BIRZEIT UNIVERSITY
FACULTY OF GRADUATE STUDIES

EFFICIENT ELLIPTIC CURVE CRYPTOSYSTEMS USING EFFICIENT EXPONENTIATION

by

Kamal Darweesh

A Thesis Submitted in Partial Fulfillment of the Requirement for the Master Degree in Scientific Computing From the Graduate Faculty at Birzeit University

Supervisor

Professor Mohammad Saleh
Department of Mathematics

Birzeit, Palestine
August, 2006

EFFICIENT ELLIPTIC CURVE CRYPTOSYSTEMS USING EFFICIENT EXPONENTIATION

by

Kamal Darweesh

This thesis was successfully defended on August 3, 2006 and approved

by:

Committee Members	Signature
1. Professor Mohammad Saleh	…….
2 Associate Professor Hasan Yousef	…….
3. Associate Professor Wasfi El Kafri	…….

ABSTRACT

The explosive growth in the use of mobile and wireless devices demands a public key cryptosystem (PCK) achieving aspects of information security with accommodate limitations on power and bandwidth, at the same time keeping with high level of security.

Elliptic curve cryptosystem (ECC) are new generation of public key cryptosystems that has smaller key sizes for the same level of security. The exponentiation in elliptic curve is the most important operation in ECC, So when put the ECC 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 thesis, we improve 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. Moreover, we apply this algorithm on exponentiation on elliptic curve with wMOF and analyze their 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.

الملخص

تزايد استعمال الأجهزة الخلوية واللاسلكية يتطلب أنظمة التشفير المفتاح العام التي تنجز مظاهر من سرية المعلومات و تتلاءم مع محدودية القوة والسعة لتلك الأجهزة مع الاحتفاظ بمستوى عالي من السرية.

تعد أنظمة التشفير على المنحنيات الإهليجية جيلا جديدا من أنظمة التشفير المفتاح العام والتي لها مفتاح أقل حجما لنفس مستوى السرية. إن العملية الأسية على المنحنيات الإهليجية أهم عملية في أنظمة التشفير على المنحنيات الإهليجية، لذلك تكمن المشكلة الرئيسية عند تطبيق تلك الأنظمة في كيفية تسريع العملية الأسية، لذا إن من أهم الاهتمامات هو تطوير طرق وخوارزميات تسمح بتنفيذ أنظمة التشفير على المنحنيات الإهليجية بفاعلية.

في هذه الرسالة، قمنا بتطوير طريقة فعالة لحساب العملية الأسية على المنحنيات الإهليجية المعرفة على الحقل المنتهي باستخدام الإحداثيات الديكارتية المتعامدة. تستطيع هذه الخوارزمية حساب $2^{م}(2^{ن}$ أ + ب) مباشرة من نقطتين أ، ب عشوائيتين على المنحنيات الإهليجية، وبدون حساب النقاط الوسطية، وبالإضافة إلى ذلك قمنا بتطبيق هذه الخوارزمية من أجل حساب العملية الأسية على المنحنيات الإهليجية وتحليل مدى الوقت المستهلك. لقد دلت النتائج عند تنفيذ هذه الخوارزمية أنها تستطيع تسريع العملية الأسية على المنحنيات الإهليجية عند أخذ مفتاح بحجم 160- بت حوالي 21.7 % .

ACKNOWLEDGMENTS

I would like to thank Dr. Mohammed Saleh, my supervisor, for his many suggestions and constant support during this research. I would also like to thank my committee members, Dr. Wasfi El Kafri and Dr. Hasan Yousef, for their valuable comments on my thesis.

I am grateful to my parents and my wife for their *patience* and *love*. I would like to thank my colleagues who also gave me valuable comments on my research.

CONTENTS

LIST OF ALGORITHMS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHd	Average Hamming density.
DLP	Discrete Logarithm Problem.
DECDBL (*w*)	Direct computing of point addition adjoint with *w* doublings
ECC	Elliptic Curve cryptosystem
ECADD	Elliptic Curve Point Addition.
ECDBL	Elliptic Curve Point Doubling.
ECDH	The Elliptic Curve Diffie-Hellman Key Exchange
ECDLP	Elliptic Curve Discrete Logarithm Problem.
ECDSA	The Elliptic Curve Digital Signature Algorithm
Hd	Hamming density.
Hw	Hamming weight.
MOF	Mutual opposite Form.
PCK	Public key cryptosystem
SW	Sliding window
*w*NAF	Width-w Non Adjacent Form.
*w*MOF	Width-w Mutual Opposite Form.

LIST OF SYMBOLS

A	Affine coordinates.
t	Bit length of an exponent.
k	Exponent, i.e. a positive integer.
k_i	The i-th bit of the exponent k, i = 1, . . . , n.
$\overline{x}$	-*x*, where *x* is an integer.
D	Digit set.
D^*	Digit set without zero.
|D|	The order of the digit set.
χ	Class of D-representations.
M	Field multiplication.
S	Field squaring.
I	Field inversion.
F	Field
$\mathbf{F}_q$	The field with q elements
$\mathbf{F}_p$	Prime finite field
$\mathbf{F}_{2m}$	Binary finite field
$E(\mathbf{F}_q)$	Additive group of points on an elliptic curve over finite field $\mathbf{F}_q$
P	Projective coordinates
$\#E(\mathbf{F}_q)$	The number of points on E over finite field $\mathbf{F}_q$

CHAPTER 1

1 Introduction

Cryptography is the science of securely transmitting messages from a sender to a receiver. So the need of cryptography is on the increase, it enable people to communicate securely. People interact electronically, through e-mail, e-commerce, ATM machines, or mobile. A cryptosystem is a system of algorithms for encrypting and decrypting messages for this purpose. Many of modern cryptosystems have been proposed to achieve aspects of information security as confidently, data integrity, authentication, and non-repudiation.

1. Data integrity: service guarantees that the content of the message, that was sent, has not been tampered with.
2. Confidentiality: service protects against unauthorized disclosure of the information.
3. Authentication: service related to identification, and consists of two components data origin and entity authentication.
4. Non-repudiation: service protects against denial by one of the entities involved in a communication of having participated in all or part of the communication.

In order to obtain these aspects of information security, cryptographers have developed a toolbox of cryptographic primitives such as encryption schemes and digital signature. These primitive *called cryptographic schemes* and are also so-called *cryptosystems* [32].

The purpose of encryption schemes is to cover confidentiality of encrypting the message. This is done by an encryption function E. The reverse process, the decryption, is done by a decryption function D. Besides the message m, the encryption function requires the input of an encryption key e. It returns the encrypted message, the ciphertext c. The ciphertext and a decryption key d are the input for the decryption function which returns the original message, the plaintext. The respective formulas are given as

$$E_e(m) = c \quad , \quad D_d(c) = m$$

There are two mainly different approaches to encrypt messages: symmetric schemes, and symmetric schemes.

In symmetric schemes encryption and decryption are performed using the same secret key. This method is known as secret key or symmetric cryptography. Suppose Alice wishes to securely communicate some plaintext to Bob. She generally accomplishes this by applying an encryption function E to the plaintext, obtaining ciphertext. Bob must have the inverse function D, and it should not be easy for an eavesdropper to recover the plaintext from the ciphertext.

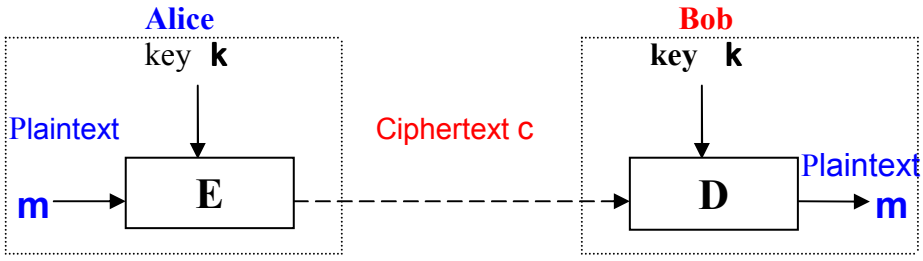

Figure 1.1 Secret-key schemes

While the encryption and decryption with symmetric schemes is very fast, it has drawback, namely the key-exchange between communicating parties. When

the sender and the receiver are physically apart, they want to agree on the secret key without anyone else finding out. Then they must share a secret key through secure channel. Consequently, in a large system many secret keys must typically be generated, stored, managed, and destroyed in a highly secure way. If, for example, n entities want to securely communicate with each other, then there are n(n-1)/2 secret keys that must be generated, stored, managed, and destroyed. Another approach to agree on secret key between two parties is using a trusted third party to prevent the disclosure of the secret key. Unfortunately, this method has many disadvantages. The most important disadvantage is that each entity must unconditionally trust the of third party and share a secret key with it. There are situations in which this level of trust is neither justified nor can be accepted by the communicating entities.

In 1976 Whitfield Diffie and Martin Hellman [8] published their paper "New Directions in Cryptography" and proposed the Diffie-Hellman (DH) key Exchange protocol [8] which allows users to exchange secret keys over an insecure channel without any prior shared secret. This paper resolves the key-exchange problem and becomes the theoretical concept of asymmetric schemes.

1. Alice and Bob agree on some finite group G and an element $g \in G$.
2. Alice privately chooses an integer $a \in \{1,\ldots|g|\}$, and computes
   $\alpha = g^a$. She sends α to Bob
3. Bob privately chooses an integer $b \in \{1,\ldots|g|\}$, and computes
   $\beta = g^b$. He sends β to Alice.
4. Alice and Bob can both compute
   $k = g^{ab} = (g^a)^b = (g^b)^a$ as common secret key.

Figure 1.2 Diffie-Hellman Key Exchange protocol

In a asymmetric schemes, each user gets a pair of keys: private key for decryption which is kept secret, and public key for encryption which can be made public for that reason asymmetric schemes are also referred to as public-key schemes. It is computationally infeasible to deduce the private key from the public key. Anyone who has a public key can encrypt information but cannot decrypt it. Only the person who has the corresponding private key can decrypt the information.

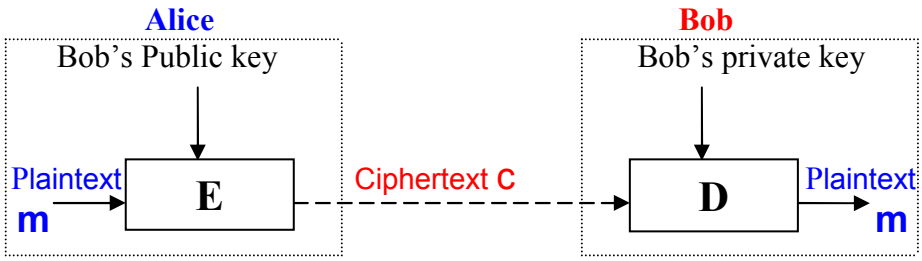

Figure 1.3 Asymmetric schemes

Digital signature schemes work similar to asymmetric schemes, namely they are based on a complex mathematical problem. They are designed to provide the digital counterpart to handwritten signatures to provide data integrity, data origin authentication, and non-repudiation. A digital signature is generated based on the content of the message being signed and some secrets known only to the signer including the private key and the signing key. It must be verifiable by any user in the system without accessing the signer's secret information.

There are only three classes of public key cryptosystems that are considered to be both secure and efficient. They are classified below according to the mathematical problem on which they are based [1].

