IEICE TRANS. FUNDAMENTALS, VOL.E89–A, NO.1 JANUARY 2006

LETTER

Refined Computations for Points of the Form 2^kP Based on Montgomery Trick*

Daisuke ADACHI[†a)], *Nonmember and* **Tomio HIRATA**[††b)], *Member*

SUMMARY This paper focuses on algorithms for an efficient scalar multiplication. It proposes two algorithms for computing points of the form 2^kP in affine coordinates. One works for $k = 2$, and the other works for an arbitrary natural number k. The efficiency of these algorithms is based on a trade-off between a field inversion and several field multiplications. Montgomery trick is used to implement this trade-off. Since a field inversion is usually more expensive than 10 field multiplications, the proposed algorithms are efficient in comparison with existing ones.

key words: *scalar multiplication, elliptic curve arithmetic, Montgomery trick, window method*

1. Introduction

In recent years, elliptic curve cryptographic schemes have become prevailed in commercial use, and they have been built in tiny IC chips. This is because IC cards like smart cards are used for credit cards, insurance certificates, commuter tickets and so on, in place of usual magnetic cards. The execution time of elliptic curve cryptographic schemes heavily depends on that of scalar multiplications. This multiplication takes a point P on an elliptic curve over a finite field and computes a scalar multiple dP for some scalar d.

The 2^w-ary method [3], [7] and the sliding window method [5] are useful for a scalar multiplication. These "window methods" usually use the signed binary representation of the scalar [8], [9], [14] and repeatedly compute points of the form 2^kP from point P on an elliptic curve. These computations use two arithmetics on the elliptic curve; an addition ($P + Q$) and a doubling ($2P$).

Here we focus on an elliptic curve defined over GF(p). In affine coordinates, additions and doublings include inversions over GF(p). However, a field inversion is much expensive than a field squaring or a field multiplication. For example, Sakai and Sakurai [13] reported that the ratio of computation time of a field inversion to a field multiplication is 25.0 for 160-bit p in their implementation. Therefore, reducing the number of field inversions is important for an efficient scalar multiplication.

One method for reducing the number of field inversions is *direct computation* for points on an elliptic curve. For example, the direct computation for 2^kP computes 2^kP directly from P, computing no intermediate points $2P, 4P, \ldots, 2^{k-1}P$. The concept of direct computation was firstly proposed by Guajardo and Paar [6]. They gave algorithms for direct computation of $4P$, $8P$ and $16P$ on an elliptic curve defined over GF(2^n) in affine coordinates. In recent years, several algorithms of direct computation for 2^kP in affine coordinates have been proposed [6], [11], [13]. Sakai and Sakurai [13] proposed an efficient algorithm for 2^kP on an elliptic curve defined over GF(p). This algorithm works for arbitrary natural number k. This paper proposes new efficient algorithms for computing 2^kP based on "Montgomery trick."

2. Preliminaries

2.1 Arithmetics over GF(p)

Additions and doublings are implemented by several kinds of field arithmetics. Among these arithmetics, a field squaring, a field multiplication and a field inversion are more expensive than other field arithmetics, such as a field addition and a field subtraction. We intend to estimate the efficiency of algorithms for computing points on an elliptic curve by the number of the former three field arithmetics**. Moreover, as in [2], [13], we will assume that the cost of a field squaring is 80% as expensive as that of a field multiplication. By dropping "field," we call these field arithmetics just as squaring, multiplication and inversion.

2.2 Addition Formula on Affine Coordinates

Let p denote a prime. $E_p : y^2 \equiv x^3 + ax + b \pmod p (4a^3 + 27b^2 \not\equiv 0)$ is an elliptic curve defined over GF(p). Let $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ be points on E_p. The point $P + Q = (x_{P+Q}, y_{P+Q})$, the result of addition, is derived from the following formulae:

$$x_{P+Q} = \lambda^2 - x_P - x_Q,$$
$$y_{P+Q} = \lambda(x_P - x_{P+Q}) - y_P.$$

Here, $\lambda = \frac{y_Q - y_P}{x_Q - x_P}$ if $P \neq Q$ (addition), and $\lambda = \frac{3x_P^2 + a}{2y_P}$ if

Manuscript received January 14, 2005.
Manuscript revised August 17, 2005.
Final manuscript received October 11, 2005.

