated doubling formula in Jacobian coordinate [10] can work efficiently with $t(2^w\mathcal{J}) = 4wM + (4w+2)S$.

In addition to affine and Jacobian coordinates, there exists their combination coordinate, called mixed coordinate [6]. In the case of mixed coordinate, let us denote by $t(\mathcal{C}^1 + \mathcal{C}^2 = \mathcal{C}^3)$ the time for addition of points in coordinates $\mathcal{C}^1$ and $\mathcal{C}^2$ giving a result in coordinates $\mathcal{C}^3$, and by $t(2\mathcal{C}^1 = \mathcal{C}^2)$ the time for doubling a point in coordinates $\mathcal{C}^1$ giving a result in coordinates $\mathcal{C}^2$. Their performance is summarized in Table 2.1.

Table 2.1. Computational complexity of addition and doubling of elliptic curve

	computational complexity		computational complexity
$t(\mathcal{A}+\mathcal{A}=\mathcal{J})$	$4M + 2S$	$t(2\mathcal{A}=\mathcal{J})$	$2M + 4S$
$t(\mathcal{A}+\mathcal{J}=\mathcal{J})$	$8M + 3S$	$t(2\mathcal{J})$	$4M + 6S$
$t(\mathcal{A}+\mathcal{A})$	$2M + S + I$	$t(2\mathcal{A})$	$2M + 2S + I$
$t(\mathcal{J}+\mathcal{J})$	$12M + 4S$	$t(2^w\mathcal{J})$	$4wM + (4w+2)S$
$t(\mathcal{J}+\mathcal{J}=\mathcal{A})$	$15M + 5S + I$	$t(\mathcal{J}\to\mathcal{A})^\dagger$	$3M + S + I$

† : the computational complexity of transformation from Jacobian to affine coordinate. The computational complexity without inversion is denoted by $t(\mathcal{J}\to\mathcal{A})_{nI}$.

Let us discuss the difference between these coordinates. Regarding additions, we could roughly estimate that $t(\mathcal{A}+\mathcal{A}=\mathcal{J}) < t(\mathcal{A}+\mathcal{J}=\mathcal{J}) \leq t(\mathcal{A}+\mathcal{A}) \leq t(\mathcal{J}+\mathcal{J})$. This means an addition in mixed coordinate is considerably fast but that of Jacobian coordinate is rather slow. Therefore, an addition of Jacobian coordinate had better be avoided. Regarding doublings, we could roughly estimate that $t(2\mathcal{A} = \mathcal{J}) < t(2\mathcal{J}) \leq t(2\mathcal{A})$. This means a doubling in Jacobian coordinate is considerably fast but that of affine coordinate is rather slow, which had better be avoided.

The major problem in affine coordinate is that it requires 1 inversion when it is executed. However, an addition of affine coordinate should be revisited if we consider the above fact of $t(\mathcal{A}+\mathcal{A}) \leq t(\mathcal{J}+\mathcal{J})$. Furthermore, if several additions or doublings are executed in parallel, then we can make use of the following Montgomery's trick [17] to reduce the total number of inversions.

Algorithm 1 (Minv[n]) `Montgomery's trick`
`Input:` $\alpha_0, \cdots, \alpha_{n-1}, p$
`Output:` $\alpha_0^{-1} \bmod p, \cdots, \alpha_{n-1}^{-1} \bmod p$
1. $\lambda_0 = \alpha_0$
2. `For` $i = 1$ `to` $n-1$: $\lambda_i = \lambda_{i-1}\alpha_i \bmod p$.
3. $I = \lambda_{n-1}^{-1} \bmod p$
4. `For` $i = n-1$ `to` 0: $\lambda_i = I\lambda_{i-1} \bmod p$. $I = I\alpha_i \bmod p$
5. `Output` $\{\lambda_0, \cdots, \lambda_{n-1}\}$

The Montgomery's trick $\mathsf{Minv}[n]$ works with $3(n-1)$ multiplications and 1 inversion to compute n inversions, whose computation time is denoted by $t(\mathsf{Minv}[n]) = 3(n-1)M + I$. Therefore, if several additions or doublings in affine coordinates are executed in parallel, then the total number of inversions can be reduced to 1 by executing $\mathsf{Add}_{\mathsf{p}}(P_1, P_2)$ and $\mathsf{Dbl}_{\mathsf{p}}(P_1)$ in parallel, applying the Montgomery's trick to execute $\mathsf{Add}_{\mathsf{I}}(\alpha)$ and $\mathsf{Dbl}_{\mathsf{I}}(\alpha)$ simultaneously, and finally executing $\mathsf{Add}_{\mathsf{NI}}(P_1, P_2, \lambda)$ and $\mathsf{Dbl}_{\mathsf{NI}}(P_1, \lambda)$ in parallel.

2.2 Power analysis

There are two types of power analysis, SPA and DPA, which are described in [1]. RPA is one of DPA, which uses characteristic of some elliptic curve to have a special point [9].