1. Integer factorization systems: security is based on the intractability of the integer factorization problem (IFP). Examples include RSA and Rabin signature schemes.
2. Discrete logarithm systems: security is based on the intractability of the discrete logarithm problem (DLP) in a finite field. Examples include ElGamal, and DSA.
3. Elliptic curve discrete logarithm systems: security is based on the intractability of the elliptic curve discrete logarithm problem (ECDLP). Examples include Elliptic Curve Digital Signature Algorithm (ECDSA).

Elliptic curve cryptosystem ECC is new generation of public key cryptosystem that is based on the difficulty of ECDLP. ECC has advantage over the systems which are based on the multiplicative group of a finite field ($\mathbf{F}_q$). As a result, the fastest algorithm known for solving the discrete logarithm systems DLP in the multiplicative group ($\mathbf{F}_q$) is index-calculus method which solves the DLP in sub-exponential time [18], and the best known algorithm for solving the ECDLP in this group is Pollard-rho algorithm, it takes about $\sqrt{\frac{n\pi}{2}}$ steps, where a step here is an elliptic curve addition, n is the number of elliptic curve points, these steps takes full exponential time [2][18][19]. Consequently, one can use an elliptic curve group that is smaller in size with the same level of security maintained. The outcomes are smaller key sizes, bandwidth savings and faster

implementations. Such characteristics are particularly attractive for security applications where computational power and integrated circuit space are limited.

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 Key Exchange protocol strongly depends on the efficiency of exponentiation, because it is the costliest operation. Thus it is very attractive to speed up of exponentiation, which allow for efficient implementations of elliptic curve cryptosystems.

Three are three ways to speed up of exponentiation: choosing optimal underlying field, on which modular reduction is efficient or on which inversion is efficient, reducing the number of additions, and reducing the number of multiplications and squirings of underlying field by using efficient point coordinate system or mixed coordinate systems [7].

There are two mainly types of elliptic curve exponentiation algorithms of the second way: algorithms for a fixed point, and algorithms for a random point. These algorithms can compute elliptic curve exponentiation by repeating additions and doublings, where the repeated number of additions can be reduced by a suitable algorithm, but that of doublings can not be reduced.

1. Exponentiation algorithms for a fixed point: compute an elliptic curve exponentiation by repeating only additions and no doubling. In this case, the precomputation table method [10] is useful.

2. Exponentiation algorithms for a random point: compute an elliptic curve exponentiation by repeating additions and doubling. In this case, the addition-subtraction method is usually mixed with the window method [10][20][18][26].

The binary method is the standard algorithm for computing exponentiation in the case of a random point. It based on the binary representation of the exponent, the elliptic addition point is performed if scanned bit of exponent is one and doubling of elliptic point is performed regardless of scanned bits, so this method also so-called add-double. It can scan the bits of exponent from left to right or from right to left, and can be generalized to use base-2 representation. So the average number of addition of elliptic points operations required by the binary method or the general binary method depends on the minimal hamming weight of the exponent. Here, the fact that points on an elliptic curve can be inverted at negligible costs proved very useful, namely the effort for precomputing the required points can be reduced by more than 50%, if the exponent is represented in a signed representation.

There are several base-2 representations which have minimal hamming weight, namely the Width-w Non Adjacent Form (wNAF) and siding window on some signed binary representations. While those representations speed up the exponentiation in the best possible way. The generation of the wNAF and the recoding of signed binary digits for sliding window are only possible starting at the least significant bit, i.e. right-to-left. Therefore, the recoding of the n-bit exponent must be performed in a separate stage and the whole recoded exponents

must be stored, which requires memory of the order of magnitude of n bits for both base-2 representations.

This problem is solved by using base-2 representation Width-w Mutual Opposite Form (*w*MOF) which provides the same minimal hamming weight of exponent as the *w*NAF. Their great advantage is, that they 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. In detail, the wMOF requires only memory of the order of magnitude of w bits, which is very small compared to n-bits.

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,…,2^{n-1}. Sakai and Sakurai [28] proposed formulae for computing 2^nP directly ($\forall n \geq 1$) on E($\mathbf{F}_p$) in terms of affine coordinates. Since modular inversion is more expensive than multiplication, this formula requires only one inversion for computing 2^nP instead of d inversions in regular add-double method.

In this thesis, first we derive formula to compute $2^{n_2}(2^{n_1}P+Q)$ directly from P, Q $\in$ E($\mathbf{F}_p$), without computing intermediate points $2P,2^2P,\text{L},2^{n_1}P$, $2(2^{n_1}P+Q),\text{L},2^{n_2-1}(2^{n_1}P+Q)$, where $n_1 \geq 1$. Secondly, we use this formula to improve evaluation stage for computing exponentiation with wMOF method.

Furthermore, we show in what way this new derived formula can improve the speed of the exponentiation with wMOF. 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.

This thesis is organized as follows: Chapter 2 gives background on elliptic curve and discuses their various properties. Chapter 3 presents base-2 representation of integer and introduces algorithms for the efficient computation of elliptic curve exponentiation. Those were the binary and the general binary methods which use minimal hamming weight signed representations of the exponent, namely the wNAF, siding window on some applied on NAF and wMOF which left to right minimal hamming weight representation. Finally derived formula for direct computation of several doubling of elliptic points in affine coordinates is presented. Chapter 4 presents new formula for computing $2^{n_2}(2^{n_1}P+Q)$ directly from P, Q $\in$ E($\mathbf{F}_p$), without computing intermediate points $2P, 2^2P, \mathrm{L}\ , 2^{n_1}P$. Chapter 4 also shows in what way this formula can improve the speed of the exponentiation with wMOF. Finally Chapter 5 provides concluding remarks and discussion.

CHAPTER 2

2 Elliptic Curves

Elliptic curves were found as a result of studying the problem of the arc length of an ellipse. To compute the arc length one integrates a function involving $y = \sqrt{f(x)}$, and the answer is given in terms of certain functions on "elliptic" curve $y^2 = f(x)$.

More recently, elliptic curves have been used in devising efficient algorithm for factoring integers and for primality proving. In mid 1980's, Koblitz and Miller independently proposed the use of a group of points of elliptic curves, defined over a finite field, to be used for cryptographic purpose [2].

First we must to discuss elliptic curves and their various properties.

2.1 Weierstrass Equations

Let **F** be a field. Consider the following homogeneous cubic equation, called the Weierstrass equation:

$$Y^2Z + a_1XYZ + a_3Y\,Z^3 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3, \quad a_1, K, a_6 \in K. \quad (2.1)$$

Now consider the polynomial K(X, Y, Z) defined to be the left hand side of (2.1) minus its right hand side. Let $\bar{F}$ be the algebraic closure of **F**, and let

$$E = \left\{ [X, Y, Z] \in \mathbf{P}^2(\bar{F}) \mid K(X, Y, Z) = 0 \right\}. \quad (2.2)$$

E is called the projective curve defined by Weierstrass equation, and the number of points on E (the cardinality) is denoted #E($\overline{\mathbf{F}}$).

Definition 2.1 *Let a plane projective curve E, an element [X, Y, Z]* $\in$ *E for which*

$$\left(\frac{\partial K}{\partial X}(X,Y,Z),\frac{\partial K}{\partial Y}(X,Y,Z),\frac{\partial K}{\partial Z}(X,Y,Z)\right)=(0,0,0) \qquad (2.3)$$

is called a singular point, and E is said to be smooth, or non-singular curve if there are no singular points in E.

Definition 2.2 *An elliptic curve over the field* ***F****, is a smooth curve E defined by an Weierstrass equation in the form of (2.1).*

Definition 2.3 *Let E be an elliptic curve over the field* ***F*** *defined by an Weierstrass equation in the form of (2.1).* Let $\overline{\mathbf{F}}$ be the algebraic closure of **F**, we define the set E($\overline{\mathbf{F}}$) of $\overline{\mathbf{F}}$-rational points as follows:

$$E(\overline{F}) = \left\{[X, Y, Z] \in \mathbf{P}^2(\overline{\mathbf{F}}) \mid K(X, Y, Z) = 0\right\}. \qquad (2.4)$$

When we just write E we mean the set of $\overline{F}$*-rational points,* i.e. E = E($\overline{\mathbf{F}}$) (all the points on the curve)

Recall from Appendix A.2 that $\mathbf{P}^2(\overline{\mathbf{F}})$ is a disjoint union of $\mathbf{A}^2(\overline{\mathbf{F}})$ and the line at infinity; let's study the intersection of E with each such piece. First we to study the intersection of E with line at infinity i.e. when the condition Z = 0 holds in addition to the equation which defines E.

If K(X, Y, Z) = 0 is an equation defining an elliptic curve over **F**, then we see from (2.1) that

$$K(X, Y, Z) = 0 \Leftrightarrow X^3 = 0 \Leftrightarrow X = 0,$$

and Y is allowed to be anything. Thus, E intersects the line at infinity in the points [0, Y, 0], however, since $Y \neq 0$, these are all (by the equivalence relation) the same point [0, 1, 0]. So, E intersects the line at infinity in the single point O = [0, 1, 0].

Now, to study the intersection of E with $\mathbf{A}^2(\bar{\mathbf{F}})$, we need to see what happens when $Z \neq 0$. Every element $[X, Y, Z] \in \mathbf{P}^2(\bar{\mathbf{F}})$ for which $Z \neq 0$ has a unique representative [x, y, 1], where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$.

Dividing the original equation for E by Z^3, we get

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6 \qquad a_1, \ldots, a_6 \in \bar{F}. \tag{2.5}$$

which is an equation in only two variables. This is Weierstrass non-homogeneous equation, and leads to the affine representation of E.

Now consider the polynomial f(x, y) defined to be the left hand side of (2.5) minus its right hand side. We define the set of $\bar{\mathbf{F}}$-rational points

$$E(\bar{F}) = \left\{(x,y) \in \mathbf{A}^2(\bar{\mathbf{F}}) \mid f(x, y) = 0\right\} \cup \{O\} \tag{2.6}$$

Definition 2.4 *Let E be a curve given by a non-homogeneous Weierstrass equation (2.6). Define the quantities*

$$d_2 = a_1^2 + 4a_2$$

$$d_4 = 2a_4 + a_1a_3$$

$$d_6 = a_3^2 + 4a_6$$

$$d_8 = a_1^2\ a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2$$

$$\Delta = -d_2^2\ d_8 - 8d_4^3 - 27d_6^2 + 9d_2d_4d_6$$

$$c_4 = d_2^2 - 24d_4$$

$$j(E) = c_4^3/\Delta$$

The quantity Δ is called the discriminant of the Weierstrass equation, while j(E) is called the j-invariant of E if $\Delta \neq 0$.

Theorem 2.5 *The curve E is nonsingular (that is, it's an elliptic curve) if and only if $\Delta \neq 0$*

For a proof of the above theorem, see [9].

For fields **F** with various characteristics, we can transform the Weierstrass equation (2.5) into different forms of equations of an elliptic curve E by using linear change of variables. We split it into 3 cases: char(**F**) $\neq$ 2,3, char(**F**) = 3 and char(**F**) = 2. The corresponding admissible change of variables will be given in each case.