[†]The author is with the Graduate School of Engineering, Nagoya University, Nagoya-shi, 464-8603 Japan.
[††]The author is with the Graduate School of Information Science, Nagoya University, Nagoya-shi, 464-8603 Japan.

*The preliminary version of this paper was appeared at SCIS2005. This study is being granted by the 21st Century COE program, Intelligent Media (Speech and Images) Integration for Social Information Infrastructure (at Nagoya University).

a) E-mail: adachi@hirata.nuee.nagoya-u.ac.jp
b) E-mail: hirata@is.nagoya-u.ac.jp

DOI: 10.1093/ietfec/e89–a.1.334

**We ignore field multiplication by small constant because it is much cheaper than general field multiplication.

$P = Q$(doubling). An addition requires 1 squaring, 2 multiplications and 1 inversion, and a doubling requires 2 squarings, 2 multiplications and 1 inversion.

2.3 Montgomery Trick

Montgomery trick [10] is a technique for simultaneous inversions. As a simple example, the inverses modulo p of two numbers x, y can be calculated by the following way:

$$M = xy,\ I = M^{-1},\ x^{-1} = Iy,\ y^{-1} = Ix.$$

These formulae indicate that 2 inversions can be replaced by 1 inversion and 3 multiplications. Similarly, the inverses modulo p of m numbers $x_1^{-1}, x_2^{-1}, \ldots, x_m^{-1} \in \mathrm{GF}(p)$ are calculated as follows.

`Step 1.` Calculate $\prod_{j=1}^{i} x_j$ for each $i = 2, 3, \ldots, m$ and store them.
`Step 2.` Calculate $(\prod_{j=1}^{m} x_j)^{-1}$.
`Step 3.` Calculate $(\prod_{j=1}^{i} x_j)^{-1} \cdot \prod_{j=1}^{i-1} x_j = x_i^{-1}$ and $(\prod_{j=1}^{i} x_j)^{-1} \cdot x_i = (\prod_{j=1}^{i-1} x_j)^{-1}$ for each $i = m, \ldots, 3, 2$.

In Step 1, $(m - 1)$ multiplications are required, and $(2m - 2)$ multiplications are required in Step 3. Thus, $x_1^{-1}, x_2^{-1}, \ldots, x_m^{-1} \in \mathrm{GF}(p)$ can be calculated by 1 inversion and $(3m - 3)$ multiplications. The above replacements are effective if 1 inversion costs more than 3 multiplications.

3. New Efficient Algorithms

In this section, we will propose a new algorithm for computing the quadruple point $4P$ from P which uses the Montgomery trick. Moreover, we will modify the Sakai and Sakurai's algorithm for computing $2^k P$ to introduce the Montgomery trick.

3.1 Formulae for Quadrupling

The window method usually takes window length w from the range of $2 \le w \le 6$. Especially, when the size of auxiliary table in the window method must be decreased, and when we use Straus's trick for fast simultaneous scalar exponentiation [1] , $w = 2$ or $w = 3$ is adopted. Obviously, the quadrupling repeatedly appears in the course of the window method with $w = 2$. Also, the quadrupling appears in the case of $w = 3$ if the window method computes $2^2 P$ first and then computes $2(2^2 Y) + vP$, where v is the value of a "window."

3.1.1 Existing Algorithms

A straightforward computation of a quadruple point $4P$ is to perform two successive doublings. In this computation, 4 squarings, 4 multiplications and 2 inversions are required. Müller proposed an algorithm for direct computation of $4P$ [11]. His algorithm requires 7 squarings, 14 multiplications and 1 inversion. The Sakai and Sakurai's algorithm for $2^k P$ can also compute $4P$ if $k = 2$ [13]. This algorithm requires 9 squarings, 9 multiplications and 1 inversion, and thus it is efficient in comparison with the two successive doublings if 1 inversion costs more than 9 multiplications under our assumptions.