Simple Power Analysis: SPA makes use of such an instruction performed during a scalar multiplication that depends on the data being processed. In order to be resistant to SPA, any branch instruction of scalar multiplication should be eliminated. There are mainly two types of countermeasures: the fixed procedure method and the indistinguishable method. The fixed procedure method deletes any branch instruction conditioned by a secret scalar k such as the add-and-double-always algorithm. The indistinguishable method conceals all branch instructions of scalar multiplication algorithm by using indistinguishable addition and doubling operations, in which dummy operations are inserted.

Differential Power Analysis: DPA uses correlation between power consumption and specific key-dependent bits. In order to be resistant to DPA, power consumption should be changed at each new execution. There are mainly 3 types of countermeasures, the randomized-projective-coordinate method (RPC), the randomized curve method (RC), and the exponent splitting (ES) [3]. RPC uses Jacobian or Projective coordinate to randomize a point $P = (x, y)$ into (r^2x, r^3y, r) or (rx, ry, r) for a random number $r \in \mathbb{F}_p^*$, respectively. RC maps an elliptic curve into an isomorphic elliptic curve by using an isomorphism map of (x, y) to (c^2x, c^3y) for $c \in \mathbb{F}_p^*$. ES splits a scalar and computes $kP = rP + (k-r)P$ for a random integer r.

Refined Power Analysis: RPA reveals a secret key k by using a special elliptic-curve point with a zero value, which is defined as $(x, 0)$ or $(0, y)$. These special points of $(x, 0)$ and $(0, y)$ can not be randomized by RPC or RC since they still have a zero value such as $(r^2x, 0, r)$ (resp. $(rx, 0, r)$) and $(0, r^3y, r)$ (resp. $(0, ry, r)$) in Jacobian (resp. Projective) coordinate after conversion. ES can resist RPA because an attacker cannot handle an elliptic curve point in such a way that a special point with zero value can appear during an execution.

2.3 A review of Mamiya-Miyaji-Morimoto-Algorithm

We briefly review MMM-algorithm [14]. Assume that the size of the underlying filed and the scalar k are n bits. MMM-algorithm first chooses a random initial point (RIP) R, computes $kP+R$ from left to right without any branch instruction

dependent on the data being processed, subtracts R from a result, and gets kP. By using a random initial point at each execution of exponentiation, any point or any register used in addition formulae changes at each execution, which prevents an attacker from controlling a point P itself as he needs. Thus, it is resistant to SPA, DPA, and RPA[3].

Let us briefly review MMM-algorithm (See Figure 1). First divide n bits into h blocks and choose a random point $R \in E(\mathbb{F}_p)$. Then, MMM-algorithm computes $kP+R$ by representing $kP+R$ to $(1\underbrace{\overline{11}\cdots\overline{11}}_{b})R + (\underbrace{k_{n-1}\cdots\cdots}_{b}\underbrace{\cdots k_1 k_0}_{b})P$ and executing the $(h+1)$-simultaneous scalar multiplication, where $b = \lceil \frac{n}{h} \rceil$. The simultaneous scalar multiplication first constructs a table of (2^h-1) points $T[0,s]$ for an h-bit integer $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$ $(s_\ell \in \{0,1\})$,

$$T[0,s] = \sum_{\ell=0}^{h-1} s_\ell 2^{b\ell} P - R,$$

and then repeats 1 addition to a table point $T[0,s]$ and 1 doubling from left to right in a b-bit block. The following is the detailed algorithm. In this paper, we also denote the algorithm by h-MMM-algorithm where we want to describe the number of divisions h.

Algorithm 2 (h-MMM-algorithm)
Input: $k = \sum_{i=0}^{n-1} k[i]2^i$, P
Output: kP
0. $k_j = \sum_{\ell=0}^{h-1} k[b\ell+j]2^\ell$ $(j \in \{0,\cdots,b-1\})$.
1. $T[0] =$randompoint().
Table construction
2. Compute $B[i] = 2^{bi}P$ for $0 \le i \le (h-1)$.
3. Compute $T[0,s] = \sum_{\ell=0}^{h-1} s_\ell B[\ell] - T[0]$ for an h-bit integer $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$ $(s_\ell \in \{0,1\})$. Therefore, $T[0,0] = -T[0]$.
Main computation
4. For $j = b-1$ to 0: $T[0] = 2T[0] + T[0,k_j]$.
5. Output $T[0] + T[0,0]$.

3 Generalized MMM-Algorithm

In this section, we give the generalized MMM-algorithm to improve the flexibility of tables as well as computational complexity, while it can resist SPA, DPA, and RPA. Our idea is inspired by an exponentiation algorithm with a fixed point [13,

[3] An SPA in the chosen-message-attack scenario [19] that uses a point P with order 2 is applied to MMM-algorithm. However, we can easily avoid this attack by checking $2P \neq \mathcal{O}$ before computing kP.

