IPSJ Journal Vol. 49 No. 9 2975–2988 (Sep. 2008)

Regular Paper

Revisited (Hyper)-Elliptic Curve Scalar Multiplication with a Fixed Point [⋆1]

Atsuko Miyaji[†1] and Kenji Mizosoe[†1]

Elliptic curve cryptosystems can be constructed over a smaller definition field than the ElGamal cryptosystems or the RSA cryptosystems. This is why elliptic curve cryptosystems have begun to attract notice. This paper explores an efficient fixed-point-scalar-multiplication algorithm for the case of a definition field $\mathbb{F}_p$ ($p > 3$) and adjusts coordinates to the algorithm proposed by Lim and Lee. Our adjusted algorithm can give better performance than the previous algorithm in some sizes of the pre-computed tables.

1. Introduction

An elliptic curve cryptosystem, which is constructed on the group of points of an elliptic curve over a finite field, can provide a small and fast public key cryptosystem if the elliptic curve is chosen appropriately [2),15)]. This is why elliptic curve cryptosystems have been widely used in various applications. An elliptic curve cryptosystem consists of an elliptic curve scalar multiplication kP.

There are two approaches for an efficient elliptic curve scalar multiplication kP. One is the scalar multiplication for a randomly given point P, which is executed in a signature verification, an encryption, or a decryption. The other is that for a fixed point P, which is executed in a signature generation or an encryption. Both are usually studied independently and evaluated from slightly different viewpoints. From the point of view of the computational amount, the random-point-scalar-multiplication algorithm is evaluated by the total computational complexity of both making a pre-computed table and the main computation with the pre-computed table. This is because the input point is not given beforehand. On the other hand, the fixed-point-scalar-multiplication algorithm is evaluated by the computational amount of only the main computation with the pre-computed table. This is because the input point is given beforehand and consequently every scalar multiplication can start with the main computation by making the pre-computed table beforehand. From the viewpoint of the memory, both are evaluated by the size of the pre-computed table.

Many researches on the random-point-scalar-multiplication algorithm have been proposed so far[1),7),18)]. On the other hand, two researches on the fixed-point scalar multiplication algorithms have been proposed so far, the BGMW [3)]- and LL [14)]-algorithms, which are each based on different concepts. The BGMW-algorithm uses the base of 2^w, holds the precomputed points based on $2^{wi}P$, and computes kP by repeating only additions. The LL-algorithm divides k into blocks of $h \times v$ (1 division) or $h_1 \times v_1 + h_2 \times v_2$ (2 divisions), holds the pre-computed points based on each block, and computes kP by repeating additions and doubling. The LL-algorithm is more flexible and can give a wider range of time-memory tradeoffs than the BGMW-algorithm. This paper focuses on the LL-algorithm over an elliptic curve.

The implementation time of elliptic curve scalar multiplications depends on three different factors: the definition field, the coordinate systems, and the algorithm itself such as BGMW- or LL-algorithm. This paper focuses on two factors: the coordinate systems and the efficient fixed-point scalar multiplication. There are several coordinates systems, affine ($\mathcal{A}$), Jacobian ($\mathcal{J}$), modified Jacobian, projective, and mixed Jacobian coordinates of the combination between $\mathcal{A}$ and $\mathcal{J}$ [7)]. All of these are defined in the Weierstrass form. In addition to the Weierstrass form, the Edward form was proposed [10)]. In the Edward form, affine coordinates ($\mathcal{E}_\mathcal{A}$), homogeneous coordinates ($\mathcal{E}$), and mix Edward coordinates of the combination between $\mathcal{E}_\mathcal{A}$ and $\mathcal{E}$ can be considered, however, $\mathcal{E}_\mathcal{A}$ is not usually discussed since it is rather slow. Note that the doubling and addition of Edward coordinates are faster than those of Jacobian coordinates, however, mixed Edward coordinates are slower than mixed Jacobian coordinates. We also note that addition in $\mathcal{A}$ is not necessarily slower than that in $\mathcal{E}$ or in mixed coordinates of $\mathcal{A}$ and $\mathcal{J}$.

Generally, the Jacobian coordinate system has been adopted for the random-

†1 Japan Advanced Institute Science and Technology

⋆1 This study is partly supported by Grant-in-Aid for Scientific Research (B), 17300002, and Yazaki Memorial Foundation for Science and Technology.

© 2008 Information Processing Society of Japan

point-scalar multiplication so far [1),7)] since doublings become the dominant part. This is why Edward coordinates are expected to give a faster random-point-scalar multiplication. On the other hand, the dominant part of the LL-algorithm is rather different. There would be two ways to execute the LL-algorithm over an elliptic curve, both of which assume that the points in the pre-computed table are given in affine coordinates. One way adopts Jacobian/Edward coordinates and its mixed coordinates. The other adopts affine coordinates in the entire algorithm. In the former case, the dominant factor is the computational complexity of addition in mixed coordinates. In the latter case, the dominant factor is the computational complexity of addition in affine coordinates. The performance of the LL-algorithm with 1-division in Jacobian and its mixed coordinates is shown in Ref. 6), where the optimal division (h, v) is also investigated. However, neither 2-division in Jacobian and its mixed coordinates nor the LL-algorithm in Edward and its mixed coordinates have been investigated in any paper so far. Furthermore, the applicability of affine coordinates to the LL-algorithm has not been investigated so far. There is room for further study.

Generally, affine coordinates are avoided for the scalar multiplication in the case of a definition field $\mathbb{F}_p$ (with p larger than 3) since it suffers from the heavy use of inverses. Recently, a new derivation of the SPA-resistant algorithm with a fixed point was proposed, which adopts affine coordinates by applying the Montgomery's trick [17)] to reduce the number of inverses [16)]. Their work cannot be directly extended to the LL-algorithm because the structure of the precomputed table is totally different in each case. However, their work inspires us to explore the possibility of further improvement of a fixed-point-scalar-multiplication algorithm.

In the present paper, we revisit the LL-algorithm by adopting affine coordinates; applying the Montgomery's trick to compute several inverses simultaneously; and optimizing the table's structure. We analyze the computational complexity of the proposed algorithm theoretically. Furthermore, we reconsider the division procedure in the case of $h_1 \times v_1 + h_2 \times v_2$ in order to give a tight adjusted division. We also discuss the efficiency of our proposed algorithm for the cases of the elliptic curve and the hyper-elliptic curve and show the break-even point of our algorithm against the previous algorithm. The break-even point means the ratio of I/M based on the assumption of $S = 0.8M$ when the computational complexity of our algorithms is equal to that of the previous one, where I, M, or S expresses the computational complexity of 1 modular inverse, multiplication, or square on the 160-bit field, respectively. Our algorithm inherits the flexibility of the original LL-algorithm. In the case of an elliptic curve, our algorithm with pre-computed points of 18, 26, 34, 42, or 50 can compute kP more efficiently than the previous algorithm with the best coordinates [⋆1] in the break-even point of $I/M < 12.2, 12.7, 12.7, 12.8$, or 12.3, respectively. In fact, the computational complexity of our algorithm can be reduced to 95%, 93%, 93%, 93%, or 94% of the previous algorithm with the best coordinate system in the case of $I/M = 11$. In the case of a hyper-elliptic curve with genus 2, our algorithm with pre-computed points of 18 (resp. 28, 30, 42, 46, 58) can compute kP more efficiently than the previous algorithm in the break-even point of $I_{hec}/M_{hec} > 2.9$ (resp. 3). That is, our algorithm is usually more efficient than the previous one. Here, I_{hec}, M_{hec}, or S_{hec} expresses the computational complexity of 1 modular inverse, multiplication, or square on the 80-bit field, respectively. In the case of $I_{hec} = 11M_{hec}$, the computational complexity of our algorithm with a pre-computed table of 18, 28, 30, 42, 46, or 58 can be reduced to 92%, 92%, 93%, 87%, 93%, or 88% of the previous algorithm, respectively.

This paper is organized as follows. Section 2 summarizes the known facts on (hyper-) elliptic curves. Section 3 reviews some known algorithms such as the LL-algorithm and the Montgomery trick. Section 4 presents our new scalar multiplication algorithm with a fixed point and analyzes the computational complexity theoretically. Section 5 provides the cases where our results are better than the previous results. It also describes the performance of the previous algorithm with Jacobian and Edward coordinates for comparison. Appendix A.1 summarizes the addition formulae in Edward coordinates.

⋆1 The best coordinate system for the previous algorithm is still Jacobian coordinate system as mentioned in Ref. 6) since mixed Edward coordinate system is slower than mixed Jacobian coordinate system. The adaptability and performance of Edward coordinate to the previous algorithm is also investigated in this paper.

© 2008 Information Processing Society of Japan

2. Preliminaries

This section summarizes some facts on elliptic curves and hyper elliptic curves such as the coordinate systems.

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 of 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 coordinates 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 Ref. 16). Each module means preparation for 1 inverse, computation of 1 inverse and computation without any inverse, respectively. The addition formulae are given as follows.

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

$\mathsf{Dbl}(P_1)$	
$\mathsf{Dbl_p}(P_1)$:	$\alpha = 2y_1$
$\mathsf{Dbl_I}(\alpha)$:	$\lambda = \frac{1}{\alpha}$
$\mathsf{Dbl_{NI}}(P_1, \lambda)$:	$\gamma = (3x_1^2 + a)\lambda$
	$x_3 = \gamma^2 - 2x_1$
	$y_3 = \gamma(x_1 - x_3) - y_1$

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 coordinates and multiplication (resp. inverse, resp. squaring) in $\mathbb{F}_p$ by M (resp. I, resp. S), where $\mathcal{A}$ means affine coordinates. However, it is usual to neglect the addition, the subtraction, or the multiplication by a small constant in $\mathbb{F}_p$ when evaluating the computational complexity. Then we see that $t(\mathcal{A} + \mathcal{A}) = I + 2M + S$ and $t(2\mathcal{A}) = I + 2M + 2S$.