3.1.2 Proposed Algorithm

We present a new algorithm for computing $4P$ which uses the Montgomery trick. Our algorithm is mainly based on the two successive doublings. As we have seen, the straightforward way requires two inverses $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$. If we apply the Montgomery trick for computing these two inverses, a product $(2y_P)(2y_{2P})$ is necessary. However, we cannot compute $2y_{2P} = 2\lambda_1(x_P - x_{2P}) - 2y_P$ without performing inversions. Here we focus on the formula for y_{2P}. By multiplying this formula by $16y_P^3$, we obtain

$$16y_P^3 y_{2P} = 2(3x_P^2 + a)\{12x_P y_P^2 - (3x_P^2 + a)^2\} - 16y_P^4.$$

This formula indicates that $16y_P^3 y_{2P}$ can be computed from x_P and y_P. Thus, we modify the Montgomery trick to compute a product $(2y_P)(16y_P^3 y_{2P})$ instead of $(2y_P)(2y_{2P})$. Defining $E = (2y_P)(16y_P^3 y_{2P})$ and $I = E^{-1}$, we can obtain $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$ by

$$(2y_{2P})^{-1} = 16y_P^4 I, \qquad (2y_P)^{-1} = (16y_P^3 y_{2P})I.$$

The remaining part of our quadrupling algorithm is identical to the two successive doublings. Namely, we first calculate $(2y_P)^{-1}$ and $(2y_{2P})^{-1}$, and then calculate λ_1, x_{2P}, y_{2P}, λ_2, x_{4P}, y_{4P} in this order. We show the detailed version of our quadrupling algorithm in the following.

Algorithm A: Quadrupling in affine coordinates.

`INPUT:` $P = (x_P, y_P)$
`OUTPUT:` $4P = (x_{4P}, y_{4P})$
`Step 1.` Precomputation
$m = 3x_P^2 + a$, $s = x_P(2y_P)^2$, $t = (2y_P)^4$
`Step 2.` Computation of the inverses
$e = 2m(3s - m^2) - t$, $E = (2y_P)e$, $I = E^{-1}, (2y_P)^{-1} = eI$, $(2y_{2P})^{-1} = tI$
`Step 3.` Computation of $4P$
$\lambda_1 = m(2y_P)^{-1}$,
$x_{2P} = \lambda_1^2 - 2x_P$, $y_{2P} = \lambda_1(x_P - x_{2P}) - y_P$
$\lambda_2 = (3x_{2P}^2 + a)(2y_{2P})^{-1}$,
$x_{4P} = \lambda_2^2 - 2x_{2P}$, $y_{4P} = \lambda_2(x_{2P} - x_{4P}) - y_{2P}$

We estimate the efficiency of our quadrupling algorithm. Step 1 requires 3 squarings and 1 multiplication. Step 2 requires 1 squaring, 4 multiplications and 1 inversion. Finally, Step 3 requires 3 squarings and 4 multiplications. Therefore, our quadrupling algorithm requires 7 squarings, 9 multiplications and 1 inversion.

A summary of the efficiency for quadrupling is shown

Table 1 The efficiency of quadrupling.

two doublings	4 squarings, 4 multiplications and 2 inversions
Müller [11]	7 squarings, 14 multiplications and 1 inversion
Sakai and Sakurai ($k = 2$) [13]	9 squarings, 9 multiplications and 1 inversion
proposed quadrupling	7 squarings, 9 multiplications and 1 inversion

in Table 1. Our quadrupling algorithm is more efficient than two successive doublings if one inversion is more expensive than 7.4 multiplications.

3.2 Formulae for Computing 2^kP

This section handles computation of a point 2^kP from P for arbitrary k. The sliding window method [5] is an extension of the 2^w-ary method, in which the window size is at most w. This method performs $Y \leftarrow 2^iY + vP$, where v is the value of the current window, and i is the size of the current window plus the interval between the current window and its neighbor to the left. Therefore, the sliding window method requires computation of 2^kP for various k. Sakai and Sakurai [13] proposed an efficient algorithm for direct computation of 2^kP.

The Sakai and Sakurai's algorithm

INPUT: $P = (x_P, y_P)$
OUTPUT: $2^kP = (x_{2^kP}, y_{2^kP})$
Step 1. Computation of A_1, B_1, C_1
$A_1 = x_P$, $B_1 = 3{x_P}^2 + a$, $C_1 = -y_P$
Step 2. Computation of A_i, B_i, C_i ($i = 2, 3, \ldots, k$)
$A_i = B_{i-1}^2 - 8A_{i-1}C_{i-1}^2$
$B_i = 3A_i^2 + 16^{i-1}a(\prod_{j=1}^{i-1} C_j)^4$
$C_i = -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$
Step 3. Computation of 2^kP
$D_k = 12A_kC_k^2 - B_k^2$
$x_{2^kP} = \frac{B_k^2 - 8A_kC_k^2}{\left(2^k \prod_{j=1}^{k} C_j\right)^2}$
$y_{2^kP} = \frac{8C_k^4 - B_kD_k}{\left(2^k \prod_{j=1}^{k} C_j\right)^3}$