1. <u>char(**F**) $\neq$ 2,3:</u>

If char(**F**) $\neq$ 2, and the change of variables

$$y \rightarrow y - \frac{a_1x + a_3}{2}$$

is performed, then the left side of (2.5) after the substitution for *y* becomes:

$$(y - \frac{a_1x + a_3}{2})^2 + a_1x(y - \frac{a_1x + a_3}{2}) + a_3(y - \frac{a_1x + a_3}{2}) = \ldots$$

$$\cdots = y^2 - \frac{a_1^2 x^2}{4} - \frac{a_1 a_3 x}{2} - \frac{a_3^2}{4}$$

Both, xy and y have vanished, so their coefficients a_1 and a_3 must equal zero. That reduces the left side to a single y^2, and (2.5) becomes:

$$y^2 = x^3 + a_2x^2 + a_4x + a_6 \tag{2.7}$$

Further, if char(**F**) $\neq 3$, and the change of variables

$$x \to x - \frac{a_2}{3}$$

is performed, then the right side of (2.5) after the substitution becomes:

$$(x - \tfrac{a_2}{3})^3 + a_2(x - \tfrac{a_2}{3})^2 + a_4(x - \tfrac{a_2}{3}) + a_6 = \ldots$$

$$\ldots = x^3 + (\frac{a_2}{9} + a_4)x + \frac{2}{27}a_2^3 - \frac{1}{3}a_2a_4a_6$$

Setting $(\frac{1}{9}a_2 + a_4) = a$, and $\frac{2}{27}a_2^3 - \frac{1}{3}a_2a_4a_6 = b$, we have shorter form of Weierstrass non-homogeneous equation:

$$y^2 = x^3 + ax + b \tag{2.8}$$

Recall from theorem 2.5, the curve E is nonsingular or smooth if and only if $\Delta \neq 0$. For Weierstrass equation of the form (2.8), we have $d_2 = 0$, $d_4 = 2a$, $d_6 = 4b$, $d_8 = -a^2$, $c_4 = -48a$, $c_6 = -864b$, and $\Delta = -16(4a^3 + 27b^2)$. Therefore E is smooth if and only if $(4a^3 + 27b^2) \neq 0$.

2. <u>char(**F**) =3:</u>

If $a_2 \neq 0$, and the change of variables

$$x \to x + \frac{a_4}{a_2}$$

is performed, then the right side of (2.7) after the substitution becomes:

$$(x+\frac{a_4}{a_2})^3 + a_2(x+\frac{a_4}{a_2})^2 + a_4(x+\frac{a_4}{a_2}) + a_6 = \ldots$$

$$\ldots = x^3 + a_2 x^2 + \frac{a_2^2a_4^2+a_2^3+a_4^3}{a_2^3}$$

Setting a_2 = a, and $\frac{a_2^2a_4^2+a_2^3+a_4^3}{a_2^3}$ = b, we have Weierstrass non-homogeneous equation of the form:

$$y^2 = x^3 + ax^2 + b \tag{2.9}$$

If $a_2 = 0$, then by setting a_4 = a, and a_6= b from (2.7), we immediately have the same form (2.8)

3. <u>char(**F**) =2:</u>

Case 1. The supersingular case, *j(E)* = 0, i.e. $a_1 = 0$:

When the change of variables

$$x \to x + a_2$$

is performed, then left side of (2.5) becomes:

$$y^2 + a_3y$$

and the right side of (2.5) after the substitution becomes:

$$(x+a_2)^3 + a_2(x+a_2)^2 + a_4(x+a_2) + a_6 = \ldots$$

$$\ldots = x^3 + (a_4 + a_2^2)x + a_2^2 + a_2^3 + a_4a_2$$

Setting a_3 = a, $(a_4 + a_2^2)$ = b, and $a_2^2 + a_2^3 + a_4a_2$ = c, we have Weierstrass non-homogeneous equation of the form:

$$y^2 + ay = x^3 + bx + c \tag{2.10}$$

Case 2. The nun-supersingular case, *j(E)* ≠ 0, i.e. $a_1 \neq 0$:

When the change of variables

$$x \to a_1^2 x + \frac{a_3}{a_1}$$

is performed, then (2.5) after the substitution for x becomes:

$$y^2 + a_1(a_1^3 x + \frac{a_3}{a_1})y + a_3 y = (a_1^3 x + \frac{a_3}{a_1})^3 + a_2(a_1^3 x + \frac{a_3}{a_1})^2 + a_4(a_1^3 x + \frac{a_3}{a_1}) + a_6 \quad (2.11)$$

After the simplification, (2.11) becomes:

$$y^2 + a_1^3 xy = a_1^6 x^3 + (a_1^3 a_3 + a_2 a_1^4)x^2 + (a_3^2 + a_4 a_1^2)x + (\frac{a_3^3 + a_3^2 a_2 a_1 + a_4 a_3 a_1^2 + a_1^3 a_6}{a_1^3}) \quad (2.12)$$

Now if the change of variables

$$y \to a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3}$$

is performed, then the left side of (2.12) after the substitution for y becomes:

$$(a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3})^2 + a_1^3 x(a_1^3 y + \frac{a_1^2 a_4 + a_3^2}{a_1^3})y = \ldots$$

$$\ldots = a_1^6 y^2 + \frac{(a_1^2 a_4 + a_3^2)^2}{a_1^6} + a_1^6 xy + (a_3^2 + a_4 a_1^2)x$$

Finally (2.12) after the divide by a_1^6 becomes:

$$y^2 + xy = x^3 + \frac{(a_3 + a_2 a_1)}{a_1^3}x^2 + (\frac{a_1^3 a_3^3 + a_3^2 a_2 a_1^4 + a_4 a_3 a_1^5 + a_1^6 a_6 - (a_1^2 a_4 + a_3^2)^2}{a_1^{12}}) \quad (2.13)$$

Setting $\frac{(a_3 + a_2 a_1)}{a_1^3} = a$, and $(\frac{a_1^3 a_3^3 + a_3^2 a_2 a_1^4 + a_4 a_3 a_1^5 + a_1^6 a_6 - (a_1^2 a_4 + a_3^2)^2}{a_1^{12}}) = b$, we have Weierstrass non-homogeneous equation of the form:

$$y^2 + xy = x^3 + ax^2 + b \quad (2.14)$$

2.2 The Group Law

In what follows, we shall define the operation of addition in the group of points on an elliptic curve E over a field **F**.

Let E be an elliptic curve given by the Weierstrass equation (2.5) to add two points on the curve P and Q together, pass a straight line through them and look for the third point of intersection with the curve, R. Then reflect the point R over the x-axis to get –R, the sum of P and Q. Thus, P + Q = –R. The idea behind this group operation is that the three points P, Q, and R lie on a common straight line, and the points that form the intersection of a function with the curve are considered to add up to be zero as in Figure 2.1. If P = Q then the line to be constructed is the tangent of E at P, and P + Q =2P as in Figure 2.2.

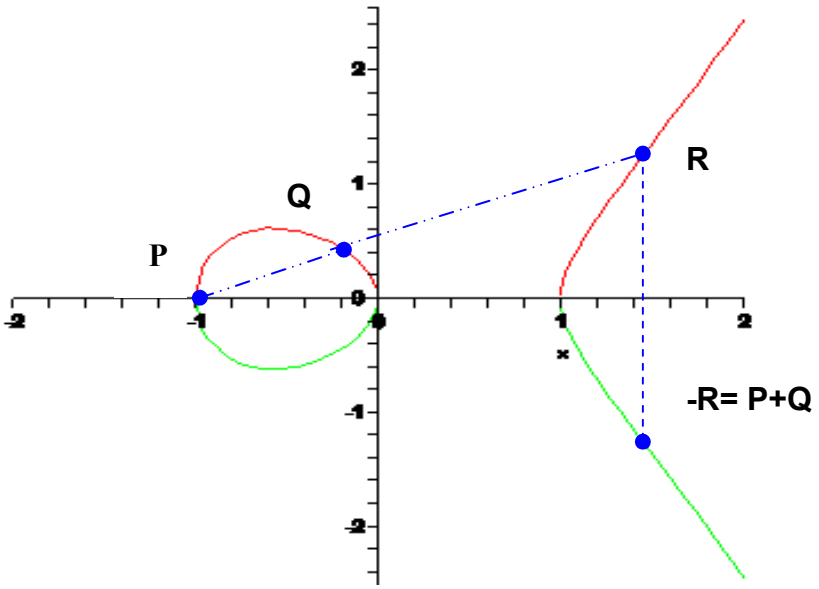

Figure 2.1 Elliptic curve point addition

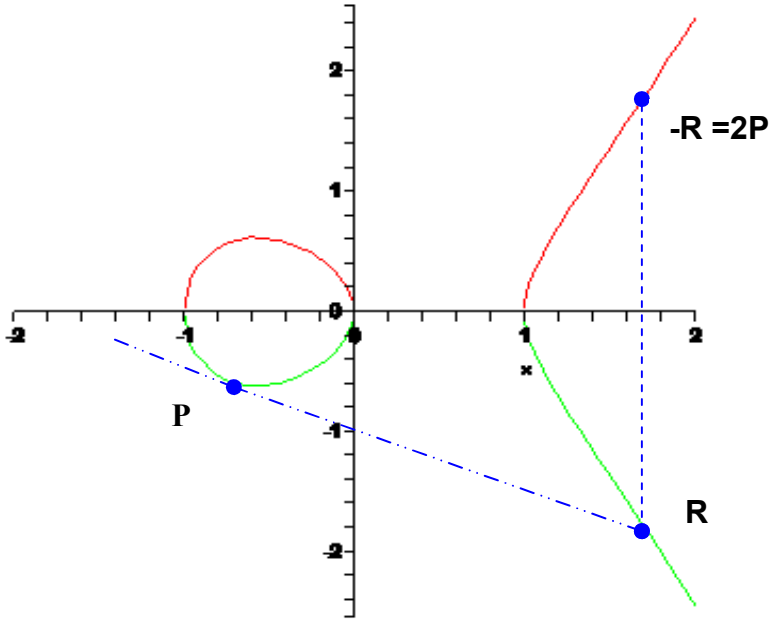

Figure 2.2 Elliptic curve point doubling

Now we want to define the identity element of E, Therefore, we find an extra point of intersection where E meets the line connecting P,Q and the point at infinity O, and call this point P + Q. By joining O to a point R on E, we mean that a vertical line is drawn through P, Q. Hence, the point at infinity O is the additive identity element and P + Q + R = O or P + Q = -R, (the inverse of R).

Now we want to define the inverse of a point P= $(x_1, y_1) \in E$. Let P= (x_1, y_1), Q=$(x_2,y_2) \in E$. Notice, that if $x_1 = x_2$ then

$$y_1^2 + a_1x_1y_1 + a_3y_1 = y_2^2 + a_1x_1y_2 + a_3y_2 \ ,$$

and hence either

$$y_1 = y_2 \quad \text{i.e. } P = Q$$

or

$$y_2 = -y_1 - a_1x_1 - a_3$$

Now define the inverse -P of the point P thus:

$$-P = (x_1, -y_1 - a_1x_1 - a_3).$$

From the previous definitions it follows that:

For all P, Q $\in$ E,

1. O + P = P and P + O = P. That is, O is the identity element.
2. -O = O
3. If P = $(x_1, x_2) \in O$, then -P = $(x_1 , -y_1 - a_1x_1 - a_3)$.
4. Q = -P, then P + Q = O.
5. If P $\neq$ O, Q $\neq$ O , Q $\neq$ -P, then let R be the third point of intersection (counting multiplicities) of either the line which intersects P and Q if P $\neq$ Q, or the tangent line to the curve at P if P = Q, with the curve. Then P + Q = -R.
6. P + Q = Q + P.

Now we can prove that the above rules make the points on an elliptic curve into an (abelian) group. The only group law that is not an immediate consequence of the geometrical rules is the associative law. It can be proved with following proposition

Proposition 2.6 Let L_1, L_2, L_3 be three lines that intersect a cubic curve in nine points $P_1, \ldots, P_9$ (counting multiplicity) and let L_1', L_2', L_3' be three lines that intersect the cubic curve in nine points $Q_1, \ldots, Q_9$. If $P_i = Q_i$ for $_i = 1, \ldots, 8$, then also $P_9 = Q_9$.

