IEICE TRANS. FUNDAMENTALS, VOL.E84–A, NO.1 JANUARY 2001

PAPER *Special Section on Cryptography and Information Security*

Efficient Scalar Multiplications on Elliptic Curves with Direct Computations of Several Doublings*

Yasuyuki SAKAI† *and* **Kouichi SAKURAI**††, *Regular Members*

SUMMARY We introduce efficient algorithms for scalar multiplication on elliptic curves defined over $\mathbb{F}_p$. The algorithms compute 2^kP directly from P, where P is a random point on an elliptic curve, without computing the intermediate points, which is faster than k repeated doublings. Moreover, we apply the algorithms to scalar multiplication on elliptic curves, and analyze their computational complexity. As a result of their implementation with respect to affine (resp. weighted projective) coordinates, we achieved an increased performance factor of 1.45 (45%) (resp. 1.15 (15%)) in the scalar multiplication of the elliptic curve of size 160-bit.

key words: *elliptic curve cryptosystems, scalar multiplication, window method, coordinate system, implementation*

1. Introduction

Elliptic curve cryptosystems, which were suggested by Miller [11] and Koblitz [8], are now widely used in various security services. IEEE and other standards bodies such as ANSI and ISO are in the process of standardizing elliptic curve cryptosystems. Therefore, it is very attractive to provide algorithms that allow efficient implementation [2], [4], [9], [12], [13], [16].

Encryption/decryption or signature generation/verification schemes require computation of scalar multiplication. The computational performance of cryptographic protocols with elliptic curves strongly depends on the efficiency of scalar multiplication. Thus, fast scalar multiplication is essential for elliptic curve cryptosystems.

In typical methods for scalar multiplication, an addition of two points and a doubling of a point, are calculated repeatedly, but the point doublings are quite costly. There are several ways to speed-up scalar multiplication, such as: (1) reducing the number of additions, (2) speeding-up doubling, and (3) using a mixed coordinate strategy [2]. Our contribution will deal with the second approach.

One method to increase the speed of doublings is *direct* computation of several doublings, which computes 2^kP directly from $P \in E(\mathbb{F}_q)$, without computing the intermediate points $2P,\ 2^2P, \cdots, 2^{k-1}P$. The concept of direct computation was first suggested by Guajardo and Paar in [4]. They formulated algorithms for direct computation of $4P$, $8P$ and $16P$ on elliptic curves over $\mathbb{F}_{2^n}$ in terms of affine coordinates. Recent related results include a formula for computing $4P$ on elliptic curves over $\mathbb{F}_p$ in affine coordinates by Müller [14] and a formula for computing $4P$ on elliptic curves over $\mathbb{F}_p$ in projective coordinates by Miyaji, Ono and Cohen [12]. These formulae are more efficient than repeated doublings. However, the known formulae work only with small k (2, 3 or 4) and formulae in terms of weighted projective coordinates have not been given. One remarkable result was given by Cohen, Mijaji and Ono [2]. They used a redundant representation of points such as (X, Y, Z, aZ^4). With this representation, point doubling can be accomplished with complexity $4S + 4M$, where S and M denote a squaring and a multiplication in $\mathbb{F}_p$ respectively. Itoh et al. also gave a similar method for doubling [7]. This representation is called *modified jacobian coordinates* [2]. We can use this coordinate system for direct computation of several doublings. However, addition of points in modified jacobian coordinates is relatively costly compared to weighted projective coordinates. Therefore, one possible strategy for efficient scalar multiplication is to mix several coordinate systems (See [2]).

In this paper, we propose efficient algorithms for speeding-up elliptic curve cryptosystems with curves over $\mathbb{F}_p$ in terms of affine coordinates and weighted projective coordinates. We construct efficient formulae which compute 2^kP directly for $\forall k \geq 1$. Our formulae compute 2^kP directly from $P \in E(\mathbb{F}_p)$ without computing the intermediate points $2P,\ 2^2P, \cdots, 2^{k-1}P$. In the case of affine coordinates, our formula has computational complexity $(4k+1)S+(4k+1)M+I$, where I denotes an inversion in $\mathbb{F}_p$. This is more efficient than k repeated doublings, which requires k inversions. When implementing our direct computation method, experimental results show that computing $16P$ achieves a 90 percent performance increase over 4 doublings in affine coordinates.

Moreover, we show a method of elliptic scalar multiplication that is combined with our direct computation methods. This method is based on a sliding signed binary window method [9]. We also implement scalar multiplication and discuss the efficiency. Our imple-

Manuscript received March 24, 2000.
Manuscript revised August 28, 2000.

†The author is with Information Technology R&D Center, Mitsubishi Electric Corporation, Kamakura-shi, 247-8501 Japan.
††The author is with the Faculty of Engineering, Kyushu University, Fukuoka-shi, 812-8581 Japan.

*This paper was partially presented at SCIS 2000.

mentation results show that in the case of affine (resp. weighted projective) coordinates, we achieved a 45% speed increase (resp. (15%)) in the scalar multiplication of the elliptic curve of size 160-bit. Moreover, our implementation with the new method shows that when $\log_2 p$ is relatively large (384 or larger), the speed of scalar multiplication in affine coordinates close to that in weighted projective coordinates.

The algorithms proposed in this paper do not depend on specific curve parameters. Therefore, our methods can be applied to any elliptic curve defined over $\mathbb{F}_p$.