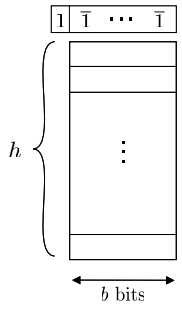

Fig. 1. MMM-algorithm

12][4]. Let k be n bits, h and v be positive integers, $\lceil \frac{n}{h} \rceil = a$, and $\lceil \frac{a}{v} \rceil = b$. Let us also use notation such as $t(2\mathcal{A} = \mathcal{J})$, $t(2^w\mathcal{J})$, $t(\mathcal{J} \to \mathcal{A})_{nI}$, $t(\mathsf{Minv}[n])$, and so on, defined in Section 2.

3.1 Algorithm Intuition

MMM-algorithm computes $kP + R$ by dividing k into $h \times 1$ blocks and executing the $(h+1)$-simultaneous-scalar multiplication of b-bit numbers, where $\lceil \frac{n}{h} \rceil = a$ and $\lceil \frac{a}{1} \rceil = b$. Our target is to generalize MMM-algorithm by dividing k into $h \times v$ blocks and executing the $(h+1)$-simultaneous-scalar multiplication of b-bit numbers. Therefore, MMM-algorithm is a special case of our generalization with $(h, v) = (h, 1)$.

Then, the issues to resolve are: how to embed a random point R into $h \times v$ blocks efficiently; and how to find the optimal (h, v) for the generalized MMM-algorithm together with the best coordinates. Regarding the former issue, one way is to use v random points R_i, compute $kP + R_1 + \cdots R_v$, and subtract $R_1 + \cdots + R_v$ from a result. Another way, which reduces the computational complexity, is to use a random point R in the same way as MMM-algorithm, compute $kP + vR$, and subtract vR. Regarding the latter issue, the best combination of coordinates in table construction, table points themselves, and main computation should be investigated for each (h, v) from the point of view of both computational and memory complexity.

3.2 GMMM-algorithm

Here we show the generalized MMM-algorithm, which is called GMMM-algorithm in this paper. The brief idea of GMMM-algorithm is described in Figure 2.

[4] The fixed-point exponentiation algorithm divides an exponent k into $h \times v$ blocks and makes a pre-computed table based on the blocks. The application to elliptic curves is discussed in [5]. MMM-algorithm could be considered as a combination of the exponentiation algorithm with $h \times 1$ blocks and a random initial point.

Algorithm 3 (**(h, v)-GMMM-algorithm**)
Input: $k = \sum_{i=0}^{n-1} k[i]2^i$, P
Output: kP
0. $k_{i,j} = \sum_{\ell=0}^{h-1} k[a\ell + j + bi]2^\ell$ $((i,j) \in \{0, \cdots, v-1\} \times \{0, \cdots, b-1\})$.
1. For $i = 0$ to $v-1$: $R[i]$ =randompoint().
Table Construction
2. Compute $B[i,\ell] = 2^{a\ell+bi}P$ for $0 \le i \le (v-1)$ and $0 \le \ell \le (h-1)$.
   Then $B[0,0] = P$.
3. Compute $T[i,s] = \sum_{\ell=0}^{h-1} s_\ell B[i,\ell] - R[i]$ for $0 \le i \le (v-1)$ and an h-bit integer $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$. Then $T[i,0] = -R[i]$ for $0 \le i \le (v-1)$.
Main computation
4. Initialization: $T[0] = R[0] + \cdots + R[v-1]$. $T[1] = -T[0]$.
5. For $j = b-1$ to 0 by -1
   main-loop: $T[0] = 2T[0] + \sum_{i=0}^{v-1} T[i, k_{i,j}]$.
6. Finalization: $T[0] = T[0] + T[1]$.
7. Output $T[0]$.

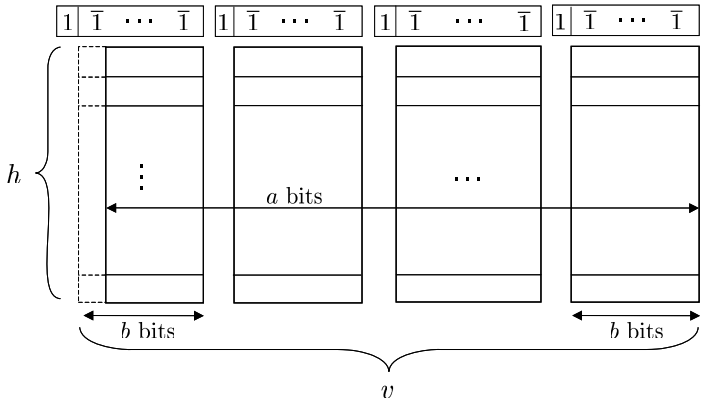

Fig. 2. Generalized MMM-algorithm

Remark:
1. To reduce the computational and memory complexity, 1 random point R would be used instead of v points $\{R[i]\}$. In this case, the initialization step in Algorithm 3 is changed to $T[0] = vR$; and only $T[0,0] = -R$ in the step 3 in Algorithm 3 is kept during the execution.
2. In the current GMMM-algorithm, one fixed power-consumption pattern is observed for any k with a bit length of $n \le bvh$ for the sake of a brief description. However, GMMM-algorithm can be described in such a way that 1 addition is saved for the first $bv - a$ rounds if the $(v-1)$-th block are not full.
3. GMMM-algorithm can also work with two divisions of $(h_1, v_1) \times (h_2, v_2)$ to