The six lines are set as follows

L_1 : the line through P,Q and -(P +Q)

L_2 : the line through R,-R and O

L_3 : the line through -P,-(Q+R) and S = P +(Q+R)

L_1' : the line through Q, R and -(Q+R)

L_2' : the line through P,-P and O

L_3' : the line through -(P +Q),-R and $S' = (P + Q) + R$

Now the lines L_1, L_2, L_3 and L_1', L_2', L_3' have eight points of intersection in common, namely P,-P,Q,R,-R,-(P + Q),-(Q + R) and O. One can therefore conclude that $S = S'$ which proves the associativity.

2.3 Addition Formulas

Let P and Q be two distinct rational points on elliptic curve E. The straight line joining P and Q must intersect the curve at one further point, R, since we are intersecting a line with a cubic curve. The point R will also be rational since the line, the curve and the points P and Q are themselves all defined over **F**. If we then reflect R in the x-axis, we obtain another rational point which we shall call P + Q as in Figure 2.1.

There are different addition formulas for fields of char(**F**) $\neq$ 2,3, char(**F**) = 3 and char(**F**) = 2. This section explains how to derive explicit formulas for point additions and point doublings where the char(**F**) $\neq$ 2,3 in affine and projective coordinate systems. For simplicity, we look at elliptic curves defined over the real number field **R**.

2.3.1 Addition Formulas in Affine Coordinates

Let $P = (x_1, y_1)$ and $Q = (x_2, y_2) \in E$ defined by an Weierstrass non-homogeneous equation (2.8) with $P \neq -Q$. The equation of the line L which intersects P and Q is given as

$$L : y = \lambda x + \beta, \tag{2.15}$$

Then $P + Q = (x_3, y_3)$ can be computed as follows:

Case1 $P \neq Q$

In this case λ is slope of intersected L

$$\lambda = \frac{(y_2 - y_1)}{(x_2 - x_1)}, \quad \beta = y_1 - \lambda x_1$$

The third point where L intersects the curve is $R = (x_R, y_R)$. Since $P + Q = (x_3, y_3) = (x_R, -y_R) = -R$ (the inverse of R, where $a_1, a_3 = 0$), holds and inserting this into (2.15) yields a formula for the y-coordinate of P + Q.

$$y_R = \lambda x_R + \beta$$

$$\Rightarrow y_3 = -\lambda x_3 + \beta$$

$$= -\lambda x_3 - y_1 + \lambda x_1$$

$$= \lambda(x_1 - x_3) - y_1$$

The x-coordinate of P + Q is obtained by inserting (2.15) into the equation of the ellptic curve defined by an Weierstrass equation in the form of (2.8). This yields

$$(\lambda x + \beta)^2 = x^3 + ax + b$$

$$\Rightarrow 0 = x^3 - \lambda^2 x^2 + (a - 2\lambda\beta)x - \lambda^2 + b$$

This equation can be solved by using the fact that the sum of the roots of a monic polynomial is equal to minus the coefficient of the variable of the second highest power. The three roots are x_1, x_2, x_3 and the coefficient is $-\lambda^2$. Therefore $x_1 + x_2 + x_3 = \lambda^2$ holds and since two of those roots are given by the x-coordinates of the points P and Q, x_3 can be calculated. Hence, the formula for a point addition in affine coordinates is:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda (x_1 - x_3) - y_1 \\ \lambda &= \frac{(y_2 - y_1)}{(x_2 - x_1)} \end{aligned} \tag{2.16}$$

Case 2: P = Q

In this case λ is given as the derivative

$$\lambda = \frac{dy}{dx} = \frac{3x_1^2 + a}{2y_1}$$

in P = (x_1, y_1), because the line L is now the tangent on the curve in P. The formula for a point doubling in affine coordinate can be derived by using the same arguments as above and is given as

$$\begin{aligned} x_3 &= \lambda^2 - 2x_1 \\ y_3 &= \lambda (x_1 - x_3) - y_1 \\ \lambda &= \frac{3x_1^2 + a}{2y_1} \end{aligned} \tag{2.17}$$

Note, that $x_1 = x_2$ holds in that case.

The drawback of affine coordinates is, that the required field inversion is very

costly compared to multiplications and squarings. To avoid inversions, alternative coordinate systems such as projective coordinates are used.

2.3.2 Addition Formulas in Projective Coordinates

Recall from section 2.1 that every element $[X, Y, Z] \in \mathbf{P}^2(\mathbf{F})$ for which $Z \neq 0$ has a unique representative $[x, y, 1]$, where $x = \frac{X}{Z}$ and $y = \frac{Y}{Z}$. The transformation between affine and Projective coordinates is:

$$\begin{aligned} &(x, y) \in \mathbf{A}^2(\mathbf{F}) \rightarrow [x, y, 1] \in \mathbf{P}^2(\mathbf{F}) \\ &[X, Y, Z] \in \mathbf{P}^2(\mathbf{F}) \rightarrow (X/Z, Y/Z) \in \mathbf{A}^2(\mathbf{F}) \end{aligned} \tag{2.18}$$

When we apply the transformation (2.18) on Weierstrass non-homogeneous equation (2.8) and multiply by a power of Z to clear denominators we get homogeneous equation:

$$Y^2Z = X^3 + aXZ^2 + bZ^3 \tag{2.19}$$

To get the formula for a point addition in projective coordinates we apply transformation (2.18) to (2.16) as follows:

Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$, $P \neq \pm Q$, and $P + Q = (X_3, Y_3, Z_3)$. Then

$$x_3 = \frac{\left(\frac{Y_2}{Z_2} - \frac{Y_1}{Z_1}\right)^2}{\frac{X_2}{Z_2} - \frac{X_1}{Z_1}} - \frac{X_1}{Z_1} - \frac{X_2}{Z_2}$$

$$= \left(\frac{Y_2Z_1 - Y_1Z_2}{X_2Z_1 - X_1Z_2}\right)^2 - \left(\frac{X_2Z_1 - X_1Z_2 - 2X_1Z_2}{Z_1Z_2}\right)$$

Let $u = Y_2Z_1 - Y_1Z_2$, $v = X_2Z_1 - X_1Z_2$; this yields

$$x_3 = \left(\frac{u}{v}\right)^2 - \left(\frac{v - 2X_1Z_2}{Z_1Z_2}\right)$$

Let A= $u^2\ Z_1Z_2 - v^3 - 2v^2\ X_1Z_2$; this yields

$$x_3 = \frac{A}{v^2 Z_1 Z_2} = \frac{X_3}{Z_3}$$

$$y_3 = \frac{\left(\frac{Y_2}{Z_2} - \frac{Y_1}{Z_1}\right)}{\left(\frac{X_2}{Z_2} - \frac{X_1}{Z_1}\right)} \left(\frac{X_1}{Z_1} - \frac{X_3}{Z_3}\right) - \frac{Y_1}{Z_1}$$

$$y_3 = \left(\frac{Y_2Z_1 - Y_1Z_2}{X_2Z_1 - X_1Z_2}\right)\left(\frac{X_1}{Z_1} - \frac{A}{v^2 Z_1 Z_2}\right) - \frac{Y_1}{Z_1}$$

$$= \left(\frac{u}{v}\right)\left(\frac{v^2X_1Z_2 - A}{v^2 Z_1 Z_2}\right) - \frac{v^3 Y_1 Z_2}{v^3 Z_1 Z_2}$$

$$= \frac{u(v^2X_1Z_2 - A) - v^3Y_1Z_2}{v^3 Z_1 Z_2} = \frac{Y_3}{Z_3}$$

In total, this yields

$$X_3 = Av \tag{2.20}$$

$$Y_3 = u(v^2X_1Z_2 - A) - v^3Y_1Z_2 \tag{2.21}$$

$$Z_3 = v^3 Z_1 Z_2 \tag{2.22}$$

Now to obtain the formula for a point doubling in projective coordinates we apply transformation (2.18) to (2.17) as follows:

Let P = (X_1, Y_1, Z_1) and 2P = (X_3, Y_3, Z_3). Then

$$x_3 = \frac{3\left(\frac{X_1}{Z_1}\right)^2 + a\left(\frac{}{}\right)^2}{2\left(\frac{Y_1}{Z_1}\right)\left(\frac{}{}\right)} - 2\frac{X_1}{Z_1}$$

$$= \frac{\left(3X_1^2 + aZ_1^2\right)^2 - 8Y_1^2Z_1}{4Y_1^2Z_1^2}$$

Let $w = 3X_1^2 + aZ_1^2$, $s = Y_1Z_1$, $B = X_1Z_1s$, $h = w^2 - 8B$; this yields

$$x_3 = \frac{h}{4Y_1^2Z_1^2} = \frac{X_3}{Z_3}$$

$$y_3 = \frac{3\left(\frac{X_1}{Z_1}\right)^2 + a\left(\frac{}{}\right)^2}{2\left(\frac{Y_1}{Z_1}\right)\left(\frac{}{}\right)}\left(\frac{X_1}{Z_1} - \frac{X_3}{Z_3}\right)\frac{Y_1}{Z_1}$$

$$= \frac{3\left(\frac{X_1}{Z_1}\right)^2 + a\left(\frac{}{}\right)^2}{2\left(\frac{Y_1}{Z_1}\right)\left(\frac{}{}\right)}\left(\frac{X_1}{Z_1} - \frac{X_3}{Z_3}\right)\frac{Y_1}{Z_1}$$

$$= \left(\frac{3X_1^2 + aZ_1^2}{2YZ_1}\right)\left(\frac{X_1}{Z_1} - \frac{h}{4Y_1^2Z_1^2}\right)\frac{Y_1}{Z_1}$$

$$= \left(\frac{w}{2YZ_1}\right)\left(\frac{4X_1Y_1^2Z_1 - h}{4Y_1^2Z_1^2}\right)\frac{4Y_1^3Z_1}{Z_1}$$

$$= \frac{w(4X_1Y_1s - h) - 4Y_1^3Z_1}{8Y_1^3Z_1^3}$$

$$= \frac{w(B - h) - 4Y_1^2s^2}{8s^3} = \frac{Y_3}{Z_3}$$

In total, this yields

$$X_3 = 2hs \tag{2.23}$$

$$Y_3 = w(B - h) - 4Y_1^2s^2 \tag{2.24}$$

$$Z_3 = 8s^3 \tag{2.25}$$

2.4 Elliptic Curves over Finite Fields

Calculations over the real numbers are slow and inaccurate due to round-off error. Cryptographic applications require fast precise arithmetic. So we are only interested in finite fields $\mathbf{F}_q$. The formulas stated previously do not change. But instead of using floating-point arithmetic use a large number and do all calculations modulo a large prime.

The key of the implementation of cryptosystem (ECC) is the selection of elliptic curve groups over the finite field of $\mathbf{F}_p$ and $\mathbf{F}_{2m}$, where p is a prime and m is positive integer. By definition, elliptic curve groups are additive groups. Any such field is isomorphic to $\mathbf{F}[x]/\langle f(x)\rangle$, where $f(x) = x^m + \sum_{i=0}^{m-1} a_i x_i, a_i \in F_p$, is a manic irreducible polynomial of degree *m* over $\mathbf{F}_p$.

Three kinds of finite fields $\mathbf{F}_q$ are especially suitable for elliptic curve cryptosystem (ECC), binary fields $\mathbf{F}_{2m}$, prime fields $\mathbf{F}_p$, and optimal extension fields (OEF) $\mathbf{F}_{pm}$ [30].