This paper is organized as follows. In Sect. 2, we give some notations. In Sect. 3, previous work will be summarized. In Sects. 4 and 5, we will describe our new algorithms for direct computation of $2^k P$ in terms of affine coordinate and weighted projective coordinate, respectively. In Sect. 6, we will discuss a method for elliptic scalar multiplications with our algorithms. In Sect. 7, timings of our implementation will be given. Finally, conclusions and further work.

2. Notations

Throughout this paper, we will use the following notations.

Let $\mathbb{F}_p$ denotes a prime finite field with p elements. We consider an elliptic curve E given by

$$E : Y^2 = X^3 + aX + b \quad (a, b \in \mathbb{F}_p,\ p > 3,\ 4a^3 + 27b^2 \neq 0)$$

Let $P_1 = (x_1, y_1)$ and $P_{2^k} = 2^k P_1 = (x_{2^k}, y_{2^k}) \in E(\mathbb{F}_p)$. Let S, M and I denote a squaring, a multiplication and an inversion in $\mathbb{F}_p$, respectively. When we estimate a computational efficiency, we will ignore the cost of a field addition, as well as the cost of a multiplication by small constants.

3. Previous Work

In this section, we summarize known algorithms for point doubling and direct computation of several doublings.

3.1 Point Doubling

In terms of affine coordinates, point doubling can be accomplished as follows: Assume $P_1 = (x_1, y_1) \neq \mathcal{O}$, where $\mathcal{O}$ denotes the point at infinity. The point $P_2 = (x_2, y_2) = 2P_1$ can be computed as follows.

$$\begin{aligned} x_2 &= \lambda^2 - 2x_1 \\ y_2 &= (x_1 - x_2)\lambda - y_1 \\ \lambda &= \frac{3x_1^2 + a}{2y_1} \end{aligned} \tag{1}$$

The formulae above have computational complexity $2S + 2M + I$ [6].

In terms of weighted projective coordinate, doubling can be accomplished as follows. Assume $P_1 = (X_1, Y_1, Z_1) \neq \mathcal{O}$, The point $P_2 = (X_2, Y_2, Z_2) = 2P_1$ can be computed as follows.

$$\begin{aligned} X_2 &= M^2 - 2S \\ Y_2 &= M(S - X_2) - T \\ Z_2 &= 2Y_1 Z_1 \\ M &= 3X_1^2 + aZ_1^4 \\ S &= 4X_1 Y_1^2 \\ T &= 8Y_1^4 \end{aligned} \tag{2}$$

In cases of general curves, the formulae above have computational complexity $6S + 4M$ [6].

3.2 Direct Doubling

The concept to use direct computations of $2^k P$ for efficient elliptic scalar multiplications was firstly proposed by Guajardo and Paar in [4]. They formulated algorithms for computing $4P$, $8P$ or $16P$ on elliptic curves over $\mathbb{F}_{2^n}$ in terms of affine coordinate. In recent years, several authors have given ways to compute $2^k P$ directly (but limited to small k). The following section summarizes previous work on direct computation of several doublings.

1. Guajardo and Paar [4] proposed formulae for computing $4P$, $8P$ and $16P$ on elliptic curves over $\mathbb{F}_{2^n}$ in terms of affine coordinates.
2. Müller [14] proposed a formula for computing $4P$ on elliptic curves over $\mathbb{F}_p$ in terms of affine coordinates.
3. Miyaji, Ono and Cohen [12] proposed a formula for computing $4P$ on elliptic curves over $\mathbb{F}_p$ in terms of projective coordinates.
4. Han and Tan [5] proposed formulae for computing $3P$, $5P$, $6P$ and $7P$, etc., on elliptic curves over $\mathbb{F}_{2^n}$ in terms of affine coordinates.

The formulae above can efficiently compute $2^k P$ compared to computing k doublings. However, the formulae listed below have not been explicitly given.

1. General (i.e., $\forall k \geq 1$) formulae for direct computations in elliptic curves over $\mathbb{F}_p$ in terms of affine coordinates.
2. Formulae for direct computations in terms of weighted projective coordinates.

In later sections, we will formulate the above algorithms, and analyze their computational complexity. In the case of $k = 2$, i.e., $4P$, on elliptic curves over $\mathbb{F}_p$ in affine coordinates, our algorithms are more efficient than Müller's algorithm.

We should remark that the algorithm proposed by Cohen, Miyaji and Ono in [2] can be efficiently used for direct computation of several doublings. The authors call their algorithm a "*modified jacobian*" coordinate system. The coordinate system uses (redundant) mixed representation, e.g., (X, Y, Z, aZ^4). Doubling in terms of the modified jacobian coordinates has computational advantages over weighted projective (jacobian) coordinates.

4. Direct Computations of 2^kP in Affine Coordinates

In this section, we provide formulae for direct computations of 2^kP, where $\forall k \geq 1$, in terms of affine coordinate. In the next section, we will show formulae in terms of weighted projective coordinate. We also discuss their computational efficiency.

In the case of affine coordinates, direct computation of several doublings may be significantly more efficient, as suggested in [4], because we can construct formulae that require only one modular inversion, as opposed to the k inversions that k separate doubling operations would require for computing 2^kP.