give the further wide range of time-memory tradeoffs, which will be shown in the final version of this paper.

Theorem 1 (Correctness). *GMMM-algorithm can compute kP correctly for a given elliptic-curve point P and a scalar $k = \sum_{i=0}^{n-1} k[i]2^i$.*

proof: For elliptic-curve points $P, R[0], \cdots, R[v-1]$ and a scalar k, set $B[i,\ell] = 2^{a\ell+bi}P$ for $0 \le i \le (v-1)$ and $0 \le \ell \le (h-1)$, $T[i,s] = \sum_{\ell=0}^{h-1} s_\ell B[i,\ell] - R[i]$ for $0 \le i \le (v-1)$ and an h-bit integer $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$, $k_{i,j} = \sum_{\ell=0}^{h-1} k[a\ell + j + bi]2^\ell$ ($(i,j) \in \{0,\cdots,v-1\} \times \{0,\cdots,b-1\}$), and $T[0] = R[0] + \cdots + R[v-1]$. Then, by describing 1 as a $(b+1)$-bit integer such as $1 = 1\underbrace{\overline{11}\cdots\overline{11}}_{b}$, we get

$$\begin{aligned} kP + R[0] + \cdots + R[v-1] &= T[0] + \sum_{j=0}^{b-1}\sum_{i=0}^{v-1}\left(\sum_{\ell=0}^{h-1} k[a\ell+bi+j]2^{a\ell+bi+j}P - 2^j R[i]\right) \\ &= T[0] + \sum_{j=0}^{b-1} 2^j \sum_{i=0}^{v-1}\left(\sum_{\ell=0}^{h-1} k[a\ell+bi+j]2^{a\ell+bi}P - R[i]\right) \\ &= T[0] + \sum_{j=0}^{b-1} 2^j \sum_{i=0}^{v-1} T[i,k_{i,j}]. \end{aligned}$$

Therefore, the `main-loop` in GMMM-algorithm computes $kP + R[0] + \cdots + R[v-1]$ and, thus, GMMM-algorithm can compute kP correctly. □

Theorem 2 (Security). *GMMM-algorithm is secure against SPA, DPA, and RPA.*

proof: GMMM-algorithm lets the power-consumption pattern be fixed regardless of the bit pattern of a secret key k, and thus it is resistant to SPA. GMMM-algorithm makes use of a random initial point at each execution and let all variables $\{T[i,s]\}$ be dependent on the random point. Thus, an attacker cannot control a point in such a way that it outputs a special point with a zero-value coordinate or zero-value register. Therefore, if $\{R[i]\}$ is chosen randomly by some ways, GMMM-algorithm can be resistant to DPA and RPA. □

In order to enhance the security against address-bit DPA[5] (ADPA) and an SPA in the chosen-message-attack scenario, the same discussion as MMM-algorithm holds in GMM-algorithm.

3.3 The Optimal Division with the Best Coordinate

Both MMM- and GMMM-algorithms aim at a random-point scalar multiplication and, thus, each execution starts with the table construction. Therefore,

[5] ADPA is one of DPA, which uses the leaked information from the address bus and can be applied on such algorithms that fix the address bus during execution.

both are evaluated by the total complexity of the table construction and main computation. MMM-algorithm has employed Jacobian coordinate in the whole procedures. Therefore, any pre-computed point is given in Jacobian coordinate as well as any computation being done in Jacobian coordinate. However, it should be revisited. Because the computational complexity of addition in Jacobian coordinate is considerably large even if iterated doublings in Jacobian coordinate compute the table construction efficiently (see Table 2.1).