Both addition and doubling formulae need one inversion over $\mathbb{F}_p$, which is much more expensive than a multiplication over $\mathbb{F}_p$. Affine coordinates are transformed into Jacobian coordinates [7], where the inversion is not used. 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 represented by $(1, 1, 0)$. Let $P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2)$, and $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$. The addition formulae have gradually been improved after they were introduced widely [6]. Here we show the latest addition formulae [4]. The total number of S and M in the latest addition formulae is the same as before, however, they decrease M and increase S. Therefore, they usually reduce the total computational complexity since $S < M$. For comparison, Appendix A.1 shows the previous addition formulae. The doubling and addition formulae in Jacobian coordinates can be represented as follows.

$\mathsf{Add}^{\mathcal{J}}(P_1, P_2)$ $(P_1 \neq \pm P_2)$
$U_1 = X_1Z_2^2$, $U_2 = X_2Z_1^2$,
$S_1 = Y_1Z_2^3$, $S_2 = Y_2Z_1^3$,
$H = U_2 - U_1$, $R = 2(S_2 - S_1)$,
$I = (2H)^2$, $J_1 = IH$, $J_2 = IU_1$
$X_3 = R^2 - J_1 - 2J_2$,
$Y_3 = R(J_2 - X_3) - 2S_1J_1$,
$Z_3 = ((Z_1 + Z_2)^2 - Z_1^2 - Z_2^2)H$.

$\mathsf{Dbl}^{\mathcal{J}}(P_1)$	
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	$= (Y_1 + Z_1)^2 - Y_1^2 - Z_1^2$

The computation times in Jacobian coordinates are $t(\mathcal{J} + \mathcal{J}) = 11M + 5S$ and $t(2\mathcal{J}) = 2M + 8S$, where $\mathcal{J}$ means Jacobian coordinates.

There are several coordinate systems, affine ($\mathcal{A}$), Jacobian ($\mathcal{J}$), and their combination, called mixed coordinates [7]. In addition to the Weierstrass form, the Edward form was proposed [10], which is described in Appendix A.1. In the Edward form, affine coordinates ($\mathcal{E}_{\mathcal{A}}$) and homogeneous coordinates ($\mathcal{E}$) can be considered as usual, however, $\mathcal{E}_{\mathcal{A}}$ is not usually discussed since both addition and doubling in $\mathcal{E}_{\mathcal{A}}$ are rather slower than those in $\mathcal{E}$ as well as those in $\mathcal{A}$. We can also consider mix coordinates of $\mathcal{E}_{\mathcal{A}}$ and $\mathcal{E}$. These performances are summarized in **Table 1**. Note that the doubling and addition of Edward coordinates are faster than those of Jacobian coordinates, however, for mixed Edward coordinates they are slower than for mixed Jacobian coordinates since $S < M$.

There are two ways to execute the LL-algorithm, both of which assume points in the pre-computed table are given in affine coordinates. One way adopts Ja-

© 2008 Information Processing Society of Japan

Table 1 Comparison of elliptic-curve coordinates.

	computation amount
$t(\mathcal{A}+\mathcal{A})$	$2M+S+I$
$t(2\mathcal{A})$	$2M+2S+I$
$t(\mathcal{J}+\mathcal{J})$	$11M+5S$
$t(\mathcal{J}+\mathcal{A}=\mathcal{J})$	$7M+4S$
$t(2\mathcal{J})$	$2M+8S$
$t(\mathcal{E}+\mathcal{E})$	$11M+S$
$t(\mathcal{E}+\mathcal{A}_{\mathcal{E}}=\mathcal{E})$	$10M+S$
$t(2\mathcal{E})$	$3M+4S$

cobian/Edward coordinates and its mix coordinates. The other adopts affine coordinates in the entire algorithm. In the former case, the dominant factor of computational complexity is the computational complexity in mixed coordinates. In the latter case, the dominant factor of computational complexity is the computational complexity of addition in affine coordinates. As a result, affine coordinates $\mathcal{A}$ or mixed coordinates of $\mathcal{J}$ and $\mathcal{A}$ are faster than mixed coordinates of $\mathcal{E}$ and $\mathcal{E}_{\mathcal{A}}$ when executing the LL-algorithm. These adaptability will be discussed in Section 4.

2.2 Hyper-Elliptic Curve

A hyper-elliptic curve $C/\mathbb{F}_p(p>3)$ with genus 2 is described as

$$C: Y^2 = F(X) = X^5 + f_4X^4 + f_3X^3 + f_2X^2 + f_1X + f_0,$$

where $F(X)$ is in $\mathbb{F}_p[X]$. In the case of $p \neq 5$, we can set $f_4 = 0$. The divisors of a hyper-elliptic curve are defined as the free abelian group of points $P_1, \ldots, P_r \in C$, $D = \sum_{P_i \in C} m_iP_i,\ m_i \in \mathbb{Z}$. The degree of D is defined as $\sum_{P_i \in C} m_i$ and the order at P_i in C is defined as $m_i = ord_{P_i}(D)$. The Jacobian variety J_C is defined as D^0/D^l, where D^0 is a group of divisors with degree 0 and D^l is a group of divisors of functions. Any divisor $D \in J_C$ is equivalent to a divisor called a reduced divisor modulo D^l,

$$D \sim \sum_{P_i \in C} m_i'' P_i - rP_\infty \ (r = \sum_{P_i \in C} m_i'' \leq g),$$

where g is a genus of C. To compute an addition of divisors, Mumford-representation is useful. In Mumford-representation, $D \in C$ with the genus 2 is described by $D = (u_1, u_0, v_1, v_0)$, where

Table 2 Comparison of hyper-elliptic-curve coordinates.

	computation amount
$t(\mathcal{A}_{hec}+\mathcal{A}_{hec})$	$22M_{hec}+3S_{hec}+I_{hec}$
$t(2\mathcal{A}_{hec})$	$22M_{hec}+5S_{hec}+I_{hec}$
$t(\mathcal{J}_{hec}+\mathcal{J}_{hec})$	$40M_{hec}+6S_{hec}$
$t(2\mathcal{J}_{hec})$	$47M_{hec}+4S_{hec}$

$$U = \prod_{P_i}(X - x_i)^{m_i''} = X^2 + u_1X + u_0 \in \mathbb{F}_p[X],$$
$$V = v_1X + v_0 \in \mathbb{F}_p[X],$$
$$V^2 \equiv F(X) \pmod{U(X)}.$$

The addition formulae in the case of the genus 2 are proposed in Ref. 13), on which the efficiency of our proposal is discussed. The computational complexity of addition formulae is presented in **Table 2**. We may note that the size of the definition field of a secure elliptic curve is about g times as large as that of a secure hyper-elliptic curve and that affine coordinates are usually faster Jacobian coordinates. Therefore, in the case of the genus 2, the size of the definition field of a hyper-elliptic curve is about half that of an elliptic curve. In order to distinguish between each definition field, let us use the following notation: we denote the affine coordinates (resp. Jacobian coordinates) by $\mathcal{A}_{hec}$ (resp. $\mathcal{J}_{hec}$) and represent the multiplication (resp. inverse, resp. squaring) in the definition field by M_{hec} (resp. I_{hec}, resp. S_{hec}).

2.3 Inversion over a Prime Field

This subsection briefly describes a ratio of I/M. A ratio of I/M depends on the algorithm used and the size of the field. The well-known fast algorithms are the extended Euclid algorithm due to Lehmer [15)] and the Montogomery inverse [11),12)]. A ratio is estimated between 3 and 10 in the literature [2)] or 4 and 10 [8)]. In fact, some software implementations without the Montgomery inversion yield a ratio of 3.8 or 4.8 for a 160-bit or 256-bit prime field, respectively [9)]. On the other hand, some hardware implementations with the Montgomery inversion yield a ratio of 4.18, 5.00, 5.42, or 6.23 for a prime field with 160, 192, 224, or 256 bits, respectively [12)]. A discussion on the ratio can be found in Ref. 5).

© 2008 Information Processing Society of Japan

3. Previous Work

3.1 Lim-Lee Algorithm

We briefly review the LL-algorithm [14], one of previous scalar multiplication algorithms with precomputation. The efficiency in the case of the elliptic curve is re-discussed [6]. Let us assume that the size of the underlying field and the scalar k are n bits. The LL-algorithm consists of two phases: the pre-computation phase and the main computation phase. The pre-computation makes $(2^h-1)\times v$ points $P[i,s]$ $((i,s)\in\{0,\cdots,v-1\}\times\{1,\cdots,2^h-1\})$,

$$P[i,s]=\sum_{j=0}^{h-1} s_j 2^{bi+aj}P,$$

where $a=\lceil\frac{n}{h}\rceil$ and $b=\lceil\frac{a}{v}\rceil$ and $s_0,\cdots,s_{h-1}$ is a representation of s in radix 2, that is $s=\sum_{j=0}^{h-1}s_j2^j$. On the other hand, the main computation divides k into $h\times v$ blocks and computes kP by repeating v additions of pre-computed points $P[i,s]$ and 1 doubling. Note that the $(v-1)$-th block may not be full since the $bv-a$ bits are empty (see **Fig. 1**). Therefore, 1 addition can be saved for the first $bv-a$ rounds. The LL-algorithm focuses on the case where P is given beforehand such as a basepoint; it assumes that the precomputed points are given beforehand, and, thus, it does not take the computational complexity of the pre-computation into account.