Modular inversion is generally more expensive than modular multiplication [17]. Therefore, direct computation of several doublings may be effective in elliptic scalar multiplication in terms of affine coordinates.

4.1 Doubling

We begin by showing doubling formulae with the purpose of constructing formulae for general (i.e., $\forall k \geq 1$) cases.

Let

$$\begin{aligned} A_1 &= x_1 \\ B_1 &= 3x_1^2 + a \\ C_1 &= -y_1 \\ D_1 &= 12A_1C_1^2 - B_1^2 \end{aligned}$$

Then doubled point $P_2 = (x_2, y_2)$ of $P_1 = (x_1, y_1)$ can be computed as follows.

$$\begin{aligned} x_2 &= \frac{B_1^2 - 8A_1C_1^2}{(2C_1)^2} \\ y_2 &= \frac{8C_1^4 - B_1D_1}{(2C_1)^3} \end{aligned} \tag{3}$$

Note that although the denominator of x_2 differs from that of y_2, the formulae above require only one inversion if we multiply the numerator of x_2 by $2C_1$.

The formulae have computational complexity $5S + 5M + I$. On the other hand, the formulae (1) have complexity $6S + 4M + I$. Therefore, it is clear that the formulae given in this subsection are inefficient. (We showed the above formulae only for the purpose of constructing formulae for $k > 1$, as stated previously.)

4.2 Computing $4P$

In affine coordinates, quadrupling a point can be accomplished by the following:

$$\begin{aligned} A_2 &= B_1^2 - 8A_1C_1^2 \\ B_2 &= 3A_2^2 + 16aC_1^4 \\ C_2 &= -8C_1^4 - B_1(A_2 - 4A_1C_1^2) \\ D_2 &= 12A_2C_2^2 - B_2^2 \\ x_4 &= \frac{B_2^2 - 8A_2C_2^2}{(4C_1C_2)^2} \\ y_4 &= \frac{8C_2^4 - B_2D_2}{(4C_1C_2)^3} \end{aligned} \tag{4}$$

where $P_4 = (x_4, y_4) = 4P_1 = 4(x_1, y_1)$.

The formulae have computational complexity $9S + 9M + I$. Müller's formula [14] has complexity $7S + 14M + I$. Therefore, our formula, above, is clearly more efficient. Moreover, we will show in a later section that computing $4P$ by the formula given in this section has less complexity than computing two separate doublings by (1).

4.3 Computing $8P$

In affine coordinates, computing $P_8 = (x_8, y_8) = 8P_1 = 8(x_1, y_1)$ can be accomplished by the following.

$$\begin{aligned} A_3 &= B_2^2 - 8A_2C_2^2 \\ B_3 &= 3A_3^2 + 256aC_1^4C_2^4 \\ C_3 &= -8C_2^4 - B_2(A_3 - 4A_2C_2^2) \\ D_3 &= 12A_3C_3^2 - B_3^2 \\ x_8 &= \frac{B_3^2 - 8A_3C_3^2}{(8C_1C_2C_3)^2} \\ y_8 &= \frac{8C_3^4 - B_3D_3}{(8C_1C_2C_3)^3} \end{aligned} \tag{5}$$

These formulae have computational complexity $13S + 13M + I$. We will later show that computing $8P$ by the formulae given in this section has less complexity than computing three separate doublings by (1).

Table 1 Complexity comparison.

Calculation	Method	Complexity			Break-Even Point
		S	M	I	
$4P$	Direct Doublings	9	9	1	$8.6M < I$
	Separate 2 Doublings	4	4	2	
$8P$	Direct Doublings	13	13	1	$6.3M < I$
	Separate 3 Doublings	6	6	3	
$16P$	Direct Doublings	17	17	1	$5.4M < I$
	Separate 4 Doublings	8	8	4	
2^kP	Direct Doublings	$4k+1$	$4k+1$	1	$\frac{3.6k+1.8}{k-1}M < I$
	Separate k Doublings	$2k$	$2k$	k	

4.4 Computing $16P$

In affine coordinates, computing $P_{16} = (x_{16}, y_{16}) = 16P_1 = 16(x_1, y_1)$ can be accomplished by the following.

$$
\begin{aligned}
A_4 &= B_3^2 - 8A_3C_3^2 \\
B_4 &= 3A_4^2 + 4096aC_1^4C_2^4C_3^4 \\
C_4 &= -8C_3^4 - B_3(A_4 - 4A_3C_3^2) \\
D_4 &= 12A_4C_4^2 - B_4^2 \\
x_{16} &= \frac{B_4^2 - 8A_4C_4^2}{(16C_1C_2C_3C_4)^2} \\
y_{16} &= \frac{8C_4^4 - B_4D_4}{(16C_1C_2C_3C_4)^3} \qquad (6)
\end{aligned}
$$

The formulae have computational complexity $17S + 17M + I$. We will show in a later section that computing $16P$ by the formulae given in this section has less complexity than computing four separate doublings by (1).

4.5 The Formulae Computing 2^kP in Affine Coordinate

From the formulae, which compute $4P$, $8P$ or $16P$, given in the previous subsections, we can easily obtain general formulae that allow direct doubling $P \mapsto 2^kP$, where $k \geq 1$. The figure shown below describes these formulae, and their computational complexity is given as Theorem 1.

Algorithm 1: Direct computation of 2^kP in affine coordinates, where $k \geq 1$ and $P \in E(\mathbb{F}_p)$.