As we have noted, this algorithm does not compute intermediate points $2P, 4P, \ldots, 2^{k-1}P$ in an explicit form. These intermediate points are stored in three terms, A_{i+1}, C_{i+1}, $2^i \prod_{j=1}^{i} C_j$. From these three terms, we can induce $2^iP = (x_{2^iP}, y_{2^iP})$ $(1 \le i \le k-1)$ by

$$x_{2^iP} = A_{i+1} \Big/ \left(2^i \prod_{j=1}^{i} C_j\right)^2 \quad \text{and}$$

$$y_{2^iP} = -C_{i+1} \Big/ \left(2^i \prod_{j=1}^{i} C_j\right)^3.$$

The Sakai and Sakurai's algorithm requires only 1 inversion in Step 3. In addition, $4k + 1$ squarings and $4k + 1$ multiplications are required. We consider the following variant.

A variant of the Sakai and Sakurai's algorithm

INPUT: $P = (x_P, y_P)$
OUTPUT: $2^kP = (x_{2^kP}, y_{2^kP})$
Step 1. Compute $2^{k-1}P$ by the Sakai and Sakurai's algorithm.
Step 2. Compute 2^kP by one doubling.

This variant obviously requires two inversions. We modify this variant using the Montgomery trick and other techniques.

(1) Applying the Montgomery Trick.

The variant of the Sakai–Sakurai's algorithm requires two inverses, $(2^{k-1} \prod_{j=1}^{k-1} C_j)^{-1}$ and $(y_{2^{k-1}P})^{-1}$.

Here we set

$$D_{k-1} = 12A_{k-1}C_{k-1}^2 - B_{k-1}^2, \tag{1}$$
$$X_{2^{k-1}P} = B_{k-1}^2 - 8A_{k-1}C_{k-1}^2, \tag{2}$$
$$Y_{2^{k-1}P} = 8C_{k-1}^4 - B_{k-1}D_{k-1}, \tag{3}$$
$$Z_{2^{k-1}P} = 2^{k-1} \prod_{j=1}^{k-1} C_j. \tag{4}$$

From (4) and the Sakai and Sakurai's algorithm, the following equations are satisfied[†];

$$x_{2^{k-1}P} = X_{2^{k-1}P}/Z_{2^{k-1}P}^2, \tag{5}$$
$$y_{2^{k-1}P} = Y_{2^{k-1}P}/Z_{2^{k-1}P}^3. \tag{6}$$

Therefore, the following equations are induced.

$$\left(2^{k-1} \prod_{j=1}^{k-1} C_j\right)^{-1} = Z_{2^{k-1}P}^{-1}$$
$$(2y_{2^{k-1}P})^{-1} = (2Y_{2^{k-1}P})^{-1} Z_{2^{k-1}P}^3$$

If both $Z_{2^{k-1}P}$ and $2Y_{2^{k-1}P}Z_{2^{k-1}P}^{-3}$ are known, we can compute $(2^{k-1} \prod_{j=1}^{k-1} C_j)^{-1}$ and $(2y_{2^{k-1}P})^{-1}$ by applying the Montgomery trick. However, we cannot compute $2Y_{2^{k-1}P}Z_{2^{k-1}P}^{-3}$ without performing inversions. So we modify the Montgomery trick for computing a product $E = (Z_{2^{k-1}P})(2Y_{2^{k-1}P})$. Also, letting $I = E^{-1}$. Then, two inverses $(2^{k-1} \prod_{j=1}^{k-1} C_j)^{-1}$ and $(2y_{2^{k-1}P})^{-1}$ are calculated as follows;

$$\left(2^{k-1} \prod_{j=1}^{k-1} C_j\right)^{-1} = 2Y_{2^{k-1}P}I,$$
$$(2y_{2^{k-1}P})^{-1} = Z_{2^{k-1}P}^4 I. \tag{7}$$

[†]These equations are equivalent to the definition of Jacobian coordinates (weighted projective coordinates).

Table 2 The efficiency of computing $2^k P$.

Sakai-Sakurai [13]	$4k+1$ squarings, $4k+1$ multiplications and 1 inversion
proposed algorithm	$4k-1$ squarings, $4k+2$ multiplications and 1 inversion

It is obvious that only one inversion is necessary for calculating $(2^{k-1}\prod_{j=1}^{k-1} C_j)^{-1}$ and $(2y_{2^{k-1}P})^{-1}$. We compute $2^{k-1}P = (x_{2^{k-1}P}, y_{2^{k-1}P})$ from (5) and (6).