Algorithm 1

```
Input:  k = Σ_{i=0}^{n-1} k[i]2^i, P, {P[i,j]}
Output: kP
0.  k_{i,j} = Σ_{ℓ=0}^{h-1} k[aℓ+j+bi]2^ℓ
    ((i,j) ∈ {0,···,v-1} × {0,···,b-1}).
1.  If bv-a > 0, then ℓ_e = v-1.
    Else ℓ_e = v.
2.  T = Σ_{i=0}^{ℓ_e-1} P[i,k_{i,b-1}].
3.  For j = b-2 to 0 by -1
4.      If j ≥ b-(bv-a), then ℓ_e = v-1.
        Else ℓ_e = v.
5.      main loop:  T = 2T + Σ_{i=0}^{ℓ_e-1} P[i,k_{i,j}].
6.  Output T.
```

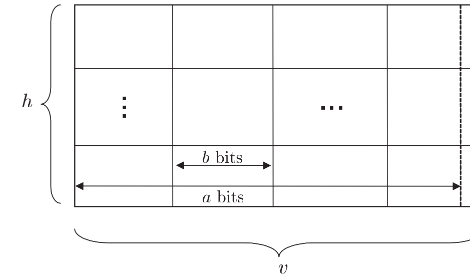

Fig. 1 Division of n with (h,v).

The memory and the computational complexity of Algorithm 1 are $(2^h-1)\cdot v$ points and $(b-1)D+(a-1)A$, respectively. Here, A (resp. D) represents the computational complexity of addition (resp. doubling).

The following is an example of Algorithm 1.

Example: Let $k=100010\ 001000$ (12-bit number) and $(h,v)=(2,3)$. Then, kP is computed as follows. First, $(a,b)=(6,2)$ and precomputed points are given by $P[0,1]=P$, $P[0,2]=2^6P$, $P[0,3]=P+2^6P$, $P[1,1]=2^2P$, $P[1,2]=2^8P$, $P[1,3]=2^2P+2^8P$, and $P[2,1]=P^4$, $P[2,2]=2^{10}P$, $P[2,3]=2^4P+2^{10}P$. Indexes of k are given by $k_{0,0}=0$, $k_{0,1}=2$, $k_{1,0}=0$, $k_{1,1}=1$, $k_{2,0}=0$, and $k_{2,1}=2$. Then, kP is computed by

$$T=P[0,2]+P[1,1]+P[2,2],$$
$$T=2T+P[0,0]+P[1,0]+P[2,0].$$

Algorithm 1 can be applied to two divisions $(h_1,v_1)\times(h_2,v_2)$. Here we describe the division procedure of k faithfully from the original as seen in **Fig. 2**. Let $(h_1,v_1)\times(h_2,v_2)$ be a division of n with $h_1v_1<h_2v_2$. Then, an n-bit k is divided into two blocks of $v_1\times h_1$ block with b_1 bits and $v_2\times h_2$ block with b_2 bits as follows.

$$\begin{aligned} b_2 &= \lceil\tfrac{n}{v_1h_1+v_2h_2}\rceil, & a_2 &= b_2v_2,\\ b_1 &= \lceil\tfrac{n-b_2v_2h_2}{h_1v_1}\rceil, & a_1 &= b_1v_1. \end{aligned} \quad (1)$$

The pre-computed points, $\{P_1[i_1,s_1],P_2[i_2,s_2]\}$ $((i_1,s_1)\in\{0,\cdots,v_1-1\}\times$

© 2008 Information Processing Society of Japan

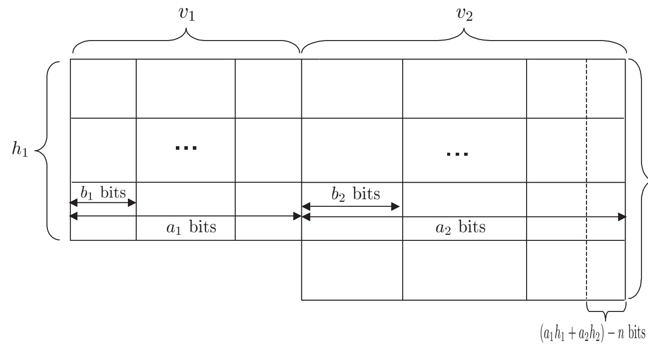

Fig. 2 Division of n with $(h_1, v_1) \times (h_2, v_2)$.

$\{1, \cdots, 2^{h_1} - 1\}$, $(i_2, s_2) \in \{0, \cdots, v_2 - 1\} \times \{1, \cdots, 2^{h_2} - 1\}$), consist of $\big((2_1^h - 1) \times v_1 + (2_2^h - 1) \times v_2\big)$ points, which are defined as

$$P_1[i_1, s_1] = \sum_{j=0}^{h_1-1} s_{1,j} 2^{b_1 i_1 + a_1 j} P,$$

$$P_2[i_2, s_2] = \sum_{j=0}^{h_2-1} s_{2,j} 2^{b_2 i_2 + a_2 j + b_1 v_1 h_1} P,$$

where $s_{1,0}, \cdots, s_{1,h_1-1}$ (resp. $s_{2,0}, \cdots, s_{2,h_2-1}$) is a representation of s_1 (resp. s_2) in radix 2, that is $s_1 = \sum_{j=0}^{h_1-1} s_{1,j} 2^j$ (resp. $s_2 = \sum_{j=0}^{h_2-1} s_{2,j} 2^j$). On the other hand, the main computation divides n-bit k into two divisions of $v_1 \times h_1$ blocks with b_1 bits and $v_2 \times h_2$ blocks with b_2 bits and, then, computes kP by repeating $v_1 + v_2$ additions to pre-computed points $P_1[i_1, s_1]$ or $P_2[i_2, s_2]$ and 1 doubling. If $(a_1h_1 + a_2h_2) - n$ bits are empty or $b_2 > b_1$, then 1 addition is saved for the first $n - (a_1h_1 + a_2h_2)$ bits in the second division or v_1 additions are saved for the first $b_2 - b_1$ bits in the first division. The detailed algorithm with 2 divisions $(h_1, v_1) \times (h_2, v_2)$ with $v_1 + v_2 \geq 2$ is given as follows.

Algorithm 2

`Input:` $k = \sum_{i=0}^{n-1} k[i]2^i$, P, $\{P_1[i,j], P_2[i,j]\}$

`Output:` kP

0. $k_{1,i,j} = \sum_{\ell=0}^{h_1-1} k[a_1\ell + j + b_1 i]2^\ell((i,j) \in \{0, \cdots, v_1 - 1\} \times \{0, \cdots, b_1 - 1\})$,
   $k_{2,i,j} = \sum_{\ell=0}^{h_2-1} k[a_2\ell + j + b_2 i + b_1 v_1 h_1]2^\ell, ((i,j) \in \{0, \cdots, v_2 - 1\} \times \{0, \cdots, b_2 - 1\})$.
1. $j = b_2 - 1$
2. `If` $b_2 > b_1$, `then` $T = \sum_{i=0}^{v_2-1} P_2[i, k_{2,i,j}]$.
   `Else if` $b_1 = b_2$ `and` $(a_1h_1 + a_2h_2) > n$, `then`
   $T = \sum_{i=0}^{v_1-1} P_1[i, k_{1,i,j}] + \sum_{i=0}^{v_2-2} P_2[i, k_{2,i,j}]$.
   `Else, then` $T = \sum_{i=0}^{v_1-1} P_1[i, k_{1,i,j}] + \sum_{i=0}^{v_2-1} P_2[i, k_{2,i,j}]$.
3. `For` $j = b_2 - 2$ `to` 0 `by` -1 `{`
4. `If` $j \geq b_1$, `then` $T = 2T + \sum_{i=0}^{v_2-1} P_2[i, k_{2,i,b_2-1}]$.
   `Else if` $j \geq b_2 - 1 - (a_1h_1 + a_2h_2 - n)$,
   `then` $T = 2T + \sum_{i=0}^{v_1-1} P_1[i, k_{1,i,b_1-1}] + \sum_{i=0}^{v_2-2} P_2[i, k_{2,i,b_2-1}]$.
   `Else, then` $T = 2T + \sum_{i=0}^{v_1-1} P_1[i, k_{1,i,b_1-1}] + \sum_{i=0}^{v_2-1} P_2[i, k_{2,i,b_2-1}]$. `}`
5. `Output` T.

3.2 Montgomery's Trick

The computational complexity of an inverse is more expensive than that of a multiplication over $\mathbb{F}_p$. The Montgomery's trick works efficiently when several inverses are executed simultaneously, denoted by $\mathsf{Minv}[n]$ in this paper. The algorithm is briefly given as follows.

Algorithm 3 ($\mathsf{Minv}[n]$) `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` $n - 1$
3. $\quad \lambda_i = \lambda_{i-1}\alpha_i \bmod p$. $\quad I = \lambda_{n-1}^{-1} \bmod p$
4. `For` $i = n - 1$ `to` 0
5. $\quad \lambda_i = I\lambda_{i-1} \bmod p$. $\quad I = I\alpha_i \bmod p$
6. `Output` $\{\lambda_0, \cdots, \lambda_{n-1}\}$

$\mathsf{Minv}[n]$ computes n inverses with $3(n-1)$ multiplications and 1 inverse. We will apply the Montgomery trick to compute ℓ additions simultaneously in affine coordinate, which will be shown in Section 4.

3.3 Mishra-Sarkar Algorithm

Mishra and Sarkar proposed an SPA-resistant scalar multiplication kP with a

© 2008 Information Processing Society of Japan

fixed point [16]), which adopts affine coordinate, and applies Montgomery's trick to reduce the computational complexity. Their algorithm expresses k in the base 2^w such as $k = \sum_{i=0}^{t} c_i 2^{wi} = \sum_{i=0}^{t} \left(\sum_{j=0}^{w-1} a_{i,j} 2^j\right) 2^{wi}$, where $t = \lceil n/w \rceil$, $c_i \in \{0, \cdots, 2^w-1\}$, $c_i = \sum_{j=0}^{w-1} a_{i,j} 2^j$, and $a_{i,j} \in \{0,1\}$. The pre-computed table consists of $\{P[j] = 2^{jw}P | 0 \le j \le t-1\}$. The algorithm then computes $c_i 2^{wi} P = \sum_{j=0}^{w-1} a_{i,j} 2^j P[i]$ for $0 \le i \le t$ with the binary method from LSB in parallel and adds each result simultaneously to get kP. Each process requires an inverse in affine coordinate independently and simultaneously, to which Montgomery trick can be applied. However, it is rather inefficient in the paradigm of an ordinary fixed-point scalar multiplication.