2.4.1 Prime Finite fields $\mathbf{F}_p$

For the finite fields $\mathbf{F}_q$ of q elements , where $q = p^m$ for some prime p and positive integer m =1, there is a finite field $\mathbf{F}_p$, which is called a prime finite field and consists of the set of integers modulo p, which are the all possible results of reduction modulo p:

$$\{0, 1, 2, \ldots., p\text{-}1\}$$

The arithmetic operation on $\mathbf{F}_p$ is the usual addition, subtraction and multiplication modulo p.

For elliptic curve E over a finite field $\mathbf{F}_p$, Weierstrass non-homogeneous equation (2.8) can be used in which the variable and coefficients all take on values in the integers modulo p. For some prime number p ≠ 2,3, Weierstrass non-homogeneous equation (2.8) can be rewritten as:

$$y^2 \bmod p = (x^3 + ax + b) \bmod p \qquad (2.26)$$

where $(4a^3 + 27b^2) \bmod p \neq 0$, for a, b ∈ $\mathbf{F}_p$.

Example 2.8 Let p = 23. Consider elliptic curve E: $y^2 = x^3 - 7x + 2$ defined over $\mathbf{F}_{23}$. Note that $4a^3 + 27b^2 = -1264 \pmod{23} \equiv 1 \neq 0$, so E is indeed an elliptic curve.

Indeed E has 26 points - all of them are explicitly shown in Table 2.1. The distribution of these points is graphically expressed in Figure 2.3.

(0,5)	(15,11)
(0,18)	(15,12)
(3,10)	(17,9)
(3,13)	(17,14)
(5,0)	(18,2)
(9,1)	(18,21)
(9,22)	(19,9)
(10,9)	(19,14)
(10,14)	(21,10)
(12,6)	(21,13)
(12,17)	(22,10)
(14,7)	(22,13)
(14,16)	*O*

Table 2.1 Points of E over field $\mathbf{F}_{23}$

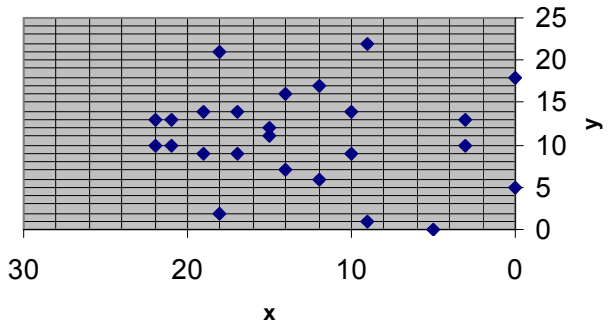

Figure 2.3 Doubling point P on E

We choose P (9, 1) is a point of E, because it satisfies the curve equation: $12 = 9^3 + 7(9) + 3 \pmod{23}$. Let's add points P + P. According to definition P + P = 2P = - R = $(x_R, -y_R)$, where $\lambda = (3*92 - 7)/2 = 236/2 = 6*12 \equiv 3 \pmod{23}$, $x_R = 32 - 2*9 = -9 \equiv 14 \pmod{23}$ and $-y_R = -1 + 3(9 - 14) = -16 \equiv 7 \pmod{23}$. Hence 2P =-R= (14, 7). Let's count also point -R = P + P + P = 2P + P, it means we add points 2P = (14,7) and P = (9,1). Results are $\lambda = (7 - 1)/(14 - 9) = 6*14 \equiv 15 \pmod{23}$, $x_R = 152 - 14 - 9 \equiv 18 \pmod{23}$, $-y_R = -7 + 15(14 - 18) \equiv 2 \pmod{23}$, thus 3P = (18, 2). Equally we can count 4P, 5P, …, 12P = (9, 22), 13P = *O*.

Obviously 14P = 13P + P = O + P = P, thus we finish the cycle and reach the starting point again.

2.4.2 Binary Finite Field $\mathbf{F}_{2m}$

The finite field $\mathbf{F}_{2m}$, called a binary finite field, of 2^m elements, can be viewed as vector space of dimension m over $\mathbf{F}_2$. That is, there exist a set of m elements $\{\alpha_0, \alpha_1, \alpha_0, \mathrm{K}, \alpha_{m-1}\}$ in $\mathbf{F}_{2m}$ such that, each $a \in \mathbf{F}_{2m}$ can be written uniquely in the form:

$$\sum_{i=0}^{m-1} a_i \alpha_i$$

where $a_i \in \{0,1\}$.

The elements of $\mathbf{F}_{2m}$ should be represented by bit strings of length m. There are several ways of performing arithmetic in $\mathbf{F}_{2m}$. The specific rules depend on how the string of bits is represented. There are two common structures for basis representation: polynomial basis representation and normal basis representation [2]

Polynomial base. A polynomial base is of the form $\{1, \alpha, \alpha^2, \mathrm{K}, \alpha^{m-1}\}$, where α is a root of an irreducible polynomial $f(x)$ of degree m over $\mathbf{F}_2$. The field is then realized as $\mathbf{F}_2[\mathrm{x}]/\langle f(x) \rangle$, and the arithmetic is that of polynomials of degree at most m-1, modulo $f(x)$, $\langle f(x) \rangle$ is the cyclic group generated by $f(x)$[2].

Normal base. A normal base of $\mathbf{F}_{2m}$ over $\mathbf{F}_2$ has the form $\{1, \alpha, \alpha^{2^2}, \mathrm{K}, \alpha^{2^{m-1}}\}$ for some $\alpha \in \mathbf{F}_{2m}$. It is known that such bases exist for all n ≥1. Normal bases are useful mostly in hardware implementations. First, the field squaring operation is

trivial in normal base representations, as it amounts to just cyclic shifting of the binary vector representing the input operand.

2.4.3 Optimal Extension Fields(OEFs) $\mathbf{F}_{(2^n \pm c)^m}$

Optimal Extension Fields(OEFs) are class of extension fields $\mathbf{F}_{pm}$, which exploit the optimization of integer arithmetic in modern processors to produce the fastest multiplication results over binary and prime fields.

The OEF is defined as $\mathbf{F}_{pm}$ which satisfies the following:

- p is a prime less than but close to word size of the processor.
- p is a pseudo-Mersenne prime given in the form $p = 2^n \pm c$, where

  $\log_2 c \leq \frac{1}{2} n$

  The elements of $\mathbf{F}_{pm}$ should be represented by a sequence of m words. All arithmetic operations are performed modulo the field polynomial.

2.5 Counting the number of points

Elliptic curve cryptosystems generally involve the selection of a suitable elliptic curve E and a point P on E called the base point. To learn more about the structure of the group E(**F**) it is useful to know the exact value of #E(**F**). We will look at the case when **F** is $\mathbf{F}_q$, a finite field of q elements. The following results are the best known methods to date for computing #E.

Theorem 2.9 [Hasse's] Let N be the number of points on an elliptic curve over $\mathbf{F}_q$, a finite field with q elements. Then

$$|N - (q + 1)| \leq 2\sqrt{q}$$

In another way, Hasse's Theorem gives the estimate #E(**F**q) = q+1-t where $|t| \leq 2\sqrt{q}$ [2].

Hasse's theorem enables one to compute #E($\mathbf{F}_q$k) from #E(**F**q) as follows.

Let t = q + 1 - #E(**F**q). Then

$$\#E(\mathbf{F}_q^{\,k}) = q^k + 1 - \alpha^k - \beta^k$$

where $1 - tT + qT^2 = (1 - \alpha T)(1 - \beta T)$[2][16].

Schoof 's Algorithm: In 1985, Schoof presented a deterministic algorithm that could compute #E(**F**q) (its precise value; not a bound or an estimate) in O(log9 q) bit operations (where *q* is some power of *p*)[2]. This deterministic polynomial time algorithm is the fastest to date, and given few alternatives, it is the best choice for computing #E. But in practice, it is awkward and costly to implement, particularly when q is large.

2.6 Discrete Logarithm Problem for Elliptic Curves

In 1980 Neal Koblitz and Victor Mille independently proposed elliptic curve cryptosystems (ECC), based on the difficulty of mathematical problem so-called the elliptic curve discrete logarithm problem (ECDLP)[2].

Indeed the elliptic curve discrete logarithm problem ECDLP is the inverse operation of exponentiation in elliptic curve, which can be stated as follows. Fix an elliptic curve. kP represents the point P added to itself k times. Suppose Q is a multiple of P, so that

$$Q = kP$$

for some integer k. Then the elliptic curve discrete logarithm problem is to determine k given P and Q.

Now we are ready to define the ECDLP as follows:

Definition 2.7 [Elliptic curve discrete Logarithm Problem-ECDLP] *given an elliptic curve E defined over finite field* $\mathbf{F}_q$*, a point* $P \in E(\mathbf{F}_q)$ *of order n, and a point* $Q \in E(\mathbf{F}_q)$*, find an integer k,* $0 \le k \le n-1$*, for which* $Q = kP$[1] [2].

2.6.1 Known Algorithms

The Phlig–Hellman algorithm reduces the determination of *k* to the determination of k modulo each of the prime factors of *n*. Therefore, to achieve the maximum possible security level, *n* should be prime. To date, the fastest algorithm for solving ECDLP is the Pollard ρ–method, as modified by Gallant, Lambert and Vanstone, and Wiener and Zuccherato, which takes about $\sqrt{\frac{\pi n}{2}}$ steps, where each step is an elliptic curve addition. In addition, Van Oorschot and Wiener showed how the Pollard ρ–method can be parallelized so that if *r* processors are used, the expected number of steps by each processor before a single discrete logarithm is obtained is $\sqrt{\frac{\pi n}{2}}$ [1][2]. For elliptic curves *E* defined over a subfield $\mathbf{F}_{2k}$ of $\mathbf{F}_{2m}$, the parallelized Pollard ρ–method for ECDLP in $E(\mathbf{F}_{2m})$ can be sped up to an expected running time of $\sqrt{\frac{\pi nk/m}{2r}}$. Therefore the fastest known algorithm that solves ECDLP in general is the Pollard ρ–method which it runs in full exponential time. Since the index calculus methods can compute discrete logarithm problem

(DLP) in the multiplicative group of a finite field ($\mathbf{F}_q$) in sub-exponential time, they cannot be applied to the case of discrete logarithms over elliptic curves[1][2].

2.6.2 Weak Curves

There are certain types of elliptic curves in which a successful attack could take place in sub-exponential time. These curves can easily be tested for and avoided. Such curves are called the supersingular curves and anomalous curves.

Supersingular curves are a special class of elliptic curves on which the elliptic curve logarithm can be reduced to the case of discrete logarithms in a multiplicative group (DLP). When combined with sub-exponential algorithms for solving the classical DLP, this yields a probabilistic subexponential running time for computing elliptic curve logarithms on supersingular curves. This was a finding due to Menezes, Okamoto and Vanstone (MOV) in 1991, in which they showed how the ECDLP could be reduced to classical DLP in an extension of a multiplicative group $\mathbf{F}_q$[1][2][9].

The other class of curves, the anomalous curves, allows an even more efficient attack when applicable. Proposed independently in 1998 by Satoh and Araki, Semaev, and the following year by Smart, this type of curves allow the ECDLP to be solved in polynomial time by reducing it to the classical DLP in an additive group $\mathbf{F}_q$ [1][2][9].