Let us investigate the optimal (h, v) with the best coordinates in GMMM-algorithm by separating two phases of table construction and main computation. The table-construction phase computes, first, hv base points $B[i, \ell]$ by iterated doublings and, then, $(2^h - 1 - hv)$ points of $T[i, s] = \sum_{\ell=0}^{h-1} s_\ell B[i, \ell] - R[i]$ by only additions. As for the iterated doublings, Jacobian coordinate is the best coordinate as we have described the above. However, the base points themselves should be converted into affine coordinate to reduce the computational complexity of the next computation of $T[i, s]$. For the conversion from Jacobian to affine coordinate, we can apply Montgomery trick (Algorithm 1). Then, all base points $B[i, \ell]$ can be transformed into affine coordinate with the computational complexity of $3(2hv - 3)M + (hv - 1)S + I$. The computation of $T[i, s]$ can be executed simultaneously for each hamming weight of $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$. Therefore, affine coordinate with the Montgomery trick would be the best. By the above two procedures, we get a pre-computed table with affine coordinate. On the other hand, the main computation repeats additions to each pre-computed point $T[i, s]$ with affine coordinate and 1 doubling. Therefore, there exist two methods. One is mixed coordinate, in which main computation is done in Jacobian coordinate while pre-computed points are given in affine coordinate. The other is affine coordinate with the Montgomery trick, in which main computation is done in affine coordinate by applying the Montgomery trick in each iteration. Then, only 1 inversion is required in each iteration.

Let us summarize the above discussion as follows:

- `Table construction:`
  - `Repeated-doubling phase:` Jacobian coordinate,
  - `Simultaneous-addition phase:` affine coordinate with the Montgomery trick,
- `Main computation:`
  - **case 1:** mixed coordinates of Jacobian and affine coordinates,
  - **case 2:** affine coordinate with the Montgomery trick.

The best combination of coordinates depends on (h, v) and the ratio of I/M, which will be shown in Section 3.4.

3.4 Performance

Let us discuss the memory and computational complexity of (h, v)-GMMM-algorithm in both cases 1 and 2, where the case 1 constructs a table by repeated doublings in Jacobian coordinate and simultaneous additions in affine coordinate

with the Montgomery trick and executes the main computation in mixed coordinates of Jacobian and affine coordinates; and the case 2 constructs a table in the same way as the case 1 and executes the main computation in affine coordinate with the Montgomery trick. To make the discussion simple, we assume that 1 random point is used for 1 execution in the same way as MMM-algorithm.

As for the memory complexity, a table with $(2^h - 1)v$ points are required, which are represented in affine coordinate in both cases. In addition, 3 points of $T[0,0]$, $T[0]$, and $T[1]$ are used. All these points are given in affine coordinate in the case 1, while, in the case 2, only $T[0,0]$ is given in affine coordinate and the others are given in Jacobian coordinate.

Let us investigate the computational complexity by separating two phases of the table construction and the main computation. First, let us discuss the table construction phase, in which both cases 1 and 2 employ the same procedure. The table construction consists of the repeated-doubling part and the simultaneous-addition part. The repeated-doubling part computes $\{B[i,\ell]\}$ by executing the iterated doublings in Jacobian coordinate and transforming their results in Jacobian coordinate to affine coordinate. Thus, the total computational complexity of the repeated-doubling part is

$$t(2\mathcal{A}=\mathcal{J})+t(2^{b(v-1)+a(h-1)-1}\mathcal{J})+(hv-1)t(\mathcal{J}\to\mathcal{A})_{nI}+t(\mathsf{Minv}[hv-1]) \text{ if } vh\neq 1.$$

In the case of $vh = 1$, we can skip the repeated-doubling part. The simultaneous-addition phase computes $T[i,s] = \sum_{\ell=0}^{h-1} s_\ell B[i,\ell] - R[i]$ simultaneously for each hamming weight of $s = \sum_{\ell=0}^{h-1} s_\ell 2^\ell$. Therefore, it starts with the hamming weight 1 of s, that is, computes $\{B[i,\ell] - R[i]\}_{\ell,i}$ simultaneously by executing additions in affine coordinate together with Montgomery's trick. Thus, the total computational complexity of the simultaneous-addition part is

$$\sum_{i=1}^{h}\left(v\binom{h}{i}t(A+A)_{nI}+t(\mathsf{Minv}[v\binom{h}{i}])\right),$$

where $\binom{h}{i}$ means the combination to choose i elements from h elements. Then, we've got a precomputed table in affine coordinate.

Let us discuss the main-computation phase, in which each case employs each different procedure. The main-computation phase consists of initialization, main loop, and finalization (step 4, 5, and 6 in Algorithm 3, respectively). Let us focus on the case 1, that is the mixed coordinates of Jacobian and affine coordinates. The computational complexity of the main loop is

$$a\cdot t(\mathcal{J}+\mathcal{A}=\mathcal{J})+b\cdot t(2\mathcal{J}).$$

The computational complexity of initialization (resp. finalization) is that to compute $T[0] = vR$ for R in affine coordinate giving a result in Jacobian coordinate (resp. $T[0] + T[1]$ of points in Jacobian coordinate giving a result in affine coordinate).