4. Improved Scalar Multiplication with a Fixed Point

In this section, we improve a scalar multiplication with a fixed point by applying Montgomery's trick to the LL-algorithm and adjusting the division procedure in $(v_1, h_1) \times (v_2, h_2)$.

4.1 Algorithm Intuition

The LL-algorithm in Section 3.1 consists of two phases: the pre-computation phase and the main computation phase. The pre-computation phase is done beforehand, which gives a set of pre-computed points in affine coordinates. Only the main computation phase is done online, which repeats v or v_1+v_2 additions of pre-computed points and 1 doubling. There are two ways of executing the main computation in elliptic curves. One way adopts Jacobian/Edward coordinates and its mix coordinates. The other adopts affine coordinates in the entire algorithm. In the former case, the dominant factor is the computational complexity of mixed coordinates. In the latter case, the dominant factor is the computational complexity of affine coordinates. The performance of the LL-algorithm with 1-division in Jacobian and its mix coordinates is shown in Ref. 6), where the optimal division (h, v) is also investigated. However, neither 2-division in Jacobian and its mix coordinates nor the LL-algorithm in Edward and its mix coordinates have been investigated in any paper so far [⋆1]. Furthermore, in the case of the Weierstrass form, affine coordinates with the Montgomery trick may work more efficiently in some cases since the main computation consists of an iteration that can be fitted to parallel processing. In fact, the optimal (h, v) in affine coordinates would be different from that in Jacobian/Edward coordinates since the performance of each coordinates is different, in fact: $t(\mathcal{J}+\mathcal{A}) > t(2\mathcal{J})$ (or $t(\mathcal{E}+\mathcal{E}_\mathcal{A}) > t(2\mathcal{E})$) and $t(\mathcal{A}+\mathcal{A}) < t(2\mathcal{A})$.

⋆1 The Edward form cannot be applied into the paradigm of its affine coordinates with the Montgomery trick since the computational complexity of its affine coordinates is rather slower than that of affine coordinates in the Weierstrass form.

4.2 Our Scalar Multiplication Algorithm with 1 Division

Here we show our scalar multiplication algorithm, Algorithm 4, which optimizes Algorithm 1 in such a way that the Montgomery trick $\mathsf{Minv}[\ell]$ is applied efficiently. A sketch of the algorithm is described as follows.

(1) Use formulae of $(\mathsf{Dbl_p}, \mathsf{Dbl_I}, \mathsf{Dbl_{NI}})$ and $(\mathsf{Add_p}, \mathsf{Add_I}, \mathsf{Add_{NI}})$ as affine coordinates in Section 2, where only $\mathsf{Dbl_I}$ and $\mathsf{Add_I}$ require 1 inverse.

(2) $\mathsf{TournamentAdd}[\ell_e]$ is the summation algorithm, for input of ℓ_e points $T_1, \cdots, T_{\ell_e}$, it arranges them in the tournament structure (**Fig. 3**), and outputs the result of $T_1 + \cdots + T_{\ell_e}$, which is used at the beginning of Algorithm 4 and the end of the main loop.

(3) $\mathsf{DblAddMinv}[\ell]$, the 1-doubling-and-$(\ell-1)$-addition algorithm, for input of 1 doubling point T_0 and $2(\ell-1)$ addition points $T_1, \cdots, T_{2\ell-2}$, it uses the Montgomery trick $\mathsf{Minv}[\ell]$ to compute ℓ inverses simultaneously, and outputs $2T_0, T_1+T_2, \cdots, T_{2\ell-3}+T_{2\ell-2}$, which are used at the beginning of the main loop.

(4) $\mathsf{AddMinv}[\ell]$, the ℓ-addition algorithm, for input of 2ℓ addition points $T_1, \cdots, T_{2\ell}$, it uses the Montgomery trick $\mathsf{Minv}[\ell]$ to compute ℓ inverses

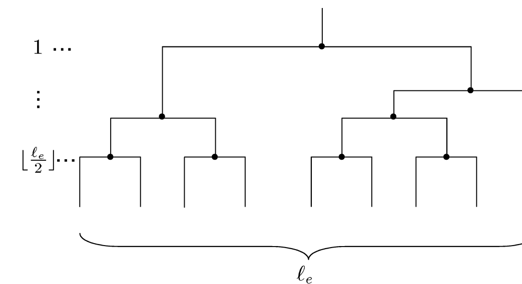

Fig. 3 Tournament structure.

© 2008 Information Processing Society of Japan

simultaneously, and outputs $T_1+T_2,\cdots,T_{2\ell-1}+T_{2\ell}$, which are used in the function of $\mathsf{TournamentAdd}[\cdot]$.

Let (h,v) be a division of n. Let us use the same notation as in Section 3.1 such as $a=\lceil \frac{n}{h} \rceil$, and $b=\lceil \frac{a}{v} \rceil$. The pre-computed points are constructed in the same way as Algorithm 1 by using the above parameters, which consist of $\big((2^h-1)\times v\big)$ points $\{P[i,s]\}$, where $P[i,s]=\sum_{j=0}^{h-1} s_j 2^{b\cdot i+a\cdot j}P$ with a representation of s in radix 2. Then, the detailed algorithm for (h,v) with $v\geq 2$ is given as follows. (see Fig. 1). Note that the main loop in Algorithm 1 corresponds to steps 6-8 in Algorithm 4.

Algorithm 4

Input: $k=\sum_{i=0}^{n-1} k[i]2^i$, P, $\{P[i,j]\}$

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. If $bv-a>0$, then $\ell_e=v-1$.
   Else $\ell_e=v$.
2. $T=\mathsf{TournamentAdd}[\ell_e](P[0,k_{0,b-1}],\cdots,P[\ell_e-1,k_{\ell_e-1,b-1}])$.
3. For $j=b-2$ to 0 by -1 {
4. If $j\geq b-(bv-a)$, then $\ell_e=v+1$.
   Else $\ell_e=v+2$.
5. $\ell=\lfloor \frac{\ell_e}{2} \rfloor$.
6. $(T_1,\cdots,T_\ell)=\mathsf{DblAddMinv}[\ell](T,P[0,k_{0,j}],\cdots,P[2(\ell-1)-1,k_{2(\ell-1)-1,j}])$.
7. If $\ell_e\equiv 1 \pmod 2$, then $T_{\ell+1}=P[2(\ell-1),k_{2(\ell-1),j}]$ and $\ell_e=\ell+1$.
   Else $\ell_e=\ell$.
8. $T=\mathsf{TournamentAdd}[\ell_e](T_1,\cdots,T_{\ell_e})$}.
9. Output T.

Algorithm 5 ($\mathsf{TournamentAdd}[\ell_e]$)

Input: $T_1,\cdots,T_{\ell_e}$

Output: $T_1+\cdots+T_{\ell_e}$

1. While $\ell_e\geq 4$ {
2. $\ell=\lfloor \frac{\ell_e}{2} \rfloor$.
3. $(T_1,\cdots,T_\ell)=\mathsf{AddMinv}[\ell](T_1,\cdots,T_{2\ell})$.
4. If $\ell_e\equiv 1 \pmod 2$, then $T_{\ell+1}=T_{2l+1}$ and $\ell_e=\ell+1$.
   Else $\ell_e=\ell$. }.
5. If $\ell_e=3$, then $T=\mathsf{Add}(\mathsf{Add}(T_1,T_2),T_3)$.
6. If $\ell_e=2$, then $T=\mathsf{Add}(T_1,T_2)$.
7. If $\ell_e=1$, then $T=T_1$.
8. Output T.

Algorithm 6 ($\mathsf{DblAddMinv}[\ell]$)

Input: $T_0,T_1,\cdots,T_{2(\ell-1)}$

Output: $2T_0,T_1+T_2,\cdots,T_{2\ell-3}+T_{2\ell-2}$

1. Compute $(\lambda_0,\lambda_1,\cdots,\lambda_{\ell-1})=\mathsf{Minv}[\ell](\mathsf{Dbl_p}(T_0),$
   $\mathsf{Add_p}(T_1,T_2),\cdots,\mathsf{Add_p}(T_{2\ell-3},T_{2\ell-2}))$
2. Compute $T_0=\mathsf{Dbl_{NI}}(T_0,\lambda_0)$ and $T_i=\mathsf{Add_{NI}}(T_{2i-1},T_{2i},\lambda_i)$ $(1\leq\forall i\leq\ell-1)$.
3. Output $\{T_0,T_1,\cdots,T_{\ell-1}\}$.

Algorithm 7 ($\mathsf{AddMinv}[\ell]$)

Input: $T_1,\cdots,T_{2\ell}$

Output: $T_1+T_2,\cdots,T_{2\ell-1}+T_{2\ell}$

1. Compute $(\lambda_1,\cdots,\lambda_\ell)=\mathsf{Minv}[\ell](\mathsf{Add_p}(T_1,T_2),\cdots,\mathsf{Add_p}(T_{2\ell-1},T_{2\ell}))$
2. Compute $T_i=\mathsf{Add_{NI}}(T_{2i-1},T_{2i},\lambda_i)$ for $i\in\{1,\cdots,\ell\}$.
3. Output $\{T_1,\cdots,T_\ell\}$.