`INPUT:` $P_1 = (x_1, y_1) \in E(\mathbb{F}_p)$
`OUTPUT:` $P_{2^k} = 2^kP_1 = (x_{2^k}, y_{2^k}) \in E(\mathbb{F}_p)$

`Step 1.` Compute A_1, B_1 and C_1

$$
\begin{aligned}
A_1 &= x_1 \\
B_1 &= 3x_1^2 + a \\
C_1 &= -y_1
\end{aligned}
$$

`Step 2.` For i from 2 to k compute A_i, B_i and C_i

$$
\begin{aligned}
A_i &= B_{i-1}^2 - 8A_{i-1}C_{i-1}^2 \\
B_i &= 3A_i^2 + 16^{i-1}a(\prod_{j=1}^{i-1} C_j)^4 \\
C_i &= -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)
\end{aligned}
$$

`Step 3.` Compute D_k

$$D_k = 12A_kC_k^2 - B_k^2$$

`Step 4.` Compute x_{2^k} and y_{2^k}

$$
\begin{aligned}
x_{2^k} &= \frac{B_k^2 - 8A_kC_k^2}{\left(2^k \prod_{i=1}^{k} C_i\right)^2} \\
y_{2^k} &= \frac{8C_k^4 - B_kD_k}{\left(2^k \prod_{i=1}^{k} C_i\right)^3}
\end{aligned}
$$

Theorem 1: In terms of affine coordinates, there exists an algorithm that computes 2^kP in at most $4k+1$ squarings, $4k+1$ multiplications, and one inversion in $\mathbb{F}_p$ for any point $P \in E(\mathbb{F}_p)$.

The proof is given in Appendix A. It should be noted that the point P_1 has to be an element with an order larger than 2^k. This requirement ensures that 2^kP will never equal $\mathcal{O}$.

4.6 Complexity Comparison

In this subsection, we compare the computational complexity of a direct doubling of 2^kP given in the previous subsection and separate k repeated doublings. The complexity of a doubling is estimated based on the algorithm given in [6]. Table 1 shows the number of squarings S, multiplications M, and inversions I in $\mathbb{F}_p$. We should point out that our method reduces inversions

at the cost of multiplications. Therefore, the performance of the new formulae depends on the cost factor of one inversion relative to one multiplication. For this purpose we introduce, as in [4], the notation of a "*break-even point*." It is possible to express the time that it takes to perform one inversion in terms of the equivalent number of multiplications needed per inversion. In this comparison, we assume that one squaring has complexity $S = 0.8M$. We also assume that the cost of field addition and multiplication by small constants can be ignored.

As we can see from Table 1, if a field inversion has complexity $I > 8.6M$, one quadrupling may be more efficient than two separate doublings. In cases that $\mathbb{F}_p$ has a size of 160-bit or larger, it is extremely likely that $I > 8.6M$ in many implementations (i.e., [17]). Moreover, in cases of $k > 2$, our direct computation method may be more efficient than individual doublings in most implementations.

5. Direct Computation of $2^k P$ in Weighted Projective Coordinates

In this section, we provide formulae for direct computation of $2^k P$ in terms of weighted projective coordinates and discuss the computational efficiency.

5.1 General Formulae for Computing $2^k P$ in Weighted Projective Coordinates

In cases where field inversions are significantly more expensive than multiplications, it is preferable to use weighted projective coordinates (also referred to as *Jacobian coordinates*), where a triplet (X, Y, Z) corresponds to the affine coordinates $(X/Z^2, Y/Z^3)$ whenever $Z \neq 0$.

From Algorithm 1, we can immediately derive formulae for direct computation of $2^k P$ in terms of weighted projective coordinates as follows. The computational complexity of Algorithm 2 is given as Theorem 2.

Algorithm 2: Direct computation of $2^k P$ in weighted projective coordinates, where $k \geq 1$ and $P \in E(\mathbb{F}_p)$.

`INPUT:` $P_1 = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$
`OUTPUT:` $P_{2^k} = (X_{2^k}, Y_{2^k}, Z_{2^k}) = 2^k P_1 \in E(\mathbb{F}_p)$

`Step 0.` Mapping: $(X_1, Y_1, Z_1) \mapsto (X_1', Y_1', 1)$
(if $Z_1 = 0$ terminate with $P_{2^k} = \mathcal{O}$)

`Step 1.` Compute A_1, B_1 and C_1

$$A_1 = X_1'$$
$$B_1 = 3{X_1'}^2 + a$$
$$C_1 = -Y_1'$$

`Step 2.` For i from 2 to k compute A_i, B_i and C_i

$$A_i = B_{i-1}^2 - 8A_{i-1}C_{i-1}^2$$
$$B_i = 3A_i^2 + 16^{i-1}a(\prod_{j=1}^{i-1} C_j)^4$$
$$C_i = -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$$

`Step 3.` Compute D_k

$$D_k = 12A_kC_k^2 - B_k^2$$

`Step 4.` Compute X_{2^k}, Y_{2^k} and Z_{2^k}

$$X_{2^k} = B_k^2 - 8A_kC_k^2$$

$$Y_{2^k} = 8C_k^4 - B_kD_k$$

$$Z_{2^k} = 2^k \prod_{i=1}^{k} C_i$$

Theorem 2: In terms of weighted projective coordinates, there exists an algorithm that computes $2^k P$ in at most $4k$ squarings, $4k - 2$ multiplications in $\mathbb{F}_p$, except `Step 0`, for any point $P \in \mathbb{F}_p$.