Let us focus on the case 2 of affine coordinate with the Montgomery trick. As for the computation of the main-loop, a tournament structure with $v + 2$

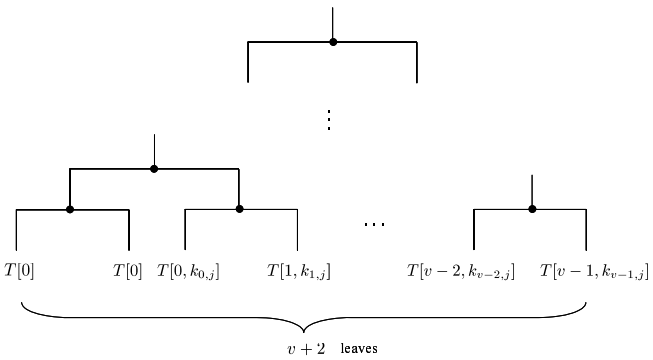

Fig. 3. Main computation of GMMM-algorithm in the case 2

leaves is applied, where v points of $T[i,k_{i,j}]$ and doubling points of $T[0]$ are in the leaf of the tournament structure (see Figure 3). We pairwise add points at leaves with a common parent, give its sum to the parent node, and carry out these procedures at each level to the root. By applying the Montgomery's trick to each level, only 1 inversion is required in each level. The simplest case is the binary tree case, which is described in [15]. Then, the computational complexity of the main loop is

$$a \cdot t(\mathcal{A}+\mathcal{A})_{nI} + b \cdot t(2\mathcal{A})_{nI} + b \cdot t(\mathsf{TournaMinv}[v+2]),$$

where $t(\mathsf{TournaMinv}[v+2])$ denotes the computational complexity to get all inversions of $(v+2)$-point summation according to the tournament structure with $(v+2)$ leaves. The computational complexity of initialization (resp. finalization) is that to compute $T[0]=vR$ of R in affine coordinate giving a result in affine coordinate (resp. $T[0]+T[1]$ of points in affine coordinate giving a result in affine coordinate).

The above discussion is summarized in the following theorem.

Theorem 3. *The total computational complexity of (h,v)-GMMM-algorithm with 1 random point, Comp, is given as follows:*
1. In the case 1: mixed coordinates in the main computation,

$$\begin{aligned}Comp &= t(2\mathcal{A}=\mathcal{J}) + t(2^{b(v-1)+a(h-1)-1}\mathcal{J}) + (hv-1)t(\mathcal{J}\to\mathcal{A})_{nI} + t(\mathsf{Minv}[hv-1]) \\ &\quad + \textstyle\sum_{i=1}^{h}\left(v\binom{h}{i}t(A+A)_{nI} + t(\mathsf{Minv}[v\binom{h}{i}])\right) + t(v\mathcal{A}=\mathcal{J}) \\ &\quad + a\cdot t(\mathcal{J}+\mathcal{A}=\mathcal{J}) + b\cdot t(2\mathcal{J}) + t(\mathcal{J}+\mathcal{J}=\mathcal{A}) \ \ (\textit{if } vh\neq 1).\end{aligned}$$

$$Comp = t(\mathcal{A}+\mathcal{A}) + a\cdot t(\mathcal{J}+\mathcal{A}=\mathcal{J}) + b\cdot t(2\mathcal{J}) + t(\mathcal{J}+\mathcal{A}=\mathcal{A}) \ \ (\textit{if } vh=1).$$

2. In the case 2: affine coordinate with the Montgomery's trick in the main computation

$$\begin{aligned}Comp &= t(2\mathcal{A}=\mathcal{J}) + t(2^{b(v-1)+a(h-1)-1}\mathcal{J}) + (hv-1)t(\mathcal{J}\to\mathcal{A})_{nI} + t(\mathsf{Minv}[hv-1]) \\ &\quad + \textstyle\sum_{i=1}^{h}\left(v\binom{h}{i}t(A+A)_{nI} + t(\mathsf{Minv}[v\binom{h}{i}])\right) + t(v\mathcal{A}) \\ &\quad + a\cdot t(\mathcal{A}+\mathcal{A})_{nI} + b\cdot t(2\mathcal{A})_{nI} + b\cdot t(\mathsf{TournaMinv}[v+2]) + t(\mathcal{A}+\mathcal{A}) \ \ (\textit{if } vh\neq 1),\end{aligned}$$

where $\lceil\frac{n}{h}\rceil = a$, $\lceil\frac{a}{v}\rceil = b$, and $t(v\mathcal{A})$ (resp. $t(v\mathcal{A}=\mathcal{J})$) denote the time to

compute v-multiple points in affine coordinate giving a result in affine (resp. Jacobian) coordinate.

4 Comparison

In this section, we compare our algorithm with the previous countermeasures to SPA, DPA, and RPA, those are ES [3], randomized window algorithm [16], LRIP [11], and MMM-algorithm [14]. Here, M, S, or I represents the computation amount of modular multiplication, square, or inversion, respectively. In order to make comparison easier, the computation complexity is also estimated in terms of M and I by assuming that $S = 0.8M$ as usual and that $S = 0.8M$ and $I = 4M$ or $I = 11M$ (typically the ratio[6] I/M is between 4 and 11). Memory complexity is evaluated by the number of finite-field elements, where 1 point in Jacobian (resp. affine) coordinate consists of 3 (resp. 2) field elements.