4.3 Our Scalar Multiplication Algorithm with 2 Divisions

Now we apply the idea of Algorithm 4 to Algorithm 2 and revisit the division procedure carefully. Let $(h_1,v_1)\times(h_2,v_2)$ be a division of n with $h_1v_1<h_2v_2$. Then, an n-bit k is divided into two blocks of $v_1\times h_1$ block with b_1 bits and $v_2\times h_2$ block with b_2 bits in the optimal division between Eq. (1) and Eq. (2).

$$\begin{aligned} b_1 &= \lceil \tfrac{n}{v_1h_1+v_2h_2} \rceil, \quad a_1 = b_1v_1, \\ a_2 &= \lceil \tfrac{n-b_1v_1h_1}{h_2} \rceil, \quad b_2 = \lceil \tfrac{a_2}{v_2} \rceil. \end{aligned} \tag{2}$$

Figure 4 shows the division procedure of Eq. (2). This division procedure is said to be *optimal* for n when the redundant part of computation, $|\ a_1h_1+a_2h_2-n\ |$, is minimal. The redundant part of computation results from $\lceil \frac{n-b_2v_2h_2}{v_1h_1} \rceil$ or $\lceil \frac{n-b_1v_1h_1}{h_2} \rceil$ and $\lceil \frac{a_2}{v_2} \rceil$ in the case of Eq. (1) or Eq. (2), respectively. Therefore, the redundant part in Eq. (2) seems to be less than or equal to that in Eq. (1). In fact, in our experimental results, the division procedure of Eq. (2) was optimal.

Then, the pre-computed points are constructed in the same way as in Algorithm 2 by using the above parameters, which consist

© 2008 Information Processing Society of Japan

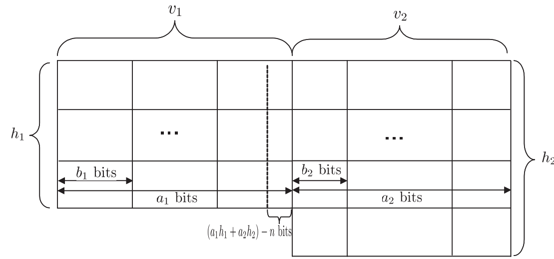

Fig. 4 Revised division of n with $(h_1, v_1) \times (h_2, v_2)$.

of $\left((2^{h_1}-1)\times v_1 + (2^{h_2}-1)\times v_2\right)$ points $\{P_1[i_1,s_1], P_2[i_2,s_2]\}$ $((i_1,s_1) \in \{0,\cdots,v_1-1\}\times\{1,\cdots,2^{h_1}-1\}$, $(i_2,s_2)\in\{0,\cdots,v_2-1\}\times\{1,\cdots,2^{h_2}-1\})$. The detailed algorithm in 2 divisions $(h_1,v_1)\times(h_2,v_2)$ with $v_1+v_2\geq 2$ is given as follows. Note that step 4 in Algorithm 2 corresponds to steps 4-6 in Algorithm 8 and that $b_1 \geq b_2$ (resp. $b_2 \geq b_1$) holds in our algorithm (resp. Algorithm 8).

Algorithm 8

Input: $k=\sum_{i=0}^{n-1} k[i]2^i$, P, $\{P_1[i,j], P_2[i,j]\}$

Output: kP

0. $k_{1,i,j}=\sum_{\ell=0}^{h_1-1} k[a_1\ell+j+b_1 i]2^{\ell}((i,j)\in\{0,\cdots,v_1-1\}\times\{0,\cdots,b_1-1\})$,
   $k_{2,i,j}=\sum_{\ell=0}^{h_2-1} k[a_2\ell+j+b_2 i+b_1v_1h_1]2^{\ell}$, $((i,j)\in\{0,\cdots,v_2-1\}\times\{0,\cdots,b_2-1\})$.
1. $j=b_1-1$.
2. If $b_1>b_2$, then $\ell_e=v_1$.
   $T=\mathsf{TournamentAdd}[\ell_e](P_1[0,k_{1,0,j}],\cdots,P_1[v_1-1,k_{1,v_1-1,j}])$.
   Else if $b_1=b_2$ and $a_1h_1+a_2h_2>n$, then $\ell_e=v_1+v_2-1$ and
   $T=\mathsf{TournamentAdd}[\ell_e](P_1[0,k_{1,0,j}],\cdots,P_1[v_1-1,k_{1,v_1-1,j}],P_2[0,k_{2,0,j}],\cdots,$
   $P_2[v_2-2,k_{2,v_2-2,j}])$.
   Else, then $\ell_e=v_1+v_2$ and
   $T=\mathsf{TournamentAdd}[\ell_e](P_1[0,k_{1,0,j}],\cdots,P_1[v_1-1,k_{1,v_1-1,j}],P_2[0,k_{2,0,j}],\cdots,$
   $P_2[v_2-1,k_{2,v_2-1,j}])$.
3. For $j=b_1-2$ to 0 by -1 {
4. If $j\geq b_2$, then $\ell_e=v_1+2$. $\ell=\lfloor\frac{\ell_e}{2}\rfloor$, and
   $(T_1,\cdots,T_\ell)=\mathsf{DblAddMinv}[\ell](T,P_1[0,k_{1,0,j}],\cdots,P_1[2(\ell-1)-1,k_{1,2(\ell-1)-1,j}])$.
   Else If $j\geq b_2-1-(a_1h_1+a_2h_2-n)$, then $\ell_e=v_1+v_2+1$, $\ell=\lfloor\frac{\ell_e}{2}\rfloor$, and
   $(T_1,\cdots,T_\ell)=\mathsf{DblAddMinv}[\ell](T,P_1[0,k_{1,0,j}],\cdots,$
   $P_1[v_1-1,k_{1,v_1-1,j}],P_2[0,k_{2,0,j}],\cdots,P_2[2(\ell-1)-v_1,k_{2,2(\ell-1)-v_1,j}])$.
   Else, $\ell_e=v_1+v_2+2$, $\ell=\lfloor\frac{\ell_e}{2}\rfloor$, and
   $(T_1,\cdots,T_\ell)=\mathsf{DblAddMinv}[\ell](T,P_1[0,k_{1,0,j}],\cdots,$
   $P_1[v_1-1,k_{1,v_1-1,j}],P_2[0,k_{2,0,j}],\cdots,P_2[2(\ell-1)-v_1,k_{2,2(\ell-1)-v_1,j}])$.
5. If $\ell_e\equiv 0 \pmod 2$, then $\ell_e=\ell$.
   Else if $j\geq b_2$, then $T_{\ell+1}=P_1[2(\ell-1),k_{1,2(\ell-1),j}]$ and $\ell_e=\ell+1$.
   Else, then $T_{\ell+1}=P_2[2(\ell-1)-v_1+1,k_{2,2(\ell-1)-v_1+1,j}]$ and $\ell_e=\ell+1$.
6. $T=\mathsf{TournamentAdd}[\ell_e](T_1,\cdots,T_{\ell_e})$.}
7. Output T.

4.4 Performance

This section shows the computational complexity of Algorithms 4 and 8 theoretically after investigating the computational complexity for the computation of all inverses in the summation algorithm, $\mathsf{TournamentAdd}[\ell_e]$, in affine coordinates[⋆1]. The computational complexity, denoted by $\mathit{Inv}[\ell_e]$ in this paper, is shown in the next theorem.

Theorem 1 Let $\mathit{Inv}[\ell_e]$ be the computational complexity for the computation of inverses in the ℓ_e-point-summation algorithm, $\mathsf{TournamentAdd}[\ell_e]$ in affine coordinates. Then the following holds:

$$(1)\quad \mathit{Inv}[\ell_e]=\begin{cases}0 & (\text{if } \ell_e=1)\\ 3(\ell_e-t-1)M+tI & (\text{if } \ell_e=2^t)\\ 3(\ell_e-t-2)M+(t+1)I & (\text{if } 2^t<\ell_e<2^{t+1})\end{cases}$$

$$(2)\quad \mathit{Inv}[\ell_e+1]=\mathit{Inv}[\ell_e]+3M,\ \text{if } 2^t<\ell_e<\ell_e+1\leq 2^{t+1},$$

where M (resp. I) represents the computational complexity of the modular multiplication (resp. inverse) on the definition field.

proof: Regarding (1), it follows from the fact that the Montgomery's trick $\mathsf{Minv}[n]$ computes ℓ_e inverses with $I+3(\ell_e-1)M$. As for (2), it immediately follows from (1). ∎

By using Theorem 1, the total computational complexity Comp_1 of Algorithm 4 with (h,v) $(v\geq 2)$ can be computed as follows:

⋆1 The number of input points for doubling is doubled. That is, the number of input points of summation of $2T_0+T_1+\cdots+T_{\ell_e-2}$ is ℓ_e.

© 2008 Information Processing Society of Japan

Theorem 2 Let $Comp_1$ be the total computational complexity of Algorithm 4 with (h, v). Then $Comp_1$ can be computed as follows:

$$Comp_1 = \begin{cases} (b-1)D_{ni} + (a-1)A_{ni} + Inv[v-1] + (r-1)Inv[v+1] \\ \quad +(b - bv + a)Inv[v+2] \qquad\qquad \text{(if } r > 0) \\ (b-1)D_{ni} + (a-1)A_{ni} + (b-1)Inv[v+2] + Inv[v] \\ \qquad\qquad\qquad\qquad\qquad\qquad \text{(if } r = 0), \end{cases}$$

where $a = \lceil \frac{n}{h} \rceil$, $b = \lceil \frac{a}{v} \rceil$, $r = bv - a$, A_{ni} (resp. D_{ni}) represents the total computational complexity of $\mathsf{Add_p}$ and $\mathsf{Add_{NI}}$ (resp. $\mathsf{Dbl_p}$ and $\mathsf{Dbl_{NI}}$).

proof: The proof easily follows from Algorithm 4. ∎

From Theorem 1, the computational complexity of $Inv[\ell_e]$ is most efficient when ℓ_e is a power of 2. Therefore, the cases of $v = 2, 6, 14, 30, \cdots$ would yield efficient performance of Algorithm 4.