The proof is given in Appendix A.

5.2 Quadrupling

In Algorithm 3, which follows, we describe formulae for computing $4P = (X_4, Y_4, Z_4)$ directly in terms of weighted projective coordinates. The algorithm does not require `Step 0` in Algorithm 2.

Algorithm 3: Computing $4P$ in weighted projective coordinates

---

`INPUT:` $P_1 = (X_1, Y_1, Z_1) \in E(\mathbb{F}_p)$
`OUTPUT:` $P_4 = (X_4, Y_4, Z_4) = 4P_1 \in E(\mathbb{F}_p)$

`Step 1.` Compute A, B, C and D

$$A = 3X_1^2 + aZ_1^4$$
$$B = A^2 - 8X_1Y_1^2$$
$$C = -8Y_1^4 + A(12X_1Y_1^2 - A^2)$$
$$D = 16aY_1^4Z_1^4 + 3B^2$$

`Step 2.` Compute X_4, Y_4 and Z_4

$$X_4 = -8BC^2 + D^2$$

$$Y_4 = -8C^4 + D(12BC^2 - D^2)$$

$$Z_4 = 4Y_1Z_1C$$

The algorithm above has computational complexity $10S + 9M$.

As we have stated before, if a modified jacobian representation (X, Y, Z, aZ^4) is used, a point doubling can be accomplished with complexity $4S + 4M$ [2]. Therefore, a quadrupling can be accomplished with complexity $8S + 8M$. Itoh et al. also gave a way to quadruplicate with complexity $8S + 8M$ [7]. However, in modified jacobian coordinates, the formulae for addition of points that have complexity $6S + 13M$ are more costly than those in weighted projective coordinates. One optimal way to speed-up scalar multiplication is to mix modified jacobian coordinates with projective, weighted projective, or Chudnovsky jacobian coordinates [1], [2].

6. Scalar Multiplications with Direct Computations of $2^k P$

6.1 The Algorithm

By using our previous formulae for direct computation of $2^k P$, we can improve elliptic scalar multiplication with the sliding signed binary window method [3], [9]. For example we apply our new formulae to the window method with windows of length 4. We represent a scalar m in $P \mapsto mP$ with a *nonadjacent form* (NAF)[†]. For example, $m = (1101110111)_2$ will be represented as $m' = (100\tilde{1}000\tilde{1}00\tilde{1})_{NAF}$, where $\tilde{1}$ denotes -1.

Algorithm 4 describes scalar multiplications on elliptic curves using our direct computations of $2^k P$ in the case of k up to 4.

Algorithm 4: Elliptic scalar multiplication using our direct computation of $2^k P$ in the case of k up to 4

`INPUT:` $P \in E(\mathbb{F}_p)$, $m \in \mathbb{Z}$
`OUTPUT:` $mP \in E(\mathbb{F}_p)$

`Step 1.` Construct NAF representation
$m = (e_t e_{t-1} \cdots e_1 e_0)_{NAF}$,
$e_i \in \{-1, 0, 1\}$
`Step 2.` Precomputation
`2.1` $P_6 \leftarrow 6P$
`2.2` For i from 7 to 10 do:
$P_i \leftarrow P_{i-1} + P$
`Step 3.` $P_m \leftarrow \mathcal{O}$, $i \leftarrow t$
`Step 4.` While $i \geq 3$ do the following:
`4.1` If $e_i = 0$ then:
find the longest bitstring $e_i e_{i-1} \cdots e_l$
such that $e_i = e_{i-1} = \cdots e_l = 0$,
and do the following
$P_m \leftarrow 2^{i-l+1} P_m$
$i \leftarrow l - 1$
`4.2` else ($e_i \neq 0$):
If $(e_i e_{i-1} e_{i-2} e_{i-3})_{NAF} > 0$ then:
$P_m \leftarrow 16P_m + P_{(e_i e_{i-1} e_{i-2} e_{i-3})_{NAF}}$
else:
$P_m \leftarrow 16P_m - P_{|(e_i e_{i-1} e_{i-2} e_{i-3})_{NAF}|}$
$i \leftarrow i - 4$
`Step 5.` $P_m \leftarrow (e_i \cdots e_0)_{NAF} P_m$
using traditional double-add method
`Step 6.` Return P_m

In Algorithm 4, we compute $16P$ directly from P in each window instead of 4 separate doublings. In `Step 4.1` with strings of zero-runs in the scalar m_{NAF}, we should choose computations $16P$, $8P$, $4P$ or $2P$ optimally. This can be done with rules such as: (1) If a length of zero equals to 4, we compute $16P$. (2) If a length of zero equals to 3, we compute $8P$, and so on.

Note that the computation for `Step 5` is not relatively expensive if m is large.

Using our algorithms for scalar multiplication, many of the doublings in an ordinary window method will be replaced by the direct computation of $16P$. Therefore, if one computation of $16P$ is relatively faster than four doublings, scalar multiplication with our method may be significantly improved. We will examine this improvement by real implementation in the next section.