2.7 Optimizing ECC Implementations

To get efficient elliptic curve cryptosystem, some important issues must be addressed before implementing that affect the efficiency of the computations. These include selection of elliptic curve domain parameters (underlying finite

field, field representation, and elliptic curve), selecting suitable coordinate systems, and choosing efficient algorithms for exponentiation which is the most elliptic curve operation.

2.7.1 Domain Parameters

When setting up an elliptic curve cryptosystem, there are three basic decisions that need to be made:

1. Selection of the underlying finite field $\mathbf{F_q}$.
2. Selection of the representation for the elements of $\mathbf{F_q}$.
3. Selection of the elliptic curve E over $\mathbf{F_q}$.

2.7.1.1 Selection of the Underlying Finite Field $\mathbf{F_q}$

The field operations of modular addition and subtraction are relatively fast and easily implemented. However, modular multiplication (which requires a modular reduction) and modular inversion are much more time consuming. The following remarks discuss how the choices of the underlying field, and its representation.

1. Three kinds of finite fields $\mathbf{F_q}$ are especially suitable for elliptic curve cryptosystem (ECC), binary fields $\mathbf{F}_{2m}$, prime fields $\mathbf{F}_p$, and optimal extension fields (OEF) $\mathbf{F}_{pm}$.

2. The arithmetic operation on $\mathbf{F}_p$ is the usual addition, subtraction and multiplication modulo *p*. However using standard modular arithmetic is not very efficient since multi-precision remaindering operations are very expensive. Hence when used in elliptic curve systems there are various choices that are often made:

General Primes For general primes the most efficient implementation technique is almost always to use Montgomery Arithmetic. Montgomery arithmetic uses a

special representation to perform efficient arithmetic, the division and remaindering essentially being performed by bit shifting.

Generalized Mersenne Primes Certain primes are highly suited for efficient reduction techniques, the most simple form of such primes being the Mersenne primes, which are primes of the form $p = 2^k - 1$. However the number of Mersenne primes of the correct size for cryptography is limited[30].

3. Looking at $\mathbf{F}_{2m}$ as a vector space of dimension m over $\mathbf{F}_2$, the elements of $\mathbf{F}_{2m}$ can be represented as binary vectors (or strings) of length m, given a suitable basis of this vector space. This makes it easy to store data in hardware (ideally in shift registers of length r). Addition in $\mathbf{F}_{2m}$ can be performed in one clock cycle by bitwise XOR-ing the operands.

4. In software environments in which an arithmetic processor is already available for modular exponentiation, the performance of $\mathbf{F}_p$ can be improved so that in some cases it exceeds the performance of $\mathbf{F}_{2m}$. This holds true for platforms such as those using Pentium processors or, in the case of smart cards, those having a crypto coprocessor to accelerate modular arithmetic.

5. If the field $\mathbf{F}_{2m}$ is selected as the underlying finite field, then there are many ways in which the elements of $\mathbf{F}_{2m}$ can be represented. The two most efficient ways are an *optimal normal basis representation* and a *polynomial basis representation*.

6. When using a normal basis representation for the elements of $\mathbf{F}_{2m}$, squaring a field element becomes a simple cyclic shift of the vector representation, and thus the multiplication count in adding two points is reduced.

2.7.1.2 Selection of an Suitable Elliptic Curve

To obtain secure ECC, elliptic curve E defined over a finite field $\mathbf{F}_q$ must satisfy the following conditions:

1. To resist the Pollard ρ-attack mentioned, $\#E(\mathbf{F}_q)$ should be divisible by a sufficiently large prime n (for example, $n > 2^{160}$).

2. To resist the Semaev–Smart–Satoh–Araki attack, $\#E(\mathbf{F}_q)$ should not be equal to q.

3. To resist the MOV reduction attack, n should not divide q^k-1 for all $1 \leq k \leq C$, where C is large enough so that it is computationally infeasible to find discrete logarithms in $\mathbf{F}^*_q c$.

Indeed, there are four techniques for selecting an appropriate elliptic curve[2].

1) Using Hasse's Theorem. This technique can be used for picking curves over $\mathbf{F}_{2m}$ where m is divisible by a small integer $l \geq 1$.

To select an appropriate curve over $\mathbf{F}_{2m}$, we first pick an elliptic curve over a small field $\mathbf{F}_{2^l}$, where l divides m, compute $\#E(\mathbf{F}_{2^l})$ exhaustively, and then use Hasse's theorem to determine $\#E(\mathbf{F}_{2m})$. If conditions (1), (2) and (3) above (with $q = 2^m$) are not satisfied, then another curve is selected and the process is repeated.

2) The Global Method. Another possibility is to choose an elliptic curve defined over a number field and then reduce it modulo a prime ideal such that the resulting curve over a finite field satisfies conditions (1), (2) and (3).

3) Multiplication Method. The method of complex multiplication (CM) allows the choice of an elliptic curve order before the curve is explicitly constructed.

Thus, orders can be generated and tested to satisfy conditions (1), (2) and (3); a curve is constructed only when these conditions are met.

4) Choosing a Curve at Random. Another approach to selecting an appropriate elliptic curve E over $\mathbf{F}_q$ is to select random parameters $a, b \in \mathbf{F}_q$ such that $(4a^3 + 27b^2) \neq 0$. One then computes $u = \#E(\mathbf{F}_q)$ and factors u. This process is repeated until conditions (1), (2) and (3) are satisfied.

2.7.2 Coordinate Systems

One of the crucial decisions when implementing an efficient elliptic curve cryptosystem is deciding which point coordinate system to use. The point coordinate system used for addition and doubling of points on the elliptic curve determines the efficiency of these routines, and hence the efficiency exponentiation.

Affine coordinates are the simplest to understand and are used for communication between two parties because they require the lowest bandwidth. The drawback of affine coordinates is, that the required field inversion is very costly compared to multiplications and squarings. So, alternative coordinate systems such as Projective coordinates can be used to avoid inversions.

Table 2.2 shows the computational complexity of point addition and doublings in two coordinate systems on the elliptic curve over $\mathbf{F}_p$, where M, S and I denote field multiplication, squaring and inversion respectively, and the cost of field additions and subtractions are ignored.

Coordinate	Addition			Doubling		
	M	S	I	M	S	I
Affine	2	1	1	2	2	1
Projective	12	2	0	7	3	0

Table 2.2 The computational complexity of affine and projective coordinate systems.

Cohen et al. [7] recommended the idea of mixed coordinates, where the inputs and outputs to point additions and doublings may be in different coordinates.

2.7.3 Exponentiation

Elliptic curve exponentiation can be computed by repeating additions and doublings, where the average number of addition of elliptic points operations depends on the minimal hamming weight of the exponent. The minimal hamming weight representation of exponent can be obtained by using windowing method, and mixed with the addition-subtraction method for reducing the number of additions [10][20].

CHAPTER 3

3 Elliptic Curve Exponentiation

Elliptic curve exponentiation is the operation of computing kP for a given point P on an elliptic curve and integer k. It is the primary operation in elliptic curve cryptosystem such as ECDH, which is denoted by

$$Q = kP$$

where Q, P are points on an elliptic curve and k is an integer; the cost of executing such cryptosystems depends mostly on the complexity of exponentiation. Thus, the performance and execution time of such elliptic curve cryptosystems is primarily determined by using efficient algorithms for exponentiation.

One approach to speed up the elliptic curve exponentiation kP, $P \in E(\mathbf{F}_q)$ is by reducing of the number of additions. It is possible to reduce the number of additions by recoding the integer k into a representation having minimal number of nonzero digits [10].

Next section presents how to represent integer k with minimal number of nonzero digits.

3.1 Base-2 Representations of Integers

The involved exponent of elliptic curve exponentiation operation is positive integer. There are several ways to represent an integer rather than well known decimal representation. One of these representations is so-called base-2 representations where the integer is represented by the sum of multiple powers of

two. In base-2 representations digits other 0 and 1 are permitted. The set of digits is called digit set and denoted by D [12].

Definition 3.1 The sequence $(k_{t-1}, \ldots, k_0)$ is called a D-representation of the integer k, if

$$k = \sum_{i=0}^{t} k_i.2^i \quad and \quad k_i \in D\ ,\ \forall i = 0, \ldots, t-1. \tag{3.1}$$

where D is called the digit set and the number of elements in the digit set, i.e. its order is denoted by |D| [12].

If D = {0, 1} holds, then we give the simplest base-2 representation which is the uniquely determined binary representation. The length of this representation, the so-called bit length t is calculated as $t = \lfloor log_2\ k \rfloor + 1$. The k_i are called bits, which is short for binary digits. If D = {0, ±1} holds, the representation is also called a signed binary representation. More general, If D = {0,±1, . . . ,±x} holds, the representation is also called a signed representation. For example the sequence (1, 0, 1, 0, 1, 1, 0) is binary representation of 86 with bit length 6, since

$$86 = 1*2^6 + 0*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 1*2^1 + 0*2^0$$

In general, D-representations loose the property of uniqueness. For example $(1, 0, 1, 1, 1, \bar{1})$ and $(1, 1, 0, \bar{1}, 0, 1)$ are both signed binary representations of 45 with bit length 6, where $\bar{1} = -1$.

It is necessary to measure the quality of representations. This can be done by using the weight of either one D-representation separately or several D-representations at once.

Definition 3.2 *Let $k = (k_{t-1}, \ldots, k_0)$ be a* D*-representation with bit length t. The Hamming weight of k is the number of non-zero digits in k and denoted by Hw(k). The Hamming density of k is given as Hd(k) = Hw(k)/n. The average Hamming density of a class of D-representations χ is the expected Hamming density of a randomly chosen D-representation in χ with bit length $n \to \infty$ and denoted by AHd(χ).*

3.1.1 Signed Binary Representation

Signed binary representation is redundant binary representation. It does not exhibit a unique minimal form. Algorithms to generate a minimal representation are widely reported for exponentiation, and multiplication [12].

In 1951 Booth presented an algorithm to multiply two numbers by converts a 2's complement binary number to a signed binary digit set D = {0, ±1}. Booth algorithm scans the bits from right to left, and replaces a consecutive block of several 1's by a block of 0's and $\bar{1}$ according to $\left(\underbrace{1,\ldots,1}_{a}\right) \to \left(1,\underbrace{0,\ldots,0}_{a-1},\bar{1}\right)$ [30].

Example 3.3 Let k = 221 with binary representation (1, 1, 0, 1, 1, 1, 0, 1) then Booth algorithm convert this binary representation to $(1, 0, \bar{1}, 1, 0, 0, \bar{1}, 1, \bar{1})$.

However, Booth recoding has a challenging, if two blocks of 1's are separated by an isolated 0, the Booth algorithm does not use the fact that $(1,\bar{1}) \equiv (0,\bar{1})$. For example, $\left(\underbrace{1,\ldots,1}_{a},0,\underbrace{1,\ldots,1}_{b}\right)$ is replaced by $\left(\underbrace{1,0,\ldots,0}_{a-1},\bar{1},\underbrace{1,0,\ldots,0}_{b-1},\bar{1}\right)$ and not by $\left(\underbrace{1,0,\ldots,0}_{a},\bar{1},\underbrace{0,\ldots,0}_{b-1},\bar{1}\right)$. Therefore Booth recoding output is not sparse, and the Hw of exponent k of large t bits, is (t +1)/2 on average which is not minimal [5].

Later, in 1960, through his investigations on how to reduce the number of additions and subtractions used in binary multiplication and division, Reitwiesner presented a method to convert an exponent k from binary to its canonical form of signed binary representation so-called NAF[10][13].

Definition 3.4 *A signed binary representation is said to be non-adjacent form (NAF), if no two adjacent digits are nonzero*.