Table 4.1 shows the computational and memory complexity of GMMM-algorithm with cases 1 and 2 in the case of a 160-bit scalar. These are arranged in ascending order of memory. Therefore, if we focus on either case of GMMM-algorithm, these are also arranged in descending order of computational complexity. However, the case 1 has advantage over the case 2 if the ratio of inversion over multiplication is rather large and, thus, better case depends on the ratio of I/M. We also compute a break-even point for the borderline between both cases. The break-even point shows I/M when the computational complexity of GMMM-algorithm with the case 1 is equal to that with the case 2 under $S = 0.8M$. Thus, if I/M is smaller than the indicated value, GMMM-algorithm with the case 2 is more efficient than that with the case 1. For example, $(3,1)$-GMMM with the case 1 is more efficient than $(3,2)$-GMMM with the case 2 if and only if $I/M > 8.4$. Each division of $(1,1)$, $(2,1)$, $(3,1)$ or $(4,1)$-GMMM-algorithm corresponds to that of 1, 2, 3, or 4-MMM-algorithm, respectively. The difference is: MMM-algorithm uses Jacobian coordinate in the whole execution but GMMM-algorithm with the case 1 employs mixed coordinate. Note that GMMM-algorithm with the case 2 can not be applied to these cases.

Table 4.2 shows the computational and memory complexity of previous algorithms in the case of a 160-bit scalar, where Jacobian coordinate is used for the whole computation and a result is transformed into affine coordinate according to their original proposals.

By generalizing MMM-algorithm to GMMM-algorithm, we see that more flexibility, that is a wider range of time-memory tradeoffs, can be realized. Furthermore, the optimization of coordinates can reduce both the computational complexity and memory even if both MMM and GMMM-algorithms use the same division. In fact, GMMM-algorithm performs better than any previous method under the above realistic assumptions concerning the ratio I/M. For example, (1,1)-GMMM-algorithm with the case 1 can reduce the computational

[6] Generally, the ratio of I/M depends on the algorithm used and the size and type of the finite field. The ratio in [7, 2] (resp. [6]) is between 4 and 10 (resp. 4 and 11) . So here we adopt the wider range. A discussion on the ratio can be found in [4].

complexity of 1-MMM-algorithm by 19% over for the range of I/M and also reduce the memory size. (4,1)-GMMM-algorithm with the case 1 can reduce the computational complexity of 4-MMM-algorithm by 13.2% over for the range of I/M and also reduce the memory by 25.4 %. Note that 4-MMM-algorithm is the most efficient case in MMM-algorithm[7]. However, even $(2,2)$-GMMM-algorithm with the case 2 or $(3,1)$-GMMM-algorithm with the case 1 can work faster than 4-MMM-algorithm under the range of $I/M < 7.9$ or 49.3 and also reduce the memory by 64.8% or 62.8%, respectively.

Table 4.1. Performance of GMMM-algorithm (160 bits)

division (h,v) case 1 or 2	Computational complexity				Memory[‡]	$I/M^{\star}$
		$S=0.8M$	$I=4M$	$I=11M$		
$(1,1)^{\dagger}$-case 1	$1933M+1445S+2I$	$3089M+2I$	$3097M$	$3111M$	7	
$(1,2)$-case 2	$1052M+648S+164I$	$1570.4M+164I$	$2226.4M$	$3374.4M$	10	
$(1,2)$-case 1	$1945M+1294S+3I$	$2980.2M+3I$	$2992.2M$	$3013.2M$	12	8.8
$(2,1)^{\dagger}$-case 1	$1305M+1051S+4I$	$2145.8M+4I$	$2161.8M$	$2189.8M$	14	
$(2,2)$-case 2	$881M+654S+85I$	$1404.2M+85I$	$1744.2M$	$2339.2M$	18	
$(2,2)$-case 1	$1334M+980S+4I$	$2118M+4I$	$2134M$	$2162M$	20	8.8
$(3,1)^{\dagger}$-case 1	$1128M+934S+5I$	$1875.2M+5I$	$1895.2M$	$1930.2M$	22	
$(3,2)$-case 2	$873M+672S+60I$	$1410.6M+60I$	$1650.6M$	$2070.6M$	34	8.4
$(4,1)^{\dagger}$-case 1	$1051M+865S+6I$	$1743M+6I$	$1767M$	$1809M$	38	
$(4,2)$-case 2	$919M+682S+47I$	$1464.6M+47I$	$1652.6M$	$1981.6M$	66	6.8

[†] : They correspond to h-MMM-algorithm. [‡] : # field elements. [⋆] : break-even point

Table 4.2. Comparison of known countermeasures (160 bits)