6.2 Number of $2^k P$ Computations in the Window Method

Table 2 shows the number of required computations of $2^k P$ and additions in the sliding signed binary window method based on Algorithm 4. The window size shown in the table is 2 and 4 as examples. The numbers were counted by our implementation. In the case of a window of length 2, direct computations of $4P$ can be used. In the case of a window of length 4, direct computations of $4P$, $8P$ and $16P$ can be used.

We can see from the table that: (1) With direct computations of up to $16P$, the computational efficiency of $16P$ significantly affects scalar multiplication. (2) With direct computations of up to $4P$, the computational efficiency of $4P$ significantly affects scalar multiplication.

6.3 Complexity Comparison of Scalar Multiplications

Based on the number of computation of $2^k P$ in scalar multiplication, given in Table 2, we compared the computational complexity of scalar multiplication. For example in the case of 160-bit scalar, the complexity of scalar multiplication using direct computation with

[†]Koyama and Tsuruoka pointed out that NAF is not necessarily the optimal representation to use [3], [9]. Although it has minimal weight, allowing a few adjacent nonzeros may increase the length of zero runs, which, in turn, reduces the total number of additions. Their method may be useful for our scalar multiplication with direct computations of $2^k P$.

Table 2 Number of computations of $2^k P$, where $k = 1, 2, 3$ or 4, and additions in the sliding signed binary window method with window length 2 or 4.

Curves	Add.	Window of length 4				Window of length 2	
		$2P$	$4P$	$8P$	$16P$	$2P$	$4P$
P160	36.82	14.93	4.99	2.58	31.75	17.51	71.07
P192	37.63	15.29	5.06	2.64	39.54	17.93	86.78
P224	37.77	15.31	5.05	2.69	47.49	18.00	102.72
P256	41.77	15.31	5.06	2.70	55.53	18.01	118.82
P384	48.76	17.13	5.10	3.59	86.26	22.72	181.21
P521	90.39	31.34	14.51	6.94	109.87	39.28	241.19

$k = 4$ can be evaluated as: $C = 36.82A + 14.93D_2 + 4.99D_4 + 2.58D_8 + 31.75D_{16}$, where A, D_2, D_4, D_8, and D_{16} denote the complexity of the computation for point addition, doubling, $4P$, $8P$, and $16P$, respectively. The complexity of those point operations can be evaluated by the algorithms given in the previous sections. When formulae for direct computation are not given, D_4 should be replaced by $2D_2$, and so on. For scalar multiplication, we used the Algorithm 4, which is based on the sliding window method with NAF representation for a scalar.

The complexity comparisons in the case of 160 bit are described below. We again assume that one squaring has complexity $S = 0.8M$. The comparison in the case of larger size may be obtained in the same way.

1. **Affine using our direct computation with $k = 4$ vs. affine using usual double-add method:**
   Our method has complexity $1270M + 91I$. Usual method has complexity $679M + 196I$. Break-even point can be derived as $5.6M < I$. Therefore our method may be faster in most implementations.
2. **Affine using our direct computation with $k = 2$[†] vs. affine using Müller's formulae:**
   Our method has complexity $1317M + 125I$. Müller's method has complexity $1559M + 125I$. Both method can quadrupled point directly, therefore have the same number of field inversion. On the other hand, the number of field multiplication was significantly reduced.
3. **Affine using our direct computation with $k = 4$ vs. modified jacobian:**
   Our method has complexity $1270M + 91I$ as described above. The scalar multiplication in modified jacobian coordinates has complexity $1807M$. Break-even point can be derived as $5.9M < I$. Therefore in most implementations, scalar multiplication in jacobian coordinates may be faster than that in affine coordinates with direct computations.

As we have described previously, one optimal way to speed-up scalar multiplication is to mix several coordinates. Which coordinates should be used is remained as a further work.

7. Implementation

In this section, we implement our methods which have been given in previous sections. We implement 6 elliptic curves over $\mathbb{F}_p$ with $\log_2 p = 160$, 192, 224, 256, 384 or 521. We call these curves as P160, P192, P224, P256, P384 or P521, respectively. Their parameters of the curves can be found in Appendix B. All these curves have prime order. Moreover, the coefficients a of the curves are chosen as equal to 1 or -3 for efficient implementation. However, in our implementation, for the purpose of making comparison with general curves, we apply general algorithms that do not depend on a parameter a.

The platform was a Pentium II 400 MHz, Windows NT 4.0 and Watcom 11.0. Programs were written in assembly language for multi-precision integer operations, which may be time critical in implementation, or in ANSI C language for other operations.

7.1 Inversion

We applied Lehmer's method [10] to modular inversions. In affine coordinate computations, modular inversion in $\mathbb{F}_p$ is much expensive than modular multiplication. Therefore, it is important to apply an efficient method to modular inversion.

Remark 1: We applied Lehmer's method to field inversions in our implementation. Although several methods of inversions have been developed, it is not have been known which method may exactly give a practical speed-up for the range of integers we are interested in (up to 521-bit). If we have an efficient method for inversion, elliptic curve cryptosystems with affine coordinates may be faster than those with projective or weighted projective coordinates.

In our implementation, the ratio of speed between a multiplication and an inversion I/M is approximately equals to 25 in cases of $\log_2 p = 160$. Our I/M tends to decreases with larger $\log_2 p$.

[†]In [14] formulae for $k > 2$ does not given. Therefore we compare with $k = 2$.