The total computational complexity $Comp_2$ of Algorithm 8 with $(h_1, v_1) \times (h_2, v_2)$ $(v_1 + v_2 \geq 2)$ is given by the following theorem.

Theorem 3 Let $Comp_2$ be the total computational complexity of Algorithm 8 with $(h_1, v_1) \times (h_2, v_2)$. Then $Comp_2$ can be computed as follows:

$$Comp_2 = \begin{cases} (b_1-1)D_{ni}+(a_1+a_2-1)A_{ni}+(b_2-r_2)Inv[v_1+v_2+2]+Inv[v_1] \\ \quad +(b_1 - b_2 - 1)Inv[v_1+2] + r_2 Inv[v_1+v_2+1] \\ \qquad\qquad\qquad\qquad \text{(if } b_1 > b_2) \\ (b_1-1)D_{ni}+(a_1+a_2-1)A_{ni}+(b_1-r_2)Inv[v_1+v_2+2] \\ \quad +Inv[v_1+v_2-1] + (r_2-1)Inv[v_1+v_2+1] \\ \qquad\qquad\qquad\qquad \text{(if } b_1 = b_2 \text{ and } r_2 > 0) \\ (b_1-1)D_{ni}+(a_1+a_2-1)A_{ni}+(b_1-1)Inv[v_1+v_2+2]+Inv[v_1+v_2] \\ \qquad\qquad\qquad\qquad \text{(if } b_1 = b_2 \text{ and } r_2 = 0) \end{cases}$$

where $b_1 = \lceil \frac{n}{v_1 h_1 + v_2 h_2} \rceil$, $a_1 = b_1 v_1$, $a_2 = \lceil \frac{n - b_1 v_1 h_1}{h_2} \rceil$, $b_2 = \lceil \frac{a_2}{v_2} \rceil$, $r_2 = b_2 v_2 - a_2$, and A_{ni} (resp. D_{ni}) represents the total computational complexity of $\mathsf{Add_p}$ and $\mathsf{Add_{NI}}$ (resp. $\mathsf{Dbl_p}$ and $\mathsf{Dbl_{NI}}$).

proof: The proof easily follows from Algorithm 8. ∎

For the same reason as $Comp_1$, the cases where $v_1 + v_2 + 2$ is a power of 2 would yield an efficient performance of Algorithm 8. That is, $v_1 + v_2 = 2, 6, 14, 30, \cdots$.

Table 3 Performance of Algorithms 4 and 8 (elliptic curve).

division (h,v), $(h_1,v_1) \times (h_2,v_2)$	Computational complexity		# points
$(1,2)$	$713M + 317S + 159I$	$(2715.6M)$	2
$(1,3)$	$583M + 265S + 159I$	$(2544M)$	3
$(2,1) \times (1,1)$	$469M + 211S + 106I$	$(1803.8M)$	4
$(2,2)$	$353M + 157S + 79I$	$(1347.6M)$	6
$(2,3)$	$288M + 131S + 79I$	$(1261.8M)$	9
$(3,1) \times (2,1)$	$281M + 125S + 63I$	$(1074.0M)$	10
$(2,4)$	$313M + 117S + 59I$	$(1055.6M)$	12
$(3,2)$	$236M + 105S + 53I$	$(903M)$	14
$(2,6)$	$334M + 105S + 42I$	$(880M)$	18
$(3,3)$	$191M + 87S + 53I$	$(843.6M)$	21
$(4,1) \times (3,1)$	$200M + 89S + 45I$	$(766.2M)$	22
$(3,2) \times (2,4)$	$288M + 89S + 34I$	$(733.2M)$	26
$(3,4)$	$207M + 79S + 41I$	$(721.2M)$	28
$(4,2)$	$173M + 77S + 39I$	$(663.6M)$	30
$(2,2) \times (3,4)$	$250M + 77S + 30I$	$(641.6M)$	34
$(3,5)$	$219M + 73S + 32I$	$(629.4M)$	35
$(2,1) \times (3,5)$	$235M + 74S + 30I$	$(624.2M)$	38
$(3,6)$	$224M + 69S + 27I$	$(576.2M)$	42
$(4,1) \times (3,5)$	$209M + 66S + 27I$	$(558.8M)$	50
$(4,2) \times (3,4)$	$198M + 61S + 24I$	$(510.8M)$	58
$(4,3) \times (3,3)$	$188M + 59S + 24I$	$(499.2M)$	66
$(3,2) \times (4,4)$	$184M + 57S + 22I$	$(471.6M)$	74
$(4,5)$	$158M + 53S + 24I$	$(464.4M)$	75
$(3,1) \times (4,5)$	$172M + 53S + 21I$	$(445.4M)$	82
$(4,6)$	$162M + 51S + 21I$	$(433.8M)$	90
$(5,1) \times (4,5)$	$157M + 50S + 21I$	$(428M)$	106
$(4,8)$	$158M + 47S + 19I$	$(404.6M)$	120
$(5,2) \times (4,4)$	$153M + 48S + 19I$	$(400.4M)$	122
$(4,3) \times (5,3)$	$146M + 45S + 18I$	$(380M)$	138
$(4,10)$	$162M + 45S + 16I$	$(374M)$	150
$(5,4) \times (4,2)$	$139M + 43S + 17I$	$(360.4M)$	154
$(5,6)$	$126M + 41S + 18I$	$(356.8M)$	186
$(3,2) \times (4,12)$	$179M + 45S + 12I$	$(347M)$	194
$(4,13) \times (2,1)$	$174M + 44S + 12I$	$(341.2M)$	198
$(4,14)$	$169M + 43S + 12I$	$(335.4M)$	210
$(4,15)$	$175M + 43S + 10I$	$(319.4M)$	225
$(5,4) \times (4,10)$	$154M + 40S + 12I$	$(318.0M)$	274
$(4,20)$	$170M + 41S + 10I$	$(312.8M)$	300
$(5,6) \times (4,8)$	$149M + 39S + 12I$	$(312.2M)$	306
$(5,7) \times (4,7)$	$147M + 38S + 11I$	$(298.4M)$	322
$(5,8) \times (4,6)$	$142M + 37S + 11I$	$(292.6M)$	338
$(5,11)$	$129M + 35S + 12I$	$(289.0M)$	341
$(5,15)$	$159M + 35S + 2I$	$(209.0M)$	465

© 2008 Information Processing Society of Japan

Table 4 Performance of Algorithms 4 and 8 (hyper-elliptic curve).

division (h, v), $(h_1, v_1) \times (h_2, v_2)$	Computational complexity		# points
$(1, 2)$	$5473M_{hec} + 872S_{hec} + 159I_{hec}$	$(1979.9M)$	2
$(1, 3)$	$4823M_{hec} + 742S_{hec} + 159I_{hec}$	$(1791.4M)$	3
$(2, 1) \times (1, 1)$	$3629M_{hec} + 580S_{hec} + 106I_{hec}$	$(1314.8M)$	4
$(2, 2)$	$2713M_{hec} + 432S_{hec} + 79I_{hec}$	$(981.9M)$	6
$(2, 3)$	$2388M_{hec} + 367S_{hec} + 79I_{hec}$	$(887.65M)$	9
$(3, 1) \times (2, 1)$	$2161M_{hec} + 344S_{hec} + 63I_{hec}$	$(782.3M)$	10
$(3, 2)$	$1816M_{hec} + 289S_{hec} + 53I_{hec}$	$(657.55M)$	14
$(4, 1) \times (2, 1)$	$1791M_{hec} + 286S_{hec} + 52I_{hec}$	$(647.95M)$	18
$(3, 3)$	$1591M_{hec} + 244S_{hec} + 53I_{hec}$	$(592.3M)$	21
$(4, 1) \times (3, 1)$	$1540M_{hec} + 245S_{hec} + 45I_{hec}$	$(557.75M)$	22
$(3, 4)$	$1527M_{hec} + 224S_{hec} + 41I_{hec}$	$(539.3M)$	28
$(4, 2)$	$1333M_{hec} + 212S_{hec} + 39I_{hec}$	$(482.9M)$	30
$(3, 6)$	$1444M_{hec} + 199S_{hec} + 27I_{hec}$	$(475.05M)$	42
$(4, 3)$	$1183M_{hec} + 182S_{hec} + 39I_{hec}$	$(439.4M)$	45
$(5, 1) \times (4, 1)$	$1195M_{hec} + 190S_{hec} + 35I_{hec}$	$(433M)$	46
$(4, 2) \times (3, 4)$	$1278M_{hec} + 176S_{hec} + 24I_{hec}$	$(420.7M)$	58
$(4, 4)$	$1113M_{hec} + 162S_{hec} + 29I_{hec}$	$(390.4M)$	60
$(5, 2)$	$1057M_{hec} + 168S_{hec} + 31I_{hec}$	$(383.1M)$	62
$(4, 5)$	$1078M_{hec} + 152S_{hec} + 24I_{hec}$	$(365.9M)$	75
$(4, 6)$	$1062M_{hec} + 147S_{hec} + 21I_{hec}$	$(352.65M)$	90
$(5, 3)$	$932M_{hec} + 143S_{hec} + 31I_{hec}$	$(346.9M)$	93
$(5, 1) \times (4, 5)$	$1037M_{hec} + 144S_{hec} + 21I_{hec}$	$(345.8M)$	106
$(4, 8)$	$1018M_{hec} + 137S_{hec} + 19I_{hec}$	$(334.15M)$	120
$(5, 2) \times (4, 4)$	$993M_{hec} + 138S_{hec} + 19I_{hec}$	$(328.1M)$	122
$(5, 4)$	$881M_{hec} + 128S_{hec} + 23I_{hec}$	$(309.1M)$	124
$(4, 2) \times (5, 4)$	$921M_{hec} + 127S_{hec} + 18I_{hec}$	$(305.15M)$	154
$(5, 5)$	$865M_{hec} + 123S_{hec} + 20I_{hec}$	$(295.9M)$	155
$(4, 1) \times (5, 5)$	$896M_{hec} + 124S_{hec} + 18I_{hec}$	$(298.3M)$	170
$(5, 6)$	$846M_{hec} + 118S_{hec} + 18I_{hec}$	$(284.6M)$	186
$(5, 7)$	$824M_{hec} + 113S_{hec} + 17I_{hec}$	$(275.4M)$	217
$(5, 8)$	$805M_{hec} + 108S_{hec} + 15I_{hec}$	$(264.1M)$	248
$(5, 8) \times (4, 6)$	$842M_{hec} + 109S_{hec} + 11I_{hec}$	$(262.6M)$	338
$(5, 11)$	$789M_{hec} + 103S_{hec} + 12I_{hec}$	$(250.9M)$	341
$(5, 15)$	$819M_{hec} + 103S_{hec} + 2I_{hec}$	$(230.9M)$	465