(2) Modified formula for computing λ.

Next, we will compute $2^k P = (x_{2^k P}, y_{2^k P})$ by one doubling. From (5) and (7), we induce the modified formula for λ as follows[†]:

$$\begin{aligned}\lambda &= (3x_{2^{k-1}P}^2 + a)/2y_{2^{k-1}P} \\ &= (3x_{2^{k-1}P}^2 + a)\cdot Z_{2^{k-1}P}^3 \cdot (2Y_{2^{k-1}P})^{-1} \\ &= (3x_{2^{k-1}P}^2 + a)\cdot Z_{2^{k-1}P}^4 \cdot I \\ &= (3X_{2^{k-1}P}^2 + aZ_{2^{k-1}P}^4)I \\ &= (3X_{2^{k-1}P}^2 + 2\cdot aZ_{2^{k-2}P}^4 \cdot 8C_{k-1}^4)I. \end{aligned} \quad (8)$$

The induced formula (8) contains two terms; $8C_{k-1}^4$ and $aZ_{2^{k-2}P}^4$. These terms have been already calculated because $8C_{k-1}^4$ appears in (3), and $aZ_{2^{k-2}P}^4$ appears in the formula for B_{k-1}[††]. Therefore, if the values of $8C_{k-1}^4$ and $aZ_{2^{k-2}P}^4$ are stored, we can calculate λ from (8). Finally, we compute $(x_{2^k P}, y_{2^k P})$ using $x_{2^{k-1}P}$, $y_{2^{k-1}P}$ and λ.

(3) Summarizing the proposed algorithm

We summarize the variant of the Sakai and Sakurai's algorithm with the Montgomery trick in the following.

Algorithm B: Computing $2^k P$ in affine coordinates.

INPUT: $P = (x_P, y_P)$
OUTPUT: $2^k P = (x_{2^k P}, y_{2^k P})$
Step 1. Computation of A_{k-1}, B_{k-1}, C_{k-1} by the Sakai and Sakurai's algorithm
Step 2. Computation of $X_{2^{k-1}P}$, $Y_{2^{k-1}P}$, $Z_{2^{k-1}P}$ by (1) $\sim$ (4) (Also storing $aZ_{2^{k-2}P}^4$ and $8C_{k-1}^4$)
Step 3. Computation of $(x_{2^{k-1}P}, y_{2^{k-1}P})$
$E = 2Y_{2^{k-1}P}Z_{2^{k-1}P}$, $I = E^{-1}$
$Z_{2^{k-1}P}^{-1} = 2Y_{2^{k-1}P}I$
$x_{2^{k-1}P} = X_{2^{k-1}P}Z_{2^{k-1}P}^{-2}$
$y_{2^{k-1}P} = Y_{2^{k-1}P}Z_{2^{k-1}P}^{-3}$
Step 4. Computation of $(x_{2^k P}, y_{2^k P})$
$\lambda = (3X_{2^{k-1}P}^2 + 2\cdot aZ_{2^{k-2}P}^4 \cdot 8C_{k-1}^4)I$
$x_{2^k P} = \lambda^2 - 2x_{2^{k-1}P}$
$y_{2^k P} = \lambda(x_{2^{k-1}P} - x_{2^k P}) - y_{2^{k-1}P}$

Here we estimate the efficiency of the variant of the Sakai and Sakurai's algorithm. Step 1 requires $4k-7$ squarings and $3k-6$ multiplications. Step 2 requires 3 squarings and k multiplications. Step 3 requires 1 squaring, 5 multiplications and 1 inversion. Finally, step 4 requires 2 squarings and 3 multiplications. Therefore, our algorithm for $2^k P$ requires $4k-1$ squarings, $4k+2$ multiplications and 1 inversion.

We summarize the efficiency of the computation of $2^k P$ in Table 2. In comparison with the Sakai–Sakurai's algorithm, our variant algorithm saves 2 squarings and requires 1 additional multiplication. Namely, our variant algorithm saves 0.6 multiplications.