Table 3 Timings of a point addition and direct doublings in msec (Pentium II 400 MHz).

Curves	Affine					Weighted Projective			Modified Jacobian	
	Add.	$2P$	$4P$	$8P$	$16P$	Add.	$2P$	$4P$	Add.	$2P$
P160	0.140	0.141	0.194	0.243	0.291	0.0582	0.0373	0.0729	0.0714	0.0355
P192	0.151	0.156	0.229	0.284	0.337	0.0827	0.0538	0.105	0.103	0.0502
P224	0.158	0.167	0.256	0.325	0.390	0.0963	0.0630	0.127	0.125	0.0563
P256	0.172	0.175	0.279	0.345	0.415	0.110	0.0713	0.139	0.142	0.0623
P384	0.219	0.229	0.392	0.497	0.616	0.198	0.123	0.238	0.244	0.108
P521	0.358	0.378	0.695	0.892	1.179	0.347	0.214	0.407	0.423	0.180

Table 4 Timings of a scalar multiplication of a random point in msec (Pentium II 400 MHz).

Curves	Affine		Weighted Projective		Modified Jacobian
	Usual method	Proposed method	Usual method	Proposed method	Usual method
P160	26.8	18.4	9.1	7.9	8.2
P192	35.2	22.5	15.5	13.1	13.2
P224	42.1	27.8	20.6	18.2	18.3
P256	50.9	34.5	25.7	22.3	22.8
P384	98.3	65.7	68.2	58.5	59.0
P521	215	148	163	139	140

7.2 Timings

Table 3 shows timings of point additions and direct computations of 2^kP in our implementation. Table 4 shows timings of scalar multiplication. In Table 4, "*usual method*" means a sliding signed binary window method by additions and doublings. These two tables also show timings for modified jacobian coordinates. In the case of affine coordinates, we used direct computation of $4P$, $8P$ and $16P$. In the case of weighted projective coordinates, we used direct computation of $4P$.

As can be seen from Table 4, in the case of affine coordinates, in which direct computation of 2^kP with $2 \leq k \leq 4$ was used, we achieved a 45 percent performance increase in the scalar multiplication of the elliptic curve of size 160-bit. In the case of weighted projective coordinates, in which direct computation of 2^kP with $k = 2$ was used, we achieved a 15 percent performance increase in the scalar multiplication of the elliptic curve of size 160-bit.

In the case of modified jacobian coordinates, we have not yet developed formulae for direct computation of several doublings. Our experimental results suggest that direct computation provides a performance increase. Therefore, once efficient formulae are available, modified jacobian coordinates may provide superior results.

The addition in modified jacobian coordinates is expensive compared to that in weighted projective coordinates. Therefore, the scalar multiplications in weighted projective coordinate. with direct computation are slightly faster than those in modified jacobian coordinates.

We have one other observation to make from Table 4. In most implementation of curves of size 160-bit, as authors knowledge, scalar multiplication in projective or weighted projective coordinates are faster than those in affine coordinates. Our implementation supports that facts. However, in our implementation with new method, when $\log_2 p$ is relatively large (P384 and P521), the speed of scalar multiplication in affine coordinates tend to close to those in weighted projective coordinates. As previously stated, the answer to the question "which coordinate system is faster" strongly depends on the ratio I/M. In our implementation, I/M decreases with larger $\log_2 p$. This is the main reason that the speed of scalar multiplication in affine coordinates close to that in weighted projective coordinates when p is large.

8. Conclusions and Further Work

In this paper, we have constructed formulae for computing 2^kP directly from $P \in E(\mathbb{F}_p)$, without computing the intermediate points $2P,\ 2^2P, \cdots, 2^{k-1}P$ in terms of affine and weighted projective coordinates. We showed that our algorithms are more efficient than k separate doublings and lead to a running time improvement of scalar multiplication. Combining our method with the mixed coordinate strategy proposed in [2] may increase elliptic scalar multiplication performance.

References

[1] D.V. Chudnovsky and G.V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests," Advances in Applied Math., vol.7, pp.385–434, 1986.
[2] 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.
[3] D.M. Gordon, "A survey of fast exponentiation methods," J. Algorithms, vol.27, pp.129–146, 1998.
[4] 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.
[5] Y. Han and P.C. Tan, "Direct computation for elliptic curve cryptosystems," Pre-proc. CHES'99, pp.328–340, Springer-Verlag, 1999.
[6] IEEE P1363/D11 (Draft Version 11), 1999. `http://grouper.ieee.org/groups/1363/`
[7] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara, "Fast implementation of public-key cryptography on a DSP TMS320C6201," Cryptography Hardware and Embedded Systems, LNCS, vol.1717, pp.61–72, Springer-Verlag, 1999.
[8] N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation, vol.48, pp.203–209, 1987.
[9] 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.
[10] D.H. Lehmer, "Euclid's algorithm for large numbers," American Mathematical Monthly, vol.45, pp.227–233, 1938.
[11] V. Miller, "Uses of elliptic curves in cryptography," Advances in Cryptology—CRYPTO'85, LNCS, vol.218, pp.417–426, Springer-Verlag, 1986.
[12] A. Miyaji, T. Ono, and H. Cohen, "Efficient elliptic curve exponentiation (I)," IEICE Technical Report, ISEC97-16, 1997.
[13] A. Miyaji, T. Ono, and H. Cohen, "Efficient elliptic curve exponentiation," Advances in Cryptology—ICICS'97, LNCS, vol.1334, pp.282–290, Springer-Verlag, 1997.
[14] V. Müller, "Efficient algorithms for multiplication on elliptic curves," Proc. GI-Arbeitskonferenz Chipkarten 1998, TU München, 1998.
[15] NIST, "Recommended elliptic curves for federal government use," `http://csrc.nist.gov/encryption/`, 1999.
[16] J.A. Solinas, "An improved algorithm for arithmetic on a family of elliptic curves," Advances in Cryptology—CRYPTO'97, LNCS, vol.1294, pp.357–371, Springer-Verlag, 1997.
[17] E. De Win, S. Mister, B. Preneel, and M. Wiener, "On the performance of signature schemes based on elliptic curves," Algorithmic Number Theory III, LNCS, vol.1423, pp.252–266, Springer-Verlag, 1998.