	Computational complexity			Memory
		$S=0.8M$	$I=11M$	
ES [3]	$2563M+1601S+I$	$3843.8M+I$	$3854.8M$	14[◇]
strengthened window [16]	$1643M+1298S+I$	$2681.4M+I$	$2692.4M$	15
LRIP [11]	$2563M+1283S+I$	$3589.4M+I$	$3600.4M$	12
1-MMM [14]	$2563M+1601S+I$	$3843.8M+I$	$3854.8M$	9
2-MMM	$1651M+1139S+I$	$2562.2M+I$	$2573.2M$	15
3-MMM	$1395M+1007S+I$	$2200.4M+I$	$2211.4M$	27
4-MMM	$1315M+947S+I$	$2072.4M+I$	$2073.4M$	51

[◇]: Strictly, it needs 4 elliptic curve points and 2 field elements.

5 Conclusion

In this paper, we have generalized MMM-algorithm, which is the secure scalar multiplication with using a random initial point. Our improved algorithm is sig-

[7] 5-MMM-algorithm is not as efficient as 4-MMM-algorithm although it requires more memory than 4-MMM-algorithm.

nificantly efficient and flexible and can work efficiently even when the storage available is very small or quite large. We have also given the formulae of the computational complexity for any division of the proposed algorithm theoretically, which helps developers to choose the best division suitable for the storage available.

References

1. R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, *Handbook of Elliptic and Hyperelliptic Curve Cryptography*, Chapman & Hall/CRC, 2006.
2. I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptology*, LMS **265**(1999), Cambridge University Press.
3. M. Ciet and M. Joye, "(Virtually) Free randomization technique for elliptic curve cryptography", *Proceedings of ICICS2003*, LNCS **2836**(2003), Springer-Verlag, 348–359.
4. M. Ciet, M. Joye, K. Lauter, and P. L. Montgomey, "Trading inversions for multiplications in elliptic curve cryptography", *Designs, Codes and Cryptography*, Vol. **39**, No. **2**(2006), Springer Netherlands, 189–206.
5. H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation", *Proceedings of ICICS'97*, LNCS **1334**(1997), Springer-Verlag, 282-290.
6. H. Cohen, A. Miyaji and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates", *Advances in Cryptology-Proceedings of ASIACRYPT'98*, LNCS **1514**(1998), Springer-Verlag, 51-65.
7. C. Doche, T. Icart, and D. R. Kohel, "Efficient scalar multiplication by isogeny decompositions", *Proceedings of PKC2006*, LNCS **3958**, 191–206.
8. K. Eisenträger, K. Lauter, and P. L. Montgomey, "Fast elliptic curve arithmetic and improved Weil pairing evaluation", *Proceedings of CT-RSA2003*, LNCS **2612**(2003), Springer-Verlag, 343–354.
9. L. Goubin, "A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems", *Proceedings of PKC2003*, LNCS **2567**(2003), Springer-Verlag, 199–210.
10. K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara, "Fast implementation of public-key cryptography on DSP TMS320C6201", *Proceedings of CHES'99*, LNCS **1717**(1999), Springer-Verlag, 61–72.
11. K. Itoh, T. Izu, and M. Takenaka, "Efficient countermeasures against power analysis for elliptic curve cryptosystems", *Proceedings of CARDIS 2004*, Kluwer, 99–114.
12. C. H. Lim and P. J. Lee, "More flexible exponentiation with precomputation", *Advances in Cryptology-Proceedings of Crypto'94*, LNCS **839**(1994), Springer-Verlag, 95-107.
13. N. Pippenger, "On the evaluation of powers and related problems (preliminary version)", *17th annual symposium on foundations of computer science*, IEEE Computer Society, 1976, 258–263.
14. H. Mamiya, A. Miyaji and H. Morimoto. "Secure elliptic curve exponentiation against RPA, ZRA, DPA, and SPA." *IEICE Trans.*, Fundamentals. vol. E89-A, No.8(2006), 2207-2215.
15. P. K. Mishra, P. Sarkar, "Application of Montgomery's trick to scalar multiplication for EC and HEC using fixed base point", *Proceedings of PKC2004*, LNCS **2947**, 41–57.

16. B. Möller, "Parallelizable elliptic curve point multiplication method with resistance against side-channel attacks", *Proceedings of ISC2002*, LNCS **2433**(2002), Springer-Verlag, 402–413.
17. P. L. Montgomery, "Speeding the Pollard and elliptic curve methods for factorization", *Mathematics of Computation*, **48**(1987), 243-264.
18. Standard for efficient cryptography group, specification of standards for efficient cryptography. Available from: http://www.secg.org
19. S. M. Yen. W. C. Lien, S. Moon, and J. Ha, "Power analysis by exploiting chosen message and internal collisions - Vulnerability of checking mechanism for RSA-Decryption", *Proceedings of Mycrypt 2005*, LNCS **3715**(2005), Springer-Verlag, 183-195.