Reitwiesner scans the bits from right to left, and replaces a consecutive block of several 1's by a block of 0's and $\bar{1}$ according to $\left(1,\underbrace{1,\ldots,1}_{a}\right)\rightarrow\left(1,\underbrace{0,\ldots,0}_{a-1},\bar{1}\right)$, If two blocks of 1's are separated by an isolated 0, the Reitwiesner algorithm uses the fact that $(1,\bar{1})\equiv(0,\bar{1})$. For example, $\left(\underbrace{1,\ldots,1}_{a},0,\underbrace{1,\ldots,1}_{b}\right)$ is replaced by $\left(1,\underbrace{0,\ldots,0}_{a},\bar{1},\underbrace{0,\ldots,0}_{b-1},\bar{1}\right)$ and not by $\left(1,\underbrace{0,\ldots,0}_{a-1},\bar{1},1,\underbrace{0,\ldots,0}_{b-1},\bar{1}\right)$. So Reitwiesner's algorithm is also known as Booth canonical recoding algorithm [8].

Example 3.5 Let k = 221 with binary representation (1, 1, 0, 1, 1, 1, 0, 1) The NAF of k is generated as follows

$$
\begin{array}{cccccccccc}
 & 1 & 1 & 0 & 1 & \underline{1} & \underline{1} & 0 & 1 & \\
 & & & & & \underline{1} & 0 & \bar{1} & & \\
 & & & 1 & 1 & 0 & & & & \\
 & & 1 & 0 & \bar{1} & & & & & \\
 & 1 & 0 & & & & & & & \\
\hline
1 & 0 & 0 & \bar{1} & 0 & 0 & \bar{1} & 0 & 1 &
\end{array}
$$

Hence, the NAF of 221 is given as $(1,0,0,\bar{1},0,0,\bar{1},0,1)$.

Algorithm 3.1 (Reitwiesner algorithm), shows how to generate non-adjacent form (NAF) of exponent k from binary form of k.

Algorithm 3.1 Generation of NAF [13]

INPUT: An t-bit exponent k in its binary representation $(k_{t-1}, k_t, ..., k_0)$

OUTPUT: The NAF of $(\mu_{t-1}, \mu_t, ..., \mu_0)$ *of* k

1. $c_0 \leftarrow 0; k_{t+1} \leftarrow 0; k_t \leftarrow 0$
2. for i from 0 to t do

2.1 $c_{i+1} \leftarrow \lfloor (c_i + k_i + k_{i+1})/2 \rfloor$

2.2 $\mu_i \leftarrow c_i + k_i - 2c_{i+1}$

3. return The NAF $(\mu_{t-1}, \mu_t, ..., \mu_0)$ *of* k

Reitwiesner's proved the NAF propriety of his output, and this representation is unique and has minimal Hw [13]. Morain and Olives [20] showed that on average the minimal Hw of exponent t-digits k in binary signed representation is equal to $\frac{(t+1)}{3}$.

3.2 Algorithms for Elliptic Curve Exponentiation

Since elliptic curve exponentiation kP, where k is a positive integer and P a given point on elliptic curve is defined as

$$kP = \underbrace{P + P + P + \cdots + P}_{k \text{ times}}$$

For large integer *k*, computing exponentiation *k*P for a given point P on an elliptic curve is costly endeavor, and it is inefficient to use straightforward summation technique that requires (k-1) elliptic additions, so other techniques should be used to efficiently compute exponentiation.

3.2.1 Binary Methods

Binary methods are the standard algorithms for the efficient computation of an exponentiation. These algorithms use the binary representation of the exponent. There are two different binary methods, one that scans the bits of the exponent from right-to-left and other from left-to-right. A doubling is performed at each step, but performing addition depends on the scanned bit value, so it called double and add algorithms.

3.2.1.1 Right-to-Left Binary Method

Let P a point in elliptic curve, and t-bit exponent k in its binary representation. It possible to compute

$$\begin{aligned} kP &= (k_{t-1}2^{t-1} + k_{t-2}2^{t-2} + ...+ k_1 2 + k_0)\, P \\ &= k_{t-1}2^{t-1}P + k_{t-2}2^{t-2}P + ...+ k_1 2\, P + k_0 P \end{aligned} \tag{3.2}$$

Algorithm 3.2 is adapted from [13] to present right-to-left binary method.

Algorithm 3.2 Right-To-Left Binary Method [13]

INPUT : an element $P \in E(\mathbf{F}_q)$, t-bit exponent k in its binary representation.
OUTPUT : kP

1. $X \leftarrow O$ (where O is infinity)
2. $Q \leftarrow P$
3. For i from 0 to t-1 do the following
   3.1 if $k_i = 1$ then X← ECADD(X, Q)
   3.2 Q←ECDBL(Q)
4. return (X)

Algorithm 3.2 computes the exponentiation kP starting at the least significant bit k_0, and performs an ECADD operation each time the current bit k_i is 1, hence

with probability 1/2. An ECDBL operation is performed in each iteration. Therefore the right-to-left binary method on average requires

$$t \text{ ECDBL} + t \cdot \tfrac{1}{2} \text{ECADD} \quad \text{operations.}$$

Example 3.6 Let the exponent k = 18 with binary representation (1, 0, 0, 1, 0). The following table displays the values of X, Q, k_i during each iteration of algorithm 6 for computing 18P.

i	0	*1*	*2*	*3*	*4*	Finally
k_i	0	*1*	*0*	*0*	*1*	-
Q	*P*	*2P*	*4P*	*8P*	*16P*	*32P*
X	*O*	*2P*	*2P*	*2P*	*18P*	*18P*

Table 3.1 The values of X, Q during the iterations of right-to-left binary method

Right-to-left binary method can be generalized to work with D-representations. Algorithm 3.3 is adapted from [18] shows the general right-to-left binary method.

Algorithm 3.3 General Right-To-left Binary Method [18]

INPUT : an element P $\in$ E($\mathbf{F}_q$), and t-digits exponent k in D-representation

OUTPUT : kP

1. $X \leftarrow O$
2. $Q_d \leftarrow dP$, $\forall d \in D^*$
3. For i from 0 to t-1 do the following

   3.1 if $k_i \neq 0$ then $X \leftarrow$ ECADD(X, Q_{k_i})

   3.2 $Q_d \leftarrow$ ECDBL(Q_d) , $\forall d \in (D - \{0\})$
4. return (X)

We noticed Algorithm 3.3 precompute all points in the form dP, d $\in$ D-{0,1} then performs an ECADD operation each time the current digit e_i is non-zero,

hence with probability *AHd(χ)*. Since the point $d \cdot 2^i P$, $d \in D$ has to be added in the i-th iteration, all the |D|-1 points have to be doubled in each iteration. On average, the general right-to-left binary method requires

$$t(|D|-1)\ \text{ECDBL} + t \cdot \text{AHd}(c)\text{ECADD}$$

operations to compute a exponentiation kP. Also in this case, additional ECADD and ECDBL operations are required for the precomputation.

3.2.1.2 Left-to-Right Binary Method

The basic idea of the left-to-right binary method is to successively factor out 2 in (3.2), which yields

$$\begin{aligned} kP &= k_{t-1}2^{t-1}P + k_{t-2}2^{t-2}P + \ldots + k_1 2P + k_0 P \\ &= 2(k_{t-1}2^{t-2}P + k_{t-2}2^{t-3}P + \ldots + k_1 P) + k_0 P \\ &\quad M \\ &= 2(2(K\ 2(k_{t-1}2P + k_{t-2}P) + \ldots) + k_1 P) + k_0 P \\ &= 2(2(K\ 2(2(k_{t-1}P) + k_{t-2}P) + \ldots) + k_1 P) + k_0 P \end{aligned} \qquad (3.3)$$

Now it is possible to start the evaluation at the most significant bit k_{t-1}, i.e. left-to-right. In the i-th iteration, the intermediate result *Q* is doubled and if the current bit k_i is 1, P is added as shown in Algorithm 3.4.

Algorithm 3.4 Left-To-Right Binary Method [13][18]

INPUT : an element $P \in E(\mathbf{F}_q)$, t-bit exponent k in its binary representation.

OUTPUT : kP

1. $Q \leftarrow O$
2. For i from t-1 down 0 do the following
   2.1 Q ←ECDBL(Q)
   2.2 if $k_i = 1$ then Q←ECADD(Q,P)
3. return (Q)

Algorithm 3.4 performs an ECADD operation each time the current bit k_i is 1, hence with probability 1/2. An ECDBL operation is performed in each iteration. Therefore, to compute exponentiation, the left-to-right binary method on average requires

$$t \text{ ECDBL} + t \cdot \tfrac{1}{2} \text{ECADD} \quad \text{operations.}$$

Example 3.7 Let the exponent k = 18 with binary representation (1, 0, 0, 1, 0). Table 3.2 displays the values of Q, k_i during each iteration of algorithm 3.4 for computing 18P.

i	*4*	*3*	*2*	*1*	0
k_i	*1*	*0*	*0*	*1*	0
ECDBL	*O*	*2P*	*4P*	*8P*	*18P*
ECADD	*P*	-	-	*9P*	-
Q	*P*	*2P*	*4P*	*9P*	*18P*

Table 3.2 The value of Q during the iterations of left-right binary method

Left-to-right binary can also be generalized to work with D-representations as Algorithm 3.5 which is adapted from[18][23].

Algorithm 3.5 General Left-To-Right Binary Method [18][23]

INPUT : an element $P \in E(\mathbf{F}_q)$, and t-digits exponent k in D-representation

OUTPUT : kP

1. $Q \leftarrow O$ (where O is infinity)
2. $Q_d \leftarrow dP$, $\forall d \in D^*$
3. For i from t-1 to 0 do the following

   3.1 $Q \leftarrow ECDBL(Q)$,

   3.2 if $k_i \neq 0$ then $Q \leftarrow ECADD(Q, Q_{k_i})$

4. return (Q)

We noticed that algorithm 3.5 precompute all points in the form dP, d ∈ D-{0, 1}, then performs an ECADD operation each time the current digit k_i is non-zero, hence with probability *AHd(χ)*. Also, one ECDBL operation is performed in each iteration to double the intermediate result. On average, the general left-to-right binary method requires

$$t \text{ ECDBL} + t \cdot AHd(X) \text{ECADD}$$

operations to compute a exponentiation kP. Also in this case, additional ECADD and ECDBL operations are required for the precomputation.

3.2.1.3 Why Left-to-Right

From Algorithms (3.2, 3.4), we noticed that both methods require the same amount of ECADD and ECDBL operations, but right-to-left binary method requires one additional register X to store $2^i P$.

In the case of the general methods:

1) The left-to-right binary method requires only one ECDBL operation in each iteration, while the right-to-left binary method requires one ECDBL operation for each precomputed point in each iteration. This means, that the right-to-left binary method requires (|D|-1) times more ECDBL operations than its left-to-right counterpart.

2) The precomputed points for the ECADD step in left-to-right remain fixed during the whole runtime. So it is possible to use mixed coordinates as in [7] for the ECADD step. We can conclude that left-to-right algorithms are preferable.

3.2.1.4 Exponentiation with Precomputation

If there is extra memory, then it is possible to use general binary algorithms. As it turned out, the precomputation of several points is required by both algorithms, depending on the D-representation used for the exponent, so additional ECADD and ECDBL operations are required for the precomputation stage.

In general binary methods, and D-representation used for the exponent, the number of points to precompute equal (|D| - 1), because the points dP is computed for all d $\in D^*$.