Tables 3 and **4** show the performances of various cases of Algorithms 4 and 8 in the case of an elliptic curve over a 160-bit definition field and a hyper-elliptic curve with genus 2 over an 80-bit definition field, respectively. Let M (resp. S, resp. I) represent a modular multiplication (resp. squaring, resp. inverse) on the 160-bit field. Let M_{hec} (resp. S_{hec}, resp. I_{hec}) represent a modular multiplication (resp. squaring, resp. inverse) on the 80-bit field.

In order to make the comparison easier, the computational amount is also estimated in terms of M by assuming that $S = 0.8M$, $I = 11M$, $S_{hec} = 0.8M_{hec}$, $I = 11M_{hec}$, and $M = 4M_{hec}$. A ratio of inversion to multiplication in the case of a prime field $\mathbb{F}_p$ can be estimated between 3 and 10 as described in Section 2.3, however, we adopt an even bigger ratio of 11 since our algorithm is efficient when the ratio is small.

5. Comparison

In this section, we compare performances of our algorithms 4 and 8 with those of the previous algorithm [7),14)] in each case of an elliptic curve and a hyper-elliptic curve. In the case of an elliptic curve, the previous algorithm uses the mixed coordinate, that is, points of the pre-computed table are given as affine (resp. Edward affine) coordinate and the main computation is done in Jacobian (resp. Edward) coordinate. In the case of a hyper-elliptic curve, the previous algorithm uses affine coordinate without using Montgomery's method. **Tables 5** and **6** show divisions that our algorithms are better than the previous algorithm and their performances in the cases of an elliptic curve and a hyper-elliptic curve, respectively. Both Tables present the computational complexity based on the number of modular multiplications, squares, and inverses over the definition field as well as the estimation based on the assumption that $S = 0.8M$, $I = 11M$, $S_{hec} = 0.8M_{hec}$, $I = 11M_{hec}$, and $M = 4M_{hec}$. The reduction ratio of our algorithm to the previous algorithm is computed under the estimation. The break-even point shows I/M or I_{hec}/M_{hec} when the computational complexity of our algorithms is equal to that of previous ones, where $S = 0.8M$ is assumed.

In the case of an elliptic curve, we see that our algorithm with a pre-computed table of 18, 26, 34, 42, or 50 can compute kP more efficiently than the previous algorithm with Jacobian coordinate (resp. Edward coordinate) if $I/M < 12.2, 12.7, 12.7, 12.8$, or 12.3 (resp. $12.6, 13.2, 13.2, 13.3$, or 12.7). In the case of $I = 11M$, the computational complexity of our algorithm with a pre-computed table of 18, 26, 34, 42, or 50 can be reduced to 95%, 93%, 93%, 93%, or 94% (resp 93%, 91%, 91%, 91%, or 93%) of that of the previous algorithm with Jacobian (resp. Edward) coordinate. Our experimental results also indicate that affine coordinate with Montgomery's trick or mixed coordinate with Jacobian

© 2008 Information Processing Society of Japan

Table 5 Comparison of performance (elliptic curve).

(h, v) $(h_1, v_1) \times (h_2, v_2)$	# points	Computational complexity		BE-point†
		Ours	Previous results ($\mathcal{J}/\mathcal{E}$)	reduction ratio
$(2, 6)$	18	$334M + 105S + 42I$	($\mathcal{J}$) $582M + 421S + I$	12.2
		$(418M + 42I)$	$(918.8M + I)$	0.95
		$(I = 11M : 880.0M)$	($\mathcal{E}$) $831M + 131S + I$	12.6
			$(935.8M + I)$	0.93
$(3, 2) \times (2, 4)$	26	$288M + 89S + 34I$	($\mathcal{J}$) $494M + 357S + I$	12.7
		$359.2M + 34I$	$(779.6M + I)$	0.93
		$(I = 11M : 733.2M)$	($\mathcal{E}$) $705M + 111S + I$	13.2
			$(793.8M + I)$	0.91
$(2, 2) \times (3, 4)$	34	$250M + 77S + 30I$	($\mathcal{J}$) $434M + 309S + I$	12.7
		$311.6M + 30I$	$(681.2M + I)$	0.93
		$(I = 11M : 641.6M)$	($\mathcal{E}$) $619M + 95S + I$	13.2
			$(695M + I)$	0.91
$(3, 6)$	42	$224M + 69S + 27I$	($\mathcal{J}$) $390M + 277S + I$	12.8
		$279.2M + 27I$	$(611.6M + I)$	0.93
		$(I = 11M : 576.2M)$	($\mathcal{E}$) $556M + 85S + I$	13.3
			$(624M + I)$	0.91
$(4, 1) \times (3, 5)$	50	$209M + 66S + 27I$	($\mathcal{J}$) $369M + 265S + I$	12.3
		$261.8M + 27I$	$(581M + I)$	0.94
		$(I = 11M : 558.8M)$	($\mathcal{E}$) $526M + 82S + I$	12.7
			$(591.6M + I)$	0.93

†: the break-even point is the value of I/M for which our result becomes the same efficiency as the previous result.

Table 6 Comparison of Performance (hyper-elliptic curve).

(h, v) $(h_1, v_1) \times (h_2, v_2)$ # points	Computational complexity		BE-point† reduction ratio
	Ours	Previous results	
$(4, 1) \times (2, 1)$	$1791M_{hec} + 286S_{hec} + 52I_{hec}$	$1716M_{hec} + 286S_{hec} + 78I_{hec}$	2.9
18	$(648.0M)$	$(700.7M)$	0.92
$(3, 4)$	$1527M_{hec} + 224S_{hec} + 41I_{hec}$	$1452M_{hec} + 224S_{hec} + 66I_{hec}$	3
28	$(539.3M)$	$(589.3M)$	0.92
$(4, 2)$	$1333M_{hec} + 212S_{hec} + 39I_{hec}$	$1276M_{hec} + 212S_{hec} + 58I_{hec}$	3
30	$(482.9M)$	$(520.9M)$	0.93
$(3, 6)$	$1444M_{hec} + 199S_{hec} + 27I_{hec}$	$1342M_{hec} + 199S_{hec} + 61I_{hec}$	3
42	$(475.1M)$	$(543.1M)$	0.87
$(4, 3)$	$1183M_{hec} + 182S_{hec} + 39I_{hec}$	$1144M_{hec} + 182S_{hec} + 52I_{hec}$	3
46	$(433.0M)$	$(467.0M)$	0.93
$(4, 2) \times (3, 4)$	$1278M_{hec} + 176S_{hec} + 24I_{hec}$	$1188M_{hec} + 176S_{hec} + 54I_{hec}$	3
58	$(420.7M)$	$(480.7M)$	0.88

†: the break-even point is the value of I_{hec}/M_{hec} for which our result becomes the same efficiency as the previous result.

coordinate are better than mixed coordinate with Edward coordinate.

In the case of a hyper-elliptic curve, we see that our algorithm with a pre-computed table of 18 (resp. 28, 30, 42, 46, 58) can compute kP more efficiently than the previous algorithm if $I_{hec}/M_{hec} > 2.9$ (resp. 3). That is, our algorithm is usually more efficiently than the previous one. In the case of $I_{hec} = 11M_{hec}$, the computational complexity of our algorithm with a pre-computed table of 18, 28, 30, 42, 46, or 58 can be reduced to 92%, 92%, 93%, 87%, 93%, or 88% of the previous algorithm, respectively. We note that hyper-elliptic curves in our algorithm give a better performance than elliptic curves in any size of a pre-computed table.

6. Conclusion

In this paper, we have explored an efficient scalar multiplication with a fixed point and improved the LL-algorithm by applying affine coordinate to it and adjusting the division procedure. We have also investigated the previous algorithm with Edward coordinate as well as Jacobian coordinate. Our algorithm can improve the computational complexity of the previous algorithm with the best coordinate in some sizes of the pre-computed table. We have also given the formulae of the computational complexity of the proposed algorithm with any division theoretically, which helps developers to choose the best division suitable for the storage available.

Acknowledgments The authors express their gratitude to anonymous referees for invaluable comments.