Table 3 Average time of field arithmetics (μsec).

add	subtract	multiply	square	invert
0.103	0.092	1.209	0.970	37.417

Table 4 Average execution time of the sliding window method (msec).

w	2	3	4	5	6
Case 1	10.629	10.088	9.749	9.557	9.477
Case 2	6.918	5.754	5.051	4.624	4.381
Case 3	6.854	5.714	5.022	4.599	4.358
C3/C1	0.644	0.566	0.515	0.481	0.469
C3/C2	0.990	0.993	0.994	0.995	0.995

Table 5 Average execution time of the 2^w-ary method (msec).

w	2	3	4	5	6
Case 1	10.717	10.334	9.996	9.803	9.766
Case 2	8.185	6.601	5.674	5.144	4.888
Case 3	7.897	6.540	5.630	5.109	4.856
C3/C1	0.737	0.633	0.563	0.521	0.497
C3/C2	0.965	0.991	0.992	0.993	0.993

4. Application to Scalar Multiplication

The window method repeatedly computes points of the form $2^k P$, where k is sliding width to the next window. We consider the following three cases in implementation of computing $2^k P$. In Case 1, only doublings are used. In Case 2, the Sakai and Sakurai's algorithm is applied for the computations of the form $2^k P$. Similarly, In Case 3, Algorithm A (for $k = 2$) and Algorithm B (for $k \geq 3$) are applied for $2^k P$.

For the three cases, we perform the following experiment. P is a random point on the elliptic curve, and d is a random scalar. We measure the execution time for computing a dP for 10,000 times and calculate the average. This experiment is performed on a PC which has Pentium III 700 MHz processor and uses an elliptic curve "NIST P-192" [12] defined over GF(p), where p is a 192 bit prime. Table 3 shows the average execution time of field arithmetics on GF(p). From this table, the ratio of a squaring to a multiplication is observed as 0.802, which verifies our assump-

[†]From (4), $aZ_{2^{k-1}P}^4 = a(2^{k-1}\prod_{j=1}^{k-1} C_j)^4$
$= 16C_{k-1}^4 \cdot a(2^{k-2}\prod_{j=1}^{k-2} C_j)^4$
$= 2\cdot 8C_{k-1}^4 \cdot aZ_{2^{k-2}P}^4$.

[††]From (4), $16^{k-2}a(\prod_{j=1}^{k-2} C_j)^4 = a(2^{k-2}\prod_{j=1}^{k-2} C_j)^4 = aZ_{2^{k-2}P}^4$.

Table 6 The average number of multiplications, squarings and inversions.

	Sliding window, $w = 2$			Sliding window, $w = 3$			Sliding window, $w = 4$			Sliding window, $w = 5$		
	Mul	Sqr	Inv	Mul	Sqr	Inv	Mul	Sqr	Inv	Mul	Sqr	Inv
Case 1	255.48	383.32	255.48	241.93	385.14	240.93	243.85	388.61	231.85	258.93	393.66	225.93
Case 2	893.77	894.02	128.13	862.87	863.12	97.85	853.82	846.07	79.59	861.03	837.28	67.71
Case 3	925.61	765.89	128.13	910.80	767.27	97.85	892.11	769.47	79.59	892.89	773.57	67.71
	Sliding window, $w = 6$			2^w-ary, $w = 2$			2^w-ary, $w = 3$			2^w-ary, $w = 4$		
	Mul	Sqr	Inv	Mul	Sqr	Inv	Mul	Sqr	Inv	Mul	Sqr	Inv
Case 1	295.88	402.73	221.88	255.48	383.32	255.48	251.49	385.32	245.49	264.08	389.32	236.08
Case 2	891.99	836.24	59.53	920.48	922.14	160.48	881.49	879.15	119.49	875.08	855.74	95.08
Case 3	919.26	781.70	59.53	920.48	732.14	160.48	944.49	753.15	119.49	922.08	761.74	95.08
	2^w-ary, $w = 5$			2^w-ary, $w = 6$								
	Mul	Sqr	Inv	Mul	Sqr	Inv						
Case 1	307.97	399.32	228.97	403.34	419.32	223.34						
Case 2	908.95	848.95	79.30	992.34	853.00	68.34						
Case 3	946.61	772.95	79.30	1023.34	791.00	68.34						

tion in 2.1.

We first show the experimental results of the sliding window method in Table 4. For every window length w, we use "width-w NAF" [14] to represent d. In this table, Case 3 has the least average execution time among the three cases.