As explained in Section 2.2 inversions of points on an elliptic curve can be computed virtually for free just by changing the sign of the y-coordinate. When the signed representations of the exponent used, the number of points to precompute can be reduced by more than 50%, because the points $|d|P$ is computed only for d $\in$ digit-set D and stored in the precomputation stage.

Next sections present three general left-to-right methods with signed representation of exponent.

3.2.2 Sliding Window applied on NAF

Sliding window method is an approach for computing exponentiation with prcomputations. It generalizes binary method and is parameterized by a positive integer *w*, where the case *w* = 1 is the same as binary method. Sliding window method can recode the binary representation of the exponent by such windows yields a D-representation of exponent. In sliding window method there is no reasons to force the windows to be the next to each other, fewer windows of width

up to *w* can suffice to cover all non-zero exponent bits if strings of zeros are skipped, Moreover, one can arrange for all windows to be odd-valued (i.e., have a 1 as the rightmost bit). Then the bits covered by each single window correspond to a value in the digit-set D = {1, 3,..., 2^w - 1}.

It is possible to scan the binary representation of the exponent from left to right, or from right to left, starting a new window whenever a non-zero bit is encountered, choosing the maximum width up to *w* for this particular window such that the rightmost bit is also non-zero.

Example 3.8 Let k = 221 with binary representation (1, 1, 0, 1, 1, 1, 0, 1). Then left-to-right scanning with window width *w* =3, can convert d as follows

$$\underline{1\ 1}\ \ 0\ \underline{1\ 1\ 1}\ \ 0\ 1$$
$$3\ \ 0\ 0\ 0\ 7\ \ 0\ 1$$

Hence, yields a representation of k = 221 is given as 0 3 0 0 0 7 0 1 .

Such left-to-right scanning or right-to-left scanning yields a representation

$$k = \sum_{i=0}^{t} k_i 2^i, \quad k_i \in D$$

and the average density of nonzero digits of both representations is equal to

$$\frac{1}{w+1}$$ [18].

Koyama and Tsuruoka [14] suggested the application of a sliding window scheme on binary signed-digit representation as NAF to obtain a signed recoding with smaller Hamming weight. De Win et al.[33] applied the sliding window method directly on NAF giving smaller digit set D ={0,±1,±3, . . . ,±d_{max}}, where d_{max} is the largest odd NAF consisting of at most w digit equals $\frac{1}{3}(2^{w+1}-1)$ for

odd w, and $\frac{1}{3}(2^{w+1}+1)-2$ for even w. Algorithm 3.6 describes the sliding window applied on NAF as stated in [33].

Algorithm 3.6 Sliding Window applied on NAF [33]

INPUT: an element P ∈ E($\mathbf{F}_q$), NAF $(\mu_{t-1},\mu_t,...,\mu_0)$ of k , window width w ≥ 1.

OUTPUT: kP

precomputation stage:

1. P_1 ← P, P_2 ← ECDBL(P)
2. For j from 1 to $\frac{(2^w+(-1)^{w+1})}{3}-1$ do

   P_{2i+1} ← ECADD(P_{2i-1} , P_2)

Evaluation stage

3. Q ← O
4. i ← t-1
5. while i ≥ 0

   5.1 If μ_i = 0 then

   5.1.1 Q ← ECDBL(Q)

   5.1.2 i ← I -1

   5.2 else

   5.2.1 s ← max(i - w + 1, 0)

   Let l be the smallest integer such that l ≥ s and $\mu_l \neq 0$

   5.2.2 for n from 1 to i-*l*+1 do Q ← ECDBL(Q)

   5.2.3 if $(\mu_i,\mu_{i-1},...,\mu_l) > 0$ then Q ←ECADD(Q , P$(\mu_i,\mu_{i-1},...,\mu_l)$)

   5.2.4 else if $(\mu_i,\mu_{i-1},...,\mu_l) < 0$ then Q ← ECADD(Q ,- P$\left|(\mu_i,\mu_{i-1},...,\mu_l)\right|$)

   5.2.5 for n from 1 to *l*-s do

   Q ← ECDBL(Q)

   5.2.6 i ← s -1

6. Return Q

3.2.3 The width-w Non Adjacent Form (*w*NAF)

Blake, Seroussi and Smart and Solinas, proposed independently *w*NAF that is computed directly from binary strings using a generalization of NAF recoding for *w*>2. [18]

Definition 3.9 (*w*NAF) *A sequences of signed digits is called wNAF iff the following three properties hold*:

(1) The most significant non-zero bit is positive.

(2) Among any w consecutive digits, at most one is non-zero.

(3) Each non-zero digit is odd and less than 2^{w-1} *in absolute value.*

If w=2, 2NAF can simply call NAF [18].

3.2.3.1 Generation of *w*NAF

Algorithm 3.7 describes the generation of the *w*NAF from the decimal representation as stated [18].

Algorithm 3.7 Generation of *w*NAF [18]

INPUT: width w, an t-bit integer k in its decimal representation.

OUTPUT: The wNAF $(\delta_t, \delta_{t-1}, \ldots \delta_0)$ of k

1. i← 0

2. while k ≥ 1 do

if k is even then

$\delta_i = 0$

else

$\delta_i \leftarrow mods\ 2^w$; $k \leftarrow k - k_i$

k= k /2 ; i← i+1

3. return $(\delta_t, \delta_{t-1}, \ldots \delta_0)$

Algorithm 3.7 generates a *w*NAF of exponent k from least significant bit that is right-to-left generation, such that at most one of any *w* consecutive digits is non-zero. This algorithm uses the signed modulo operation such that k is odd each time a signed modulo operation is performed. For example, the representation:

$$k = (1,1,1,1,0,0,1,0,1,1,0,0,1,0,1) \tag{3.4}$$

with window size $w = 3$, is converted to 3 NAF representation:

$$k = (1,0,0,0,\overline{1},0,0,0,3,0,0,0,\overline{3},0,0,\overline{3}) \tag{3.5}$$

*w*NAF have an average density of nonzero digits of $1/(w + 1)$ for $t \rightarrow \infty$, and the signed-digit set $D = \{0,\pm1,\pm3, \ldots ,\pm(2^{w-1} - 1)\}$ where *t* is the bit-length of the binary form *of* exponent k [18]. In [22], Muir and Stinson proved that the Hw of a exponent given in its *w*NAF is minimal for any choice of *w*. This implies that the AHd of the *w*NAF is minimal amongst all D-representations. Therefore *w*NAF are optimal in the terms of time and memory for $w > 3$. Muir and Stinson [22] also proved the following these properties of the *w*NAF

(1) *w*NAF *representation is unique except for the number of leading zeros*

(2) Every integer can represented as *w*NAF.

(3) An integer's *w*-NAF is at most one digit longer than its binary representation.

3.2.3.2 Exponentiation with *w*NAF

Algorithm 3.8 is adapted from [26] describes the exponentiation with wNAF

Algorithm 3.8 Exponentiation with *w*NAF [26]

INPUT: an element P ∈ E($\mathbf{F}_q$), wNAF $(\delta_t, \delta_{t-1}, \ldots \delta_0)$ of k , window size w ≥ 2.

OUTPUT: kP

precomputation stage

1. P_1 □ P
2. P_2 □ ECDBL(P)
3. For i from 1 to 2^{w-2} -1 do

   P_{2i+1} □ ECADD(P_{2i-1} , P_2)

Evaluation stage

4. Q□ O
5. For i from t down to 0 do

   5.1 Q □ ECDBL(Q)

   5.2 if $\delta_i > 0$ then Q □ECADD(Q , P_{δ_i})

   5.3 else if $\delta_i < 0$ then Q □ ECADD(Q ,- $P_{|\delta_i|}$)
6. Return Q

Although there are slightly more point operations needed to evaluate the exponentiation if the exponent is represented in *w*NAF compared to the [33] representation, the required precomputation is less in the *w*NAF case because of the smaller digit set. Blake et al. proved that *w*NAF is asymptotically better than sliding window on NAF schemes if w > 3 [27].

3.2.4 The width-w Mutual opposite Form (*w*MOF)

As we pointed out in subsection 3.2.1 that left-to-right exponentiation algorithms is preferable, But all algorithms for generating *w*NAF, need carry-overs due to the recoding is restricted to be done right-to-left due to additional memory *O*(n) to store the recoded string before starting the left-to-right evaluation of the exponentiation. Hence it is an important task to recoding the exponent from left to right, this enables the recoding, and evaluation stage in general left-right binary

method to be merged without storing the recoded exponent, this reduce memory space [27].

Joye and Yen [13] proposed a left-to-right binary recoding algorithm, but it has been an unsolved problem to generate a left-to-right recoding for $w > 2$. wMOF is the first left-to-right signed recoding scheme for $w > 2$ that is constructed by applying left-to-right sliding window method on Mutual opposite form (MOF), and it is efficient as wNAF [27].

3.2.4.1 The Mutual opposite Form (MOF)

Okeya et al [27] defined a new canonical binary signed-digit representation called the mutual opposite form (MOF). MOF is equal Booth recoding but can compute in any order.

Definition 3.10 The *n-bit mutual opposite form (MOF) is an n-bit signed binary string that satisfies the following properties:*

1. *The signs of adjacent non-zero bits (without considering zero bits) are opposite.*
2. *The most non-zero bit and the least non-zero bit are 1 and, $\bar{1}$ respectively, unless all bits are zero.*

For example the representation $0100\bar{1}01000\bar{1}001\bar{1}0$, is of MOF. It has zero bits inserted between non-zero bits that have a mutual opposite sign.

by (t +1)-bit MOF, and the average non-zero density of t-bit MOF is 1/2 for $t \to \infty$. They proved also that the operation $\mu = 2k \square k$ converts binary string k to

its MOF, where '□'stands for a bitwise subtraction. Algorithm 3.9 shows how MOF is computed.

Algorithm 3.9 Generation MOF from Binary [27]

*INPUT: A*n t-bit exponente in its binary representation $(k_{t-1}, k_t, ..., k_0)$

OUTPUT: MOF of $(\mu_{t-1}, \mu_t, ..., \mu_0)$ *of k*

1. $\mu_i \leftarrow -k_{i-1}$
2. *for i from t-1 down to 1 do*

$\mu_i \leftarrow k_{i-1} - k_i$

3. $\mu_i \leftarrow -k_0$
4. *return The MOF* $(\mu_{t-1}, \mu_t, ..., \mu_0)$ *of k*

3.2.4.2 Generation of The width-w Mutual opposite Form (*w*MOF)

In order to apply left-to-right sliding window method on MOF, Okeya and Takagi [27] defined the conversions for MOF windows of length $l = 2, 3, \ldots, w$, such that the first and the last bit is non-zero. If $l < w$ holds, the window is filled with closing zeros instead of leading ones. These conversions lead to generate complete conversion table, for example, the look-up table width 4 is as following:

$$1000\} \rightarrow 1000 \quad 1\bar{1}00 \rightarrow 0100 \quad \left.\begin{matrix} 1\bar{1}10 \\ 10\bar{1}0 \end{matrix}\right\} \rightarrow 0030 \quad \left.\begin{matrix} 1\bar{1}01 \\ 1\bar{1}1\bar{1} \end{matrix}\right\} \rightarrow 0005 \quad \left.\begin{matrix} 100\bar{1} \\ 10\bar{1}1 \end{matrix}\right\} \rightarrow 0007$$

$$\bar{1}000\} \rightarrow \bar{1}000 \quad \bar{1}100 \rightarrow 0\bar{1}00 \quad \left.\begin{matrix} \bar{1