References

1) Avanzi, R.M.: On multi-exponentiation in cryptography, Cryptology ePrint Archive, Report 2002/154 (2002). Available at http://eprint.iacr.org/2002/154/
2) Blake, I.F., Seroussi, G. and Smart, N.P.: *Elliptic Curves in Cryptology*, LMS, Vol.265, Cambridge University Press (1999).
3) Brickell, E.F., Gordon, D.M., McCurley, K.S. and Wilson, D.B.: Fast exponentiation with preocomputation, *Advances in Cryptology—Proc. EUROCRYPT'92*, Lecture Notes in Computer Science, Vol.658, pp.200–207, Springer-Verlag (1993).
4) Bernstein, D.J. and Lange, T.: Fast addition and doubling on elliptic curves, *Advances in Cryptology-Proc. ASIACRYPT'07*, Vol.4833, pp.29–50 (2007).
5) Ciet, M., Joye, M., Lauter, K. and Montgomey, P.L.: Trading inversions for multi-

© 2008 Information Processing Society of Japan

plications in elliptic curve cryptography, *Designs, Codes and Cryptography*, Vol.39, No.2, pp.189–206, Springer Netherlands (2006).
6) Cohen, H., Miyaji, A. and Ono, T.: Efficient elliptic curve exponentiation, *Proc. Information and communications security, ICICS'97*, Lecture Notes in Computer Science, Vol.1334, pp.282–290, Springer-Verlag (1997).
7) Cohen, H., Miyaji, A. and Ono, T.: Efficient elliptic curve exponentiation using mixed coordinates, *Advances in Cryptology—Proc. ASIACRYPT'98*, Lecture Notes in Computer Science, Vol.1514, pp.51–65, Springer-Verlag (1998).
8) Doche, C., Icart, T. and Kohel, D.R.: Efficient scalar multiplication by isogeny decompositions, *Proc. PKC2006*, LNCS, Vol.3958, pp.191–206 (2006).
9) Eisenträger, K., Lauter, K. and Montgomey, P.L.: Fast elliptic curve arithmetic and improved Weil pairing evaluation, *Proc. CT-RSA2003*, LNCS, Vol.2612, pp.343–354, Springer-Verlag (2003).
10) Edwards, H.M.: A normal form for ellipticcurves, *Bulletin of the American Mathematical Society*, Vol.44, pp.393–422 (2007).
11) Kaliski Jr., B.S.: The Montgomery inverse and its applications, *IEEE Trans. Computers*, Vol.44, pp.1064–1065 (1995).
12) Koç, Ç.K. and Savaş, E.: Architectures for unified field inversion with applications in elliptic curve cryptography, *9th IEEE Interntional Conference on Electronics, Circuits and Systems, ICECS 2002*, Vol.3, pp.1155–1158 (2002).
13) Lange, T.: Formulae for Arithmetic on Genus 2 Hyperelliptic curve. Available at http://www.ruhr-uni-bochum.de/itsc/tanja/preprints.html
14) Lim, C.H. and Lee, P.J.: More flexible exponentiation with precomputation, *Advances in Cryptology-Proc. Crypto'94*, Lecture Notes in Computer Science, Vol.839, pp.95–107, Springer-Verlag (1994).
15) Menezes, A., Oorschot, P.C. and Vanstone, S.: *Handbook of applied cryptography*, CRC Press, Inc. (1996).
16) Mishra, P.K. and Sarkar, P.: Application of Montgomery's trick to Scalr Multiplication for EC and HEC Using Fixed Base Point, PKC2004, Lecture Notes in Computer Science, Vol.2947, pp.41–57 (2004).
17) Montgomery, P.L.: Speeding the Pollard and elliptic curve methods for factorization, *Mathematics of Computation*, Vol.48, pp.243–264 (1987).
18) Okeya, K., Samoa, K.S., Spahn, C. and Takagi, T.: Signed Binary Representations Revisited, *CRYPTO 2004*, Lecture Notes in Computer Science, Vol.3152, pp.123–139 (2004).
19) Standard for efficient cryptography group, specification of standards for efficient cryptography. Available from: http://www.secg.org

Appendix

A.1 Other Coordinates

This annex summarizes addition formulae in Edward coordinates and the previous addition formulae in Jacobian coordinates.

A.1.1 Edward Coordinates

Recently, a new coordinate system, called Edward Coordinates, has been proposed [10], whose comparison with other coordinate systems is investigated [4]. One of the weak points in Edward coordinates is that no prime-order elliptic curve can be represented in Edward coordinates with the same definition field since it must have a point with order 4 whereas a prime-order elliptic curve is usually recommended in standards [19]. Here we simply summarize Edward coordinates and its performance.

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

$$E_{\mathcal{E}} : (X^2 + Y^2)Z^2 = Z^4 + dX^2Y^2.$$

Then, two points (X, Y, Z) and (rX, rY, rZ) for some $r \in \mathbb{F}_p^*$ are recognized as the same point. The point at infinity, which performs the zero element, is represented with $(0, 1, 1)$. Let $P_1 = (X_1, Y_1, Z_1), P_2 = (X_2, Y_2, Z_2)$, and $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$. The doubling and addition formulae in Edward coordinates can be represented as follows.

$\mathsf{Add}^{\mathcal{E}}(P_1, P_2)$ $(P_1 \neq \pm P_2)$	$\mathsf{Dbl}^{\mathcal{E}}(P_1)$
$U_1 = Z_1Z_2$, $U_2 = X_1X_2$, $U_3 = Y_1Y_2$, $S_1 = (X_1 + Y_1)(X_2 + Y_2)$, $S_2 = dU_2U_3$, $H = U_1^2 - S_2$, $R = U_1^2 + S_2$, $X_3 = U_1H(S_1 - U_2 - U_3)$, $Y_3 = U_1R(U_3 - U_2)$, $Z_3 = HR$.	$U_1 = X_1^2$, $U_2 = Y_1^2$, $U_3 = Z_1^2$, $S_1 = (X_1 + Y_1)^2$, $S_2 = U_1 + U_2$, $H = S_2 - 2U_3$, $X_3 = (S_1 - S_2)H$, $Y_3 = S_2(U_1 - U_2)$, $Z_3 = S_2H$.

The computation times in Edward coordinates are $t(\mathcal{E} + \mathcal{E}) = 11M + S$ and $t(2\mathcal{E}) = 3M + 4S$, where $\mathcal{E}$ means Edward coordinates. In this paper, the computation time of multiplication by d is not considered as special. This is due to the fact that the limitation on d damages the flexibility of elliptic curves over a definition filed, which is exactly an advantage over RSA or DLP-based cryptosystems.

 © 2008 Information Processing Society of Japan

Furthermore, such a limitation does not seem to be used in general ⋆1.

A.1.2 Traditional Jacobian Coordinates

The traditional Jacobian coordinates [7] have been referred in many publications so far. To compare with the addition formulae in Section 2, this section describes the traditional Jacobian coordinates. Let $P_1 = (X_1, Y_1, Z_1)$, $P_2 = (X_2, Y_2, Z_2)$, and $P_3 = P_1 + P_2 = (X_3, Y_3, Z_3)$. The doubling and addition formulae in the traditional Jacobian coordinates can be represented as follows.

$\mathsf{Add}(P_1, P_2)$ $(P_1 \neq \pm P_2)$
$U_1 = X_1 Z_2^2$, $U_2 = X_2 Z_1^2$, $H = U_2 - U_1$,
$S_1 = Y_1 Z_2^3$, $S_2 = Y_2 Z_1^3$, $R = S_2 - S_1$,
$X_3 = -H^3 - 2U_1 H^2 + R^2$,
$Y_3 = -S_1 H^3 + R(U_1 H^2 - X_3)$, $Z_3 = Z_1 Z_2 H$.

$\mathsf{Dbl}(P_1)$
$S = 4X_1 Y_1^2$, $M = 3X_1^2 + aZ_1^4$,
$X_3 = M^2 - 2S$,
$Y_3 = -8Y_1^4 + M(S - X_3)$, $Z_3 = 2Y_1 Z_1$.

The computation times of addition and doubling in the traditional Jacobian coordinates are $12M + 4S$ and $4M + 6S$, respectively.

(Received December 3, 2007)

(Accepted June 3, 2008)

(Original version of this article can be found in the Journal of Information Processing Vol.16, pp.176–189.)

⋆1 The computation time in coordinates is investigated in three cases [4] $(S/M, (\text{multiplication by } d)/M) = (1,\ 1),\ (0.8,\ 0.5),\ (0.8,\ 0)$, however, the cases may not be natural since $(S/M, (\text{multiplication by } d)/M) = (0.8,\ 1)$ is usually assumed from the point of view of practicality and flexibility.

Atsuko Miyaji received the B. Sc., the M. Sc., and the Dr. Sci. degrees in mathematics from Osaka University, Osaka, Japan in 1988, 1990, and 1997 respectively. She joined Matsushita Electric Industrial Co., LTD from 1990 to 1998 and engaged in research and development for secure communication. She was an associate professor at the Japan Advanced Institute of Science and Technology (JAIST) in 1998. She has joined the computer science department of the University of California, Davis since 2002. She has been a professor at the Japan Advanced Institute of Science and Technology (JAIST) since 2007 and a director of Library of JAIST since 2008. Her research interests include the application of number theory into cryptography and information security. She received the IPSJ Sakai Special Researcher Award in 2002, the Standardization Contribution Award in 2003, Engineering Sciences Society: Certificate of Appreciation in 2005, the AWARD for the contribution to CULTURE of SECURITY in 2007, IPSJ/ITSCJ Project Editor Award in 2007, the Director-General of Industrial Science and Technology Policy and Environment Bureau Award in 2007, and Editorial Committee of Engineering Sciences Society: Certificate of Appreciation in 2007. She is a member of the International Association for Cryptologic Research, the Institute of Electronics, Information and Communication Engineers, the Information Processing Society of Japan, and the Mathematical Society of Japan.

Kenji Mizosoe received the B. Eng. degree from the National Defense Academy in 1999 and the M. Info. Sci. degree from the Japan Advanced Institute of Science and Technology in 2007. He is engaged in development and research of the information security in the government office.

© 2008 Information Processing Society of Japan