We next show the experimental results of the 2^w-ary method in Table 5. To avoid increasing the value of "windows," we use "width-2 NAF" [14] for the representation of d. Again, Case 3 has the least average execution time among the three cases.

One remarkable feature is that the improvement of execution time of Case 3 to Case 2 is relatively large (3.5%) when $w = 2$. This is because Algorithm A is always applied when $w = 2$, while Algorithm B is always applied when $w \geq 3$. From Table 1 and Table 2, we can see that Algorithm A reduces two squarings comparing to the Sakai and Sakurai's algorithm, while Algorithm B reduces two squarings but requires one extra multiplication. Therefore, the improvement of execution time is larger when $w = 2$.

On the other hand, in the case of the sliding window method, the improvement of execution time of Case 3 to Case 2 is small when $w = 2$. This is because both Algorithm A and Algorithm B are applied when $w = 2$[†].

Table 6 shows the average number of field multiplications, squarings and inversions per one scalar multiplication. For example, in the case of the sliding window method ($w = 2$), Case 3 has 128.13(= 894.02 − 765.89) less squarings and 31.84(= 925.61 − 893.77) more multiplications than Case 2. From Table 3, the effect of these reduced squarings and increased multiplications is estimated as $128.13 * 0.970 - 31.84 * 1.209 = 85.80\,(\mu\text{sec})$. We can expect the improvement of the execution time of the scalar multiplication by this amount. However, from Table 4, actual improvement is 64(= 6918 − 6854) (μsec). It is considered that this discrepancy is brought by the field operations which we ignore in this paper (multiplication by small constant, addition, subtraction, etc). Similar discrepancy is observed for other window length w and the 2^w-ary method.

[†]Width-w NAF always has sliding width more than w.

5. Conclusion

This paper has presented two algorithms for computing points of the form $2^k P$ which use the Montgomery trick. The one only works for $k = 2$, and the other works for an arbitrary natural number k. We have shown that the proposed algorithms are more efficient in comparison with existing algorithms.

References

[1] R.M. Avanzi, "On multi-exponentiation in cryptography," IACR Cryptology ePrint Archive, 2002, Available at, http://eprint.iacr.org/2002/154.ps.gz

[2] M. Ciet, M. Joye, K. Lauter, and P.L. Montgomery, "Trading inversions for multiplications in elliptic curve cryptography," IACR Cryptology ePrint Archive, 2003, Available at, http://eprint.iacr.org/2003/257.ps.gz

[3] H. Cohen, A Course in Computational Algebraic Number Theory, Graduate Texts in Math., no.138, Springer-Verlag, Berlin, 1993.

[4] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates," Advances in Cryptology–ASIACRYPT'98, LNCS, vol.1514, pp.51–65, Springer-Verlag, 1998.

[5] D.M. Gordon, "A survey of fast exponentiation methods," J. Algorithms, vol.27, pp.129–146, 1998.

[6] J. Guajardo and C. Paar, "Efficient algorithms for elliptic curve cryptosystems," Advances in Cryptology—Crypto'97, LNCS, vol.1294, pp.342–356, Springer-Verlag, 1997.

[7] D.E. Knuth, The Art of Computer Programming, vol.2, Seminumerical Algorithms, 3rd ed., Addison-Wesley, Reading, MA, 1997.

[8] K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method," Advances in Cryptology—Crypto'92, LNCS, vol.740, pp.345–357, Springer-Verlag, 1993.

[9] F. Morain and J. Olivos, "Speeding up the computations on an elliptic curve using addition-subtraction chains," Theoretical Informatics and Applications, vol.24, no.6, pp.531–544, 1990.

[10] P.L. Montgomery, "Speeding the Pollard and elliptic curve methods of factorization," Math. Comput., vol.48, pp.243–264, 1987.

[11] V. Müller, "Efficient algorithms for multiplication on elliptic curves," Proc. GI-Arbeitskonferenz Chipkarten 1998, TU München, 1998.

[12] NIST, Recommended elliptic curves for federal government

use, 1999, Available at, http://csrc.nist.gov/encryption/dss/ecdsa/NISTReCur.pdf
[13] Y. Sakai and K. Sakurai, "Efficient scalar multiplications on elliptic curves with direct computations of several doublings," IEICE Trans. Fundamentals, vol.E84-A, no.1, pp.120–129, Jan. 2001.
[14] J.A. Solinas, "Efficient arithmetic on Koblitz curves," Des. Codes Cryptogr., vol.19, pp.195–249, 2000.