Appendix A: Computational Complexity of the Direct Computations

In this Appendix we give proof of Theorems 1 and 2. In the following proofs, we will ignore the cost of a field addition and a subtraction, as well as the cost of a multiplication by small constants.

A.1 Theorem 1

In `Step 1` of Algorithm 1, one squaring is performed for the computation of x_1^2. The complexity of `Step 1` is S.

In `Step 2`, following computations are performed $k-1$ times. Firstly we perform 3 squarings for the computations of C_{i-1}^2, C_{i-1}^4 and B_{i-1}^2. Secondly we perform one multiplication for the computation of $A_{i-1}C_{i-1}^2$. Then we will obtain A_i. Thirdly we perform one squaring for the computation of A_i^2. If $i = 2$ then we perform one multiplication for the computation of aC_1^4 and set the result as U, else if $i > 2$ then we perform one multiplication for the computation of UC_{i-1}^4 and set the result as U. U will equal to $a(\prod_{j=1}^{i-1} C_j)^4$. Then we will obtain B_i. Finally we perform one multiplication for the computation of $B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$. Then we will obtain C_i. The complexity of `Step 2` is $4(k-1)S + 3(k-1)M$.

In `Step 3`, we first perform 2 squarings for the computation of C_k^2 and B_k^2. Next we perform one multiplication for the computation of $A_kC_k^2$. Then we will obtain D_k. The complexity of `Step 3` is $2S + M$.

In `Step 4`, we first perform one squaring for the computation of C_k^4. Secondly we compute $\prod_{i=1}^{k} C_i$ for which we perform $k-1$ multiplications. Thirdly we perform one inversion for the computation of $(2^k \prod_{i=1}^{k} C_i)^{-1}$ and set the result as T. Next we perform one squaring for the computation of T^2. Next we perform one multiplication for the computation of $(B_k^2 - 8A_kC_k^2)T^2$. Then we get x_{2^k}. Next we perform one multiplication for the computation of B_kD_k. Finally we perform 2 multiplication for the computation of $(8C_k^4 - B_kD_k)T^2T$. Then we get y_{2^k}. The complexity of `Step 4` is $2S + (k-1)M + 4M + I$.

By the above computation, the complexity of Algorithm 1 can be evaluated as $(4k+1)S+(4k+1)M+I$.

A.2 Theorem 2

In `Step 1`, `2` and `3` of Algorithm 2, we perform computations in the same way as Algorithm 1. Therefore, the complexity amount of `Step 1`, `2` and `3` is $S + 4(k-1)S + 3(k-1)M + 2S + M$.

In `Step 4`, we first perform one squaring for the computation of C_k^4. Next we compute $\prod_{i=1}^{k} C_i$ for which we perform $k-1$ multiplications. Finally we perform one multiplication for the computation of B_kD_k. Then we get X_{2^k}, Y_{2^k} and Z_{2^k}. The complexity of `Step 4` is $S + (k-1)M + M$.

Therefore, the complexity of Algorithm 2 can be evaluated as $4kS + (4k-2)M$.

Appendix B: Curves

In this Appendix, the curves that we have implemented are given. P192, P224, P256, P384 and P512 are given in [15] as recommended curves.

P160

$p = 736459809436275298942988210292996840747673059329$
$a = 1$
$\sharp E(\mathbb{F}_p) = 736459809436275298942987873098523465705697$
104451

Yasuyuki Sakai received the B.S. and M.S. degree from Waseda University in 1990 and 1992, respectively. He is a researcher of Information Technology R&D Center at Mitsubishi Electric Corporation. His current research interests are in cryptography and information security. He is a member of IPSJ and SITA.

Kouichi Sakurai received the B.S. degree in mathematics from Faculty of Science, Kyushu University and the M.S. degree in applied science from Faculty of Engineering, Kyushu University in 1986 and 1988, respectively. He had been engaged in the research and development on cryptography and information security at Computer & Information Systems Laboratory at Mitsubishi Electric Corporation from 1988 to 1994. He received the Dr. degree in engineering from Faculty of Engineering, Kyushu University in 1993. Since 1994, he has been working for Department of Computer Science of Kyushu University as an associate professor. His current research interests are in cryptography and computational complexity. Dr. Sakurai is a member of the Information Processing Society of Japan, the Mathematical Society of Japan and the International Association for Cryptologic Research.