Int. J. Nonlinear Anal. Appl. 9 (2018) No. 1, 161-174
ISSN: 2008-6822 (electronic)
http://dx.doi.org/10.22075/ijnaa.2018.11642.1581

Efficient elliptic curve cryptosystems

Kamal Darweesh[a], Mohammad Saleh[b,*]

[a]Applied Mathematics Department, Palestine Technical University–Kadoorie, Tulkarm, Palestine
[b]Mathematics Department, Birzeit University, P.O. Box 14, Palestine

(Communicated by M. Eshaghi)

Abstract

Elliptic curve cryptosystems (ECC) are new generations of public key cryptosystems that have a smaller key size for the same level of security. The exponentiation on elliptic curve is the most important operation in ECC, so when the ECC is put into practice, the major problem is how to enhance the speed of the exponentiation. It is thus of great interest to develop algorithms for exponentiation, which allow efficient implementations of ECC. In this paper, we improve efficient algorithm for exponentiation on elliptic curves defined over $\mathbf{F}_p$ in terms of affine coordinates. The algorithm computes $2^{n_2}(2^{n_1}P+Q)$ directly from random points P and Q on an elliptic curve, without computing the intermediate points. Moreover, we apply the algorithm to exponentiation on elliptic curves with width–w Mutual Opposite Form (wMOF) and analyze their computational complexity. This algorithm can speed up the wMOF exponentiation of elliptic curves of size 160–bit about (21.7%) as a result of its implementation with respect to affine coordinates.

Keywords: cryptography; elliptic curves; affine coordinates.
2010 MSC: 94A60.

1. Introduction

Elliptic curve cryptosystems, which were suggested independently by Miller [7] and Koblitz [5], are new generation of public key cryptosystems that have smaller key sizes for the same level of security.

The elliptic curve cryptographic operations, like encryption/ decryption schemes generation/ verification signature, require computing of exponentiation on elliptic curve. The computational performance of elliptic curve cryptographic protocol such as Diffie–Hellman [3]. Key Exchange protocol strongly depends on the efficiency of exponentiation, because it is the costliest operation. Therefore, it is very attractive to speed up exponentiation by providing algorithms that allow efficient implementations of elliptic curve cryptosystems [1, 4, 6, 8, 9, 12].

*Corresponding author

Email addresses: kdarweesh.scom@gmail.com (Kamal Darweesh), msaleh@birzeit.edu (Mohammad Saleh)

Received: April 2018 *Revised:* May 2018

There are typical methods for exponentiation such as binary methods and windowing methods [9]. These methods can speed up exponentiation by reducing additions, where addition of two points and doubling of two points are performed repeatedly.

One of the efficient windowing methods is wMOF [11]. It is a base–2 representation which provide the minimal hamming weight of exponent. Its great advantage is that it can be generated from left–to–right which means, that the recoding doesn't have to be done in a separate stage, but can be performed on–the–fly during the evaluation. As a result, it is no longer necessary to store the whole recoded exponent, but only small parts at once.

Another approach to speed up exponentiation is by increasing the speed of doublings. One method to speed the doublings is direct computation of several doubling, which computes 2^nP directly from $P \in E(\mathbf{F}_q)$, without computing intermediate points $2P, 2^2P, \ldots, 2^{n-1}P$. Sakai and Sakurai [12] proposed formulae for computing 2^nP directly ($\forall n \geqslant 1$) on $E(\mathbf{F}_p)$ in terms of affine coordinates. Since modular inversion is more expensive than multiplication, their formulae requires only one inversion for computing 2^nP instead of n inversions in usual add–double method.

In this paper, we improve efficient algorithm for exponentiation on elliptic curve defined over $\mathbf{F}_p$ in terms of affine coordinates. We construct efficient formulae to compute $2^{n_2}(2^{n_1}P + Q)$ directly from $P, Q \in E(\mathbf{F}_p)$, without computing intermediate points $2P, 2^2P, \ldots, 2^{n_1}P$, $2(2^{n_1}P + Q), \ldots, 2^{n_2-1}(2^{n_1}P + Q)$ where $n_1 \geqslant 1$. Our formulae have computational complexity $(4n + 10)M + (4n + 6)S + I$, where M, S and I denote multiplication, squaring and inversion respectively in $\mathbf{F}_p$, and $n = n_1 + n_2$.

Moreover, we show in which way this new algorithm for direct computing $2^{n_2}(2^{n_1}P + Q)$ can be combined with wMOF exponentiation method [11]. We also implement wMOF exponentiation with and without these formulae and discuss the efficiency. The result of this implementation shows that 21.7% speed increase in wMOF exponentiation with these formulae on elliptic curve of size 160–bit.

Let $\mathbf{F}_p$ denotes a prime finite field with p elements. We consider an elliptic curve E given by Weierstrass non–homogeneous equation: $E : y^2 = x^3 + ax + b$, where $a, b \in \mathbf{F}_p$, $p > 3$, and $4a^3 + 27b^2 \neq 0$ (i.e. E is smooth). Let

$$P_1 = (x_1, y_1),\ P_1' = (x_1', y_1'),$$
$$P_{2^n} = 2^nP_1 = (x_{2^n}, y_{2^n}) \in E(\mathbf{F}_p).$$

Let the elliptic curve point addition and doubling be denoted by ECADD and ECDBL, respectively. Let M, S and I denote multiplication, squaring and inversion, respectively in $\mathbf{F}_p$, where $S = 0.8M$, as it is customary nowadays.

2. Previous work

In this section, we summarize the known algorithms for point addition, point doublings, and direct doublings.

2.1. Point addition

In terms of affine coordinates, point addition can be computed as follows:
Let $P_1 = (x_1, y_1)$, and $Q = (x, y) \neq O$ where O denotes the point at infinity, then $P' = (x', y')$ can be computed as follows

$$\begin{aligned} x' &= \lambda^2 - x_1 - x; \\ y' &= \lambda(x_1 - x') - y_1; \\ \lambda &= \frac{(y - y_1)}{(x - x_1)}. \end{aligned}$$

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

2.2. Point doubling

In terms of affine coordinates, point addition can be computed as follows: Assume Let $P1 = (x_1, y_1) \neq O$ where O denotes the point at infinity, then $2P = P_2 = (x_2, y_2)$ can be computed as follows

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

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

2.3. Direct doubling

One method to increase the speed of doublings is direct computation of several doublings, which can compute $2^n P$ directly from $P \in E(\mathbf{F}_q)$, without computing the intermediate points $2P, 2^2P, \ldots, 2^{n-1}$ (see [12]).

Guajardo and Paar [4] suggested increase doubling speed by formulating algorithms for direct computation of $4P, 8P$, and $16P$ on elliptic curves over $\mathbf{F}_{2^m}$ in terms of affine coordinates.

Sakai and Sakurai [12] proposed formulae for computing $2^n P$ directly ($\forall n \geqslant 1$) on $E(\mathbf{F}_p)$ in terms of affine coordinates. These formulae require only one inversion for computing $2^n P$ instead of n inversions in regular add–double method.

In affine coordinate, direct computation requires only one inversion for computing $2^n P$ instead of n inversions in regular add–double method. Therefore direct computation of several doublings may be effective in elliptic curve exponentiation in terms of affine coordinate, since modular inversion is more expensive than modular multiplication [12].

3. Direct Computation of $2^{n_2}(2^{n_1}P + Q)$ in affine coordinate

In this section, we derive formulae for computing $2^{n_2}(2^{n_1}P + Q)$ directly from a given points $P, Q \in E(\mathbf{F}_p)$ without computing the intermediate points $2P, 2^2P, \ldots, 2^{n_1}P, 2(2^{n_1}P + Q), \ldots, 2^{n_2-1}(2^{n_1}P + Q)$, where $n_1 \geq 1$, in terms of affine coordinate. These formulae can work with wMOF exponentiation method [11].

We begin by constructing formulae for small n_1, n_2, then we will construct algorithm for general n_1, n_2.

As an example, let $n_1 = 2$, $n_2 = 1$, and $P_1 = (x_1, y_1)$, $Q = (x, y)$, $P_1'(x_1', y_1') \in E(\mathbf{F}_p)$. Then for an elliptic curve with Weierstrass form in terms of affine coordinates $P_2' = 2P_1' = 2(4P_1 + Q) = (x_2', y_2')$ can be computed as the following:

1) Computing $4P_1$ as in [12] $4P_1 = P_4 = (x_4, y_4)$ can be computed as follows:
   Let

$$\begin{aligned} C_0 &= y_1; \\ A_0 &= x_1; \\ B_0 &= 3x_1^2 + a; \\ A_1 &= B_0^2 - 8A_0C_0^2; \\ C_1 &= -8C_0^4 - B_0(A_1 - 4A_0C_0^2); \\ B_1 &= 3A_1^2 + 16aC_0^4; \\ A_2 &= B_1^2 - 8A_1C_1^2; \\ C_2 &= -8C_1^4 - B_1(A_2 - 4A_1C_1^2). \end{aligned}$$

Then $4P_1 = P_4 = (x_4, y_4)$ can be computed as follows:

$$x_4 = \frac{A_2}{(4C_0C_1)^2}, \tag{3.1}$$

$$y_4 = \frac{C_2}{(4C_0C_1)^3}. \tag{3.2}$$

2) Computing $(4P_1 + Q)$
Assume $4P_1 = (x_4, y_4) \neq -Q$, recall from Section 2, the point addition then $P_1' = (x_1', y_1') = (4P_1 + Q)$ in term of affine coordinates, can be computed as follows:

$$x_1' = \lambda^2 - x - x_4, y_1' = \lambda(x - x_1') - y. \tag{3.3}$$

Substituting x_4, y_4 in $\lambda = \frac{y_4 - y}{x_4 - x}$ we get

$$\lambda = \frac{\frac{C_2}{(4C_0C_1)^3} - y}{\frac{A_2}{(4C_0C_1)^2} - x}. \tag{3.4}$$

Now let

$$T = C_2 - (4C_0C_1)^3 y,$$
$$S = A_2 - (4C_0C_1)^2 x.$$

Then we get:

$$\lambda = \frac{T}{(4C_0C_1)S}. \tag{3.5}$$

Substituting, λ and x_4 into the expression for x_1', we find

$$x_1' = \frac{T^2 - S^2(A_2 + (4C_0C_1)^2 x}{(4C_0C_1)^2 S^2}. \tag{3.6}$$

Let $M = A_2 + (4C_0C_1)^2 x$, we get:

$$x_1' = \frac{T^2 - MS^2}{(4C_0C_1)^2 S^2}. \tag{3.7}$$

Let $A_0' = T^2 - MS^2$ and, substituting λ, and x_1' into the expression for y_1'. Then we get:

$$y_1' = \frac{-(4C_0C_1)^3 yS^3 - T(A_0' - (4C_0C_1)^2 xS^2)}{(4C_0C_1)^3 S^3}. \tag{3.8}$$

Let

$$C_0' = -(4C_0C_1)^3 yS^3 - T(A_0' - (4C_0C_1)^2 xS^2).$$

Then we get:

$$y_1' = \frac{C_0'}{(4C_0C_1)^3 S^3}. \tag{3.9}$$

3) Computing

$$2(4P_1 + Q) = 2P_1'$$

Recall from Section 2, the point doubling, then $2P_1' = P_2' = (x_2', y_2')$ in term of affine coordinates, can be computed as follows:

$$\lambda = \frac{3{A_0'}^2 + a(4C_0C_1)^4S^4}{2C_0'(4C_0C_1)S}. \tag{3.10}$$

Now, let $B_0' = 3{A_0'}^2 + a(4C_0C_1)^4S^4$ and, substituting λ, and x_1' into the expression for x_2'. Then we find:

$$x_2' = \frac{{B_0'}^2 - 8A_0'{C_0'}^2}{(2C_0')^2(4C_0C_1)^2S^2}. \tag{3.11}$$

Let $A_1' = {B_0'}^2 - 8A_0'{C_0'}^2$, and substituting λ, y_1', x_1' and x_2' into the expression for y_2'. Then we find

$$y_2' = \frac{-8{C_0'}^4 - B_0'(A_1' - 4A_0'{C_0'}^2)}{(2C_0')^3(4C_0C_1)^3S^3}. \tag{3.12}$$

Let $C_1' = -8{C_0'}^4 - B_0'(A_1' - 4A_0'{C_0'}^2)$. Then we get finally:

$$y_2' = \frac{C_1'}{(2C_0')^3(4C_0C_1)^3S^3}. \tag{3.13}$$

The formulae above have computational complexity $18S + 22M + I$.

3.1. The formulae computing $2^{n_2}(2^{n_1}P + Q)$ in affine coordinate

From the above formulae for direct computing $2(4P_1 + Q)$, we can easily obtain general formulae that allow direct computing $2^{n_2}(2^{n_1}P + Q)$ for $n_1 \geqslant 1$. Algorithm 1 describes these formulae.

Theorem 3.1 (bellow) describes the computational complexity of this formula.

Theorem 3.1. *In terms of affine coordinates, there exits an algorithm that computes $2^{n_2}(2^{n_1}P+Q)$ at most $[4(n+2)+2]M$, $[4(n+1)+2]S$, and I in $\mathbf{F}_p$ for any point $P, Q \in E(\mathbf{F}_p)$ where M, S and I denote multiplication, squaring and inversion respectively and $n = n_1 + n_2$.*

Proof . The complexity of step 1 and step 2 (the same as in ([12] Algorithm 1) involve $(2M + 3S)n_1 + (M + S)(n_1 - 1) + S$.
In step 3, we first compute $\prod_{i=0}^{n_1-1} C_i$ which takes $n_1 - 1$ multiplication. Secondly, we perform one squaring to compute $(2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2$. Next, we perform one multiplication to compute $(2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2x$. Then we obtain N and V. Next, we perform two multiplications, one multiplication to compute $(2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2y$ and other to compute $(2^{n_1}\prod_{i=0}^{n_1-1} C_i)(2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2y = (2^{n_1}\prod_{i=0}^{n_1-1} C_i)^3y$. Then we obtain W. Third, we perform two squaring to compute W^2, N^2, and one multiplication to compute VN^2. Then we obtain A_0'. Fourth, we perform one multiplication to compute $(2^{n_1}\prod_{i=0}^{n_1-1} C_i)N$. Then we obtain Z. Next, we perform two squaring to compute Z^2, Z^4 and one multiplication to compute Z^3. Next, we perform two multiplications to compute Z^2x, Z^3y. Finally, we perform one multiplication to compute $W(A_0' - Z^2x)$. Then we obtain C_0'. The complexity of step 3 involves $(n_1 - 1)M + 9M + 5S$. In step 4 we perform one squaring to compute ${A_0'}^2$. Next we perform one multiplication to compute aZ^4 where Z^4 is computed in step 3. Then we obtain B_0' . The complexity of step 4 involve $M + S$ and the complexity of step 4 involves $(2M + 3S)n_2 + (M + S)(n_2 - 1)$ as step 2.

Algorithm 1 Direct computation of $2^{n_2}(2^{n_1}P + Q)$ in affine coordinate, where $n_1 \geqslant 1$, and $P, Q \in E(\mathbf{F}_p)$

```
**Input:** $p_1 = (x_1, y_1), Q = (x, y) \in E(\mathbf{F}_p)$
**Output:** $P'_{2^{n_2}} = 2^{n_2}P' = 2^{n_2}(2^{n_1}P_1 + Q) = (x'_{2^{n_2}}, y'_{2^{n_2}}) \in E(\mathbf{F}_p)$
1. Compute $A_0$ and $C_0$ and $B_0$
2. For i from 1 to n1 Compute Ai, Ci, for i from 1 to n1 -1 Compute Bi
**for** $i = 1$ to $n_1$ **do**
    $A_i = B_{i-1}^2 - 8A_{i-1}C_{i-1}^2$
    $C_i = -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$
**end for**
**for** $i = 1$ to $n_1 - 1$ **do** $B_i = 3A_i^2 + 16^i a(\prod_{j=0}^{i-1} C_j)^4$
3. Compute the $N, V, W, Z$ then,
$N \leftarrow A_{n_1} - (2^{n_1} \prod_{i=0}^{n_1-1} C_i)^2 x$
$V \leftarrow A_{n_1} - (2^{n_1} \prod_{i=0}^{n_1-1} C_i)^2 x$
$W \leftarrow C_{n_1} - (2^{n_1} \prod_{i=0}^{n_1-1} C_i)^3 y$
$Z \leftarrow (2^{k_1} \prod_{i=0}^{k_1-1} C_i)N$
 $A'_0 = W^2 - VN^2$
 $C'_0 = -Z^3 y - W(A'_0 - Z^2 x)$
4. if $(n_2 > 0)$
Compute $B'_0 = 3{A'_0}^2 + aZ^4$
For i from 1 to n2 Compute , for i from 1 to n2 -1 Compute
**for** $i = 1$ to $n_2$ **do**                                    ▷ Compute $A'$ and $C'$ values
    $A'_i = {B'}_{i-1}^2 - 8A'_i{C'}_i^2$
    $C'_i = -8{C'}_{i-1}^4 - B'_{i-1}(A'_i - 4A'_i{C'}_i^2)$
**end for**
**for** $i = 1$ to $n_2 - 1$ **do** $B'_i = 3{A'}_{i-1}^2 + 16^i\, a\, Z^4(\prod_{j=0}^{i-1} {C'}_j)^4$          ▷ Compute $B'$ values
Compute $Z \leftarrow Z(2^{n_2} \prod_{i=0}^{n_2-1} {C'}_i)$
Compute $x'_{2^{n_2}} \leftarrow \frac{A'_{n_2}}{Z^2}$ , $y'_{2^{n_2}} \leftarrow \frac{C'_{n_2}}{Z^3}$
```

In step 4 we compute $\prod_{i=0}^{n_2-1} C'_i$ which takes $n_2 - 1$ multiplications. Secondly, we perform one multiplication to compute $Z(2^{n_2} \prod_{i=0}^{n_2-1} C'_i)$. Then we obtain new value for Z. the complexity of step 4 involves n_2M. Hence, the complexity of step 4 involves $4n_2M + 4n_2S$.
In step 5, we perform one inversion to compute Z^{-1} and the result is set to T. Next, we perform one squaring to compute T^2. Next, we perform one multiplication to compute $A'_{n_2}T^2$. Then we obtain $x'_{2^{n_2}}$. Finally we perform two multiplications to compute $C'_{n_2}T^2T$. Then we obtain $y'_{2^{n_2}}$.

The complexity of step 5 involves $3M + S + I$. So the complexity of above computations involve $[4(n+2)+2]M, [4(n+1)+2]S$, where $n = n_1 + n_2$. □

3.2. Complexity comparison

For application in practice it is highly relevant to compare the complexity of $2^{n_2}(2^{n_1}P + Q)$ our algorithm for direct computing of $2^{n_2}(2^{n_1}P + Q)$ with regular add–double method which requires $(n_1 + n_2)$ separated doublings and one addition, and with Sakai–Sakuri algorithm [12] for computing $2^{n_1+n_2}P$ and $2^{n_2}Q$. The performance of the new method depends on the cost factor of one inversion relatively to the cost of one multiplication. For this purpose, we introduce, as [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 multiplication needed per inversion.

In general let $n = n_1 + n_2$, let us denote the direct computing of $2^{n_2}(2^{n_1}P + Q)$ by symbol DECDBL(n). Then our formulae can outperform the regular double and add algorithm if the following relation to hold:

$$Cost(separatenECDBL + ECADD) > Cost(DECDBL(n))$$

Table 1: Complexity comparison: direct computing of $2^{n_2}(2^{n_1}P + Q)$ vs. Individual $(n_1 + n_2)$ doublings and one addition.

Calculation $2^{n_2}(2^{n_1}P + Q)$ where	Method	Complexity			Break–Even Point
		S	M	I	
$n_1 + n_2 = 3$	DECDBL(3)	18	22	1	$7.6M < I$
	3 doublings + 1 addition	8	7	4	
$n_1 + n_2 = 4$	DECDBL(4)	22	26	1	6.6 M ¡ I
	4 doublings + 1 addition	10	9	5	
$n_1 + n_2 = 5$	DECDBL(5)	26	30	1	6 M ¡ I
	5 doublings + 1 addition	12	11	6	
$n_1 + n_2 = n$	DECDBL(n)	4n+6	4n+10	1	$\frac{(3.6n+12)}{n}M < I$
	n doublings + 1 addition	2n+1	2n+2	n+1	

Ignoring squarings and additions and expressing the Cost function in terms of multiplications and inversions, we have:

$$(2nM + 2nS + nI + 2M + S + I) > (4(n+2)M + 4(n+1)S + 2M + 2S + I).$$

We define $r = I/M$ (the ratio of speed between a multiplication and inversion), and assume that one squaring has complexity $S = 0.8M$ [12]. We also assume that the cost of field addition and multiplication by small constants can be ignored. One can rewrite the above expressions as:

$$nrM > (2nM + 8M + 1.6nM + 4M).$$

Solving for r in terms of M one obtains:

$$r > \frac{(3.6n + 12)}{n}.$$

Table 2: Complexity comparison: direct computing of $2^{n_2}(2^{n_1}P+Q)$ vs. direct computing of $2^{n_1+n_2}P$ and $2^{n_2}Q$.

Calculation $2^{n_2}(2^{n_1}P+Q)$ where	Method	Complexity			Break–Even Point
		S	M	I	
$n_1=4, n_2=0$	DECDBL(4)	22	26	1	$4.2M < I$
	Sakai–Sakuri algorithm	19	20	3	
$n_1=3, n_2=2$	DECDBL(4)	22	26	1	$0.6M < I$
	Sakai–Sakuri algorithm	23	24	3	
$n_1=2, n_2=2$	DECDBL(4)	22	26	1	$-3M < I$
	Sakai–Sakuri algorithm	27	28	3	
$n_1+n_2=n$	DECDBL(4)	$4n+6$	$4n+10$	1	$\frac{8.4-7.2n_2}{2}M < I$
	Sakai–Sakuri algorithm	$4(n+n_2)+3$	$4(n+n_2+1)$	3	

As we see from Table 1, if a field inversion has complexity $I > 7.6M$, direct computation of 3 doublings and one addition may be more efficient than 3 separate doubling and one addition.

Moreover, our algorithm for direct computing of $2^{n_2}(2^{n_1}P+Q)$ can outperform Sakai–Sakuri algorithm for computing $2^{n_1+n_2}P$ and $2^{n_2}Q$ if: Cost(direct computing of $2^{n_1+n_2}P$ and direct computing of $2^{n_2}Q$ and then simply adding the two) > Cost(DECDBL(n_1+n_2)).

In case, we ignore squaring and additions and expressing the Cost function in terms of multiplications and inversions, we have:

$$[(4n+1)M+(4n+1)S+(4n2+1)M+(4n2+1)S+3I+2M+S] > \\ [4(n+2)M+4(n+1)S+2M+2S+I].$$

After simplification we can rewrite the above expressions as:

$$2I > 6M+3S-4n2S-4n2M$$

Solving for r in terms of M one obtains:

$$r > \frac{8.4-7.2n_2}{2}.$$

As we see from Table 2, if a field inversion has complexity $I > 4.2M$, direct computation of 4 doublings and one addition by using our algorithm is more efficient than 4 doublings by using Sakai–Sakuri algorithm and then performing one addition. Also, it is clear from the table and the above discussion that DECDBL(n) is different from the Sakai–Sakuri algorithm for computing $2^{n_1+n_2}P$ and $2^{n_2}Q.2^{n_2}(2^{n_1}P+Q)$.

3.3. Exponentiation with direct computation of $2^{n_2}(2^{n_1}P+Q)$

By using our previous formulae for direct computation of $2^{n_2}(2^{n_1}P+Q)$, where $n_1 \geqslant 1$, and $P, Q \in E(\mathbf{F}_p)$, we can improve algorithm B.1 [11] for elliptic curve exponentiation with wMOF by change the step 3.2 of algorithm B.1 [11] with a new step that compute $2^{n_2}(2^{n_1}P + Q)$ directly as in the following algorithm.

Algorithm 2 Exponentiation with wMOF Using Direct Computation of $2^{n_2}(2^{n_1}P + Q)$

```
**Input:** a non–zero t–bit binary string $k, P \in E(\mathbf{F}_p)$, and the multiple of the point $P, \gamma_{0\cdots tw}$ and $\xi_{0\cdots tw}$, the precomputed table look–up .
**Output:** exponentiation $kP$.
$i \leftarrow t$ ; $Q \leftarrow O$
while $i \geqslant 0$ do
    if $(k_i\ XOR\ k_{i-1}) = 0$ then
        $Q \leftarrow ECDBL(Q)$
        $i \leftarrow i - 1$
    else
        $index \leftarrow ((k >> (i - w))\&(2^{w+1} - 1)) - 2^{w-1}$
        if $i < w$ then
            $Q \leftarrow 2^{i-(w-\xi_{index})+1}(2^{(w-\xi_{index}}Q + \gamma_{index}P)$
        else
            $Q \leftarrow 2^{\xi_{index}}(2^{(w-\xi_{index}}Q + \gamma_{index}P)$
            $i \leftarrow i - 1$
        end if else
    end if else
end While
if $i = 0$ then
    $Q \leftarrow ECDBL(Q)$
    if $k_0 = 1$ then $Q \leftarrow ECADD(Q, -P)$
return Q
```

In algorithm 2, for each window width w of wMOF, Step 3.1 performs direct computation of $2^{i-(w-\xi_{index})+1}(2^{(w-\xi_{index}}Q + \gamma_{index}P)$ if $(i < w)$ otherwise Step 3.2 performs direct computations of $2^{\xi_{index}}(2^{(w-\xi_{index}}Q+\gamma_{index}P)$ if $(i \geqslant w)$, where $\xi_{index} = 0, 1, \ldots, w-1$, $\gamma_{index}P = \{\pm 1, \pm 3, \ldots, \pm(2^{w-1}-1)\}$.

3.4. Complexity analysis of the wMOF method

In this subsection, we perform an analysis of wMOF method when it used in conjunction with the $2^{n_2}(2^{n_1}P + Q)$ formulae. In addition, we compare the complexity of wMOF method, with and without formulae. Moreover we derive an expression that predicts the theoretical improvement of the wMOF method by using the formulae, in terms of the ratio between inversion and multiplication times. Theorem 3.2 describes the complexity of algorithm B.1 [11] for computing exponentiation with wMOF.

Theorem 3.2. *In terms of affine coordinate, let $P \in E(\mathbf{F}_p)$, t–digits exponent in wMOF, then the complexity of algorithm B.1 [11] for computing kP requires on average*

$$\frac{(2w+4)t}{w+1}M + \frac{(2w+3)t}{w+1}S + \frac{(w+2)t}{w+1}I,$$

where M, S and I denote multiplication, squaring and inversion respectively.

Proof . We noticed that algorithm B.1 [11] performs an ECADD operation each time the current digit is non–zero, recall from theorem 4[11] that the average non–zero density of wMOF is asymptotically $\frac{1}{w+1}$ also, one ECDBL operation is performed in each iteration (where $i \geqslant 0$) to double the intermediate result. Then on average, algorithm B.1 [11] for computing exponentiation with wMOF requires

$$t\ ECDBL + \frac{t}{w+1} ECADD.$$

Recall that the computational costs for doubling and additions operations in affine coordinate. Then we can rewrite previous expression as:

$$(2M + 2S + I)t + \frac{t}{w+1}(2M + S + I)$$

We can rewrite previous expression in terms of M, S and I as:

$$\frac{(2w+4)t}{w+1}M + \frac{(2w+3)t}{w+1}S + \frac{(w+2)t}{w+1}I.$$

□

Theorem 3.3. *In terms of affine coordinate, let $P \in E(\mathbf{F}_p)$, and t–digits exponent in wMOF, then the complexity of algorithm 1 for computing kP requires on average*

$$\frac{4(w+3)t}{w+1}M + \frac{4(w+2)t}{w+1}S + \frac{2t}{w+1}I,$$

where M, S and I denote multiplication, squaring and inversion respectively.

Proof . Recall from [11, Theorem 4] that for t–digits exponent k in its wMOF, if $t \to \infty$ the average non–zero density of wMOF is asymptotically $\frac{1}{w+1}$ and wMOF of k is infinity.
Long sequence constituted from two types of blocks:

1. $b_1 = (0)$, length of this block is 1;
2. $b_2 = (0^i * 0^{w-i-1})$, length of this block is w and $0 \leqslant i \leqslant w-1$.

Then the number of block b_2 equals $\frac{1}{w+1}$ because every block b_2 has a non–zero bit, and the number of block b_1 equals amount of $0s$ in wMOF – the amount of $0s$ in b_2 which equals

$$\frac{w}{w{+}1}t - (w-1)(\frac{1}{w{+}1})t = \frac{t}{w{+}1}.$$

Now, step 3.1 of algorithm 1 performs $\frac{1}{w+1}t$ blocks b_1 and step 3.2 performs $\frac{1}{w+1}t$ block b_2 then algorithm 1 for computing kP requires on average

$$\frac{t}{w{+}1}ECDBL + \frac{t}{w{+}1}DECDBL(w).$$

Recall the computational costs for doublings and additions operations in affine coordinate. Then we can rewrite previous expression as:

$$\frac{n}{w+1}\left(2M + 2S + I + 4(w+2)M + 4(w+1)S + 2M + 2S + I\right).$$

We can rewrite previous expression in terms of M, S and I as:

$$\frac{4(w+3)t}{w+1}M + \frac{4(w+2)t}{w+1}S + \frac{2t}{w+1}I.$$

Now Theorem 3.3 describes the complexity of algorithm 1 for computing exponentiation with wMOF by using $2^{n_2}(2^{n_1}P + Q)$. □

Relative improvement

Let us denote the times it would take to perform exponentiation by using algorithms B.1 [11] and 1 by symbols $T_{Regularmethod}$ and $T_{Formulamethod}$, respectively. According to theorems B.1[11], and Theorem 3.1, we can derive expressions for the time it would take to perform a whole exponentiation with wMOF as:

$$T_{Regular\,method} = \frac{(2w+4)t}{w+1}M + \frac{(2w+3)t}{w+1}S + \frac{(w+2)t}{w+1}I, \tag{3.14}$$

$$T_{Formula\,method} = \frac{4(w+3)t}{w+1}M + \frac{4(w+2)t}{w+1}S + \frac{2t}{w+1}I. \tag{3.15}$$

From equations (3.14), and (3.15), one can readily derive the relative improvement by defining $r = I/M$ (the ratio of speed between a multiplication and inversion) as:

$$Relative\,Improvement = \frac{T_{Regular\,method} - T_{Formula\,method}}{T_{Regular\,method}}. \tag{3.16}$$

By using (3.14) and (3.15)

$$Relative\,Improvement = \frac{wI - [(2w+8)M + (2w+5)S]}{(w+2)I + [(2w+4)M + (2w+3)S]}. \tag{3.17}$$

In our implementation $S \approx M$ and $r = 12.6$, let $w = 4$, then

$$Relative\,Improvementis = \frac{4(r) - 29}{6(r) + 23}, \tag{3.18}$$

$$Relative\,Improvementis = \frac{4(12.6) - 29}{6(12.6) + 23} \times 100 = 21.7\%. \tag{3.19}$$

4. Implementation and results

In this section, we implement our methods and others, which have been given in previous sections to show the actual performance of exponentiation. Implementation of an ECC system has several choices. These include selection of elliptic curve domain parameters, platforms [1].

4.1. Elliptic curves domain parameters and platforms

Generating the domain parameters for elliptic curve is vary time consuming. It consists of a suitably chosen elliptic curve E defined over a prime finite field $\mathbf{F}_p$, and a base point $G \in E(\mathbf{F}_p)$. Therefore we select NIST–recommended elliptic curves domain parameters in [10]. We implement 4 elliptic curves over prime fields $\mathbf{F}_p$, the prime modulo p are of a special type (generalized Mersenne numbers) with $\log_2 p = 160, 192, 224, 256$. We call these curves as $P160, P192, P224$ or 256 respectively. The ECC is implemented on a Pentium 4 personal computer (PC) with 2.0 GHz processor and 512 MB of RAM. Programs were written in Java language for multi–precision integer operations, and are ran under Windows XP.

4.2. Timings analysis of wMOF exponentiation method

We performed timing measurements on the individual k doublings and one addition operations and the corresponding formulae for direct computation of one addition adjoint with k doublings. In addition, we developed timing estimates based on the approximately ratio of speed between a multiplication and inversion I/M in prime filed $\mathbf{F}_p$ as presented in Table 3.

Table 3: The ratio of speed between a multiplication and inversion in prime filed $\mathbf{F}_p$.

Curves	Average Timing (msec) for M	Average Timing (msec) for S	Average Timing (msec) for I	$r = I/M$
P160	7.0	6.9	88.0	12.6
P192	8.7	8.6	108.8	12.5
P224	10	9.8	123.1	12.3
P256	11.9	11.8	145.2	12.2

4.2.1. Optimal window size

To show the actual improvement of wMOF method with our new formula, we must find out the most efficiency proper window size, where the length of input binary form is 160–bits, 192–bits, 224–bits, or 256–bits. Figures (1– 4) illustrate the relation among the window size w, the speed of the evaluation and pre–computed processes. We can notice from these Figures that when the window size increases, time of the evaluation will decrease, while time of the precomputation will increase, and the optimal w is 4 when the input is 160–bits, and the optimal w is 5 when the inputs is 192, 224 or 256–bits. So all the tests in this paper will be processed for $w = 4$ in 160–bits input and $w = 5$ for 192, 224, or 256–bits.

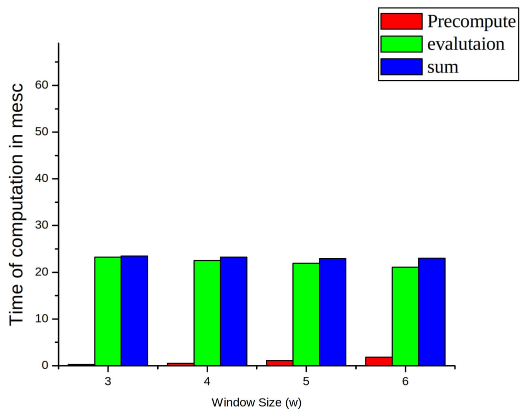

Figure 1: Pre–compute and evaluation with 160–bits input

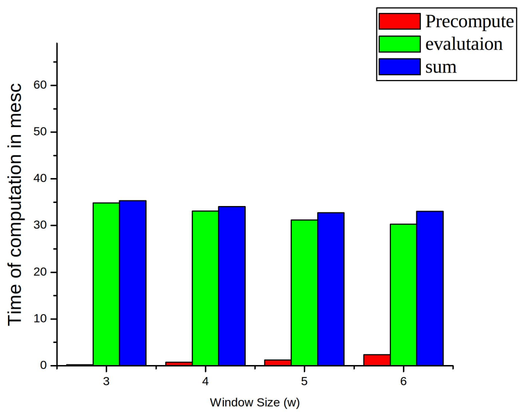

Figure 2: Pre–compute and evaluation with 192–bits input

4.2.2. The performance of improved wMOF method

Using Table 3, we can readily predict that the timings for performing a exponentiation with and without the formulae presented in Algorithm 1. In addition, using the complexity shown in equations (3.14, 3.15) and the timings shown in Table 3 we can make estimates as to how long an exponentiation with wMOF will take using both doublings with formulae and individual doublings.

5. Conclusion

In this paper, we constructed efficient algorithm for exponentiation on elliptic curve defined over $\mathbf{F}_p$ in terms of affine coordinates. The algorithm computes $2^{n_2}(2^{n_1}P + Q)$ directly from random points P and Q on an elliptic curve, without computing the intermediate points. We showed that our

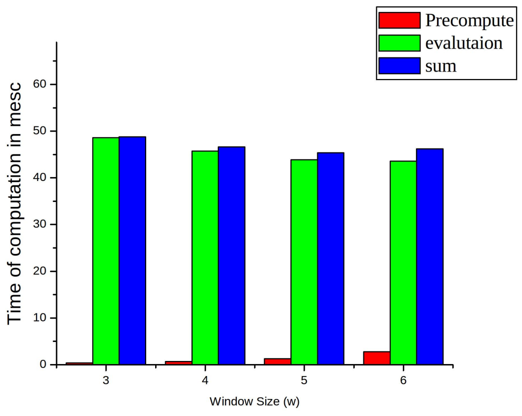

Figure 3: Pre–compute and evaluation with 224–bits input

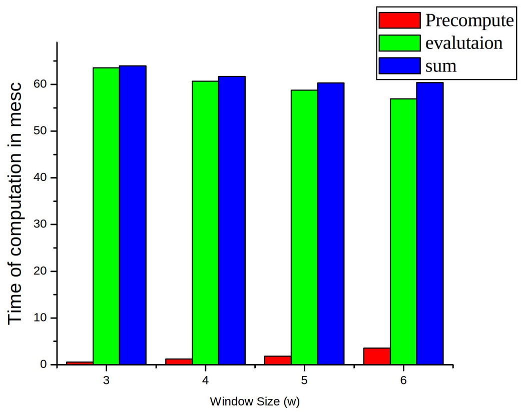

Figure 4: Pre–compute and evaluation with 256–bits input

Table 4: Average time comparison required to perform an exponentiation without pre–computations stage of a random point in mesc (Pentium IV 2.0 GHz).

Curves	Method	Predicted Timing	Measured Timing	% Improvement	
				Predicted	Measured
P 160	wMOF with formulae ($w = 4$)	17.4	18.3	21.62	21.8
	wMOF ($w = 4$)	22.2	23.4		
P 192	wMOF with formulae ($w = 5$)	23.8	24.3	25.62	25.7
	wMOF ($w = 5$)	32	32.6		
P 224	wMOF with formulae ($w = 5$)	31.7	33.9	24.52	24.6
	wMOF ($w = 5$)	42	45		
P 256	wMOF with formulae ($w = 5$)	43.8	47.4	23.5	23.3
	wMOF ($w = 5$)	57.3	61.8		

algorithm for computing $2^{n_2}(2^{n_1}P + Q)$ is more efficient than Sakai–Sakuri algorithm for computing $2^{n_1+n_2}P$ and $2^{n_2}Q$. A comparison was made based on notation of a "break even point", which is the cost factor of one inversion relatively to the cost of one multiplication. Moreover, we applied the algorithm to exponentiation on elliptic curve with wMOF and analyze its computational complexity.

This algorithm can speed the wMOF exponentiation of elliptic curve of size 160–bit about (21.7%) as a result of its implementation with respect to affine coordinates.

References

[1] M. Brown, D. Hankerson, J. López and A. Menezes, *Software implementation of the NIST elliptic curves over prime fields*, In: Naccache D. (eds) Topics in Cryptology–CT–RSA 2001. CT–RSA 2001. Lecture Notes in Computer Science, vol 2020. Springer, Berlin, Heidelberg.

[2] H. Cohen, A. Miyaji and T. Ono, *Efficient elliptic curve exponentiation using mixed coordinates* In: Ohta K., Pei D. (eds) Advances in Cryptology–ASIACRYPT'98. ASIACRYPT 1998. Lecture Notes in Computer Science, vol 1514. Springer, Berlin, Heidelberg.

[3] W. Diffie and M.E. Hellman, *New directions in cryptography* IEEE Trans. Inf. Theory, 22 (1976) 644–654.

[4] J. Guajardo and Ch. Paar, *Efficient algorithms for elliptic curve cryptosystems*, In: Kaliski B.S. (eds) Advances in Cryptology–CRYPTO '97. CRYPTO 1997. Lecture Notes in Computer Science, vol 1294. Springer, Berlin, Heidelberg.

[5] N. Koblitz, *Elliptic curve cryptosystems*, Math. Comp., 48 (1987) 203–209.

[6] K. Koyama and Y. Tsuruoka. *Speeding up elliptic cryptosystems by using a signed binary window method*, In: Brickell E.F. (eds) Advances in Cryptology–CRYPTO' 92. CRYPTO 1992. Lecture Notes in Computer Science, vol 740. Springer, Berlin, Heidelberg.

[7] V.S. Miller. *Use of elliptic curves in cryptography*, In: Williams H.C. (eds) Advances in Cryptology – CRYPTO '85 Proceedings. CRYPTO 1985. Lecture Notes in Computer Science, vol 218. Springer, Berlin, Heidelberg.

[8] A. Miyaji, T. Ono and H. Cohen, *Efficient elliptic curve exponentiation*, In: Han Y., Okamoto T., Qing S. (eds) Information and Communications Security. ICICS 1997. Lecture Notes in Computer Science, vol 1334. Springer, Berlin, Heidelberg.

[9] B. Möller. *Improved techniques for fast exponentiation*, In: Lee P.J., Lim C.H. (eds) Information Security and Cryptology — ICISC 2002. ICISC 2002. Lecture Notes in Computer Science, vol 2587. Springer, Berlin, Heidelberg.

[10] National Institute of Standards and Technology. Digital signature standard (dss). In Pil Joong Lee and Chae Hoon Lim, editors, *Information Security and Cryptology-ICISC 2002*. FIPS PUB 186-2, 2000.

[11] K. Okeya, K. Schmidt–Samoa, Ch. Spahn and T. Takagi, *Signed binary representations revisited*, In: Franklin M. (eds) Advances in Cryptology – CRYPTO 2004. CRYPTO 2004. Lecture Notes in Computer Science, vol 3152. Springer, Berlin, Heidelberg.

[12] Y. Sakai and K. Sakurai, *Efficient scalar multiplications on elliptic curves with direct computations of several doublings*, IEICE Transactions on Fundamentals of Electronics, Commun. Comput. Sci., E84–A (2001) 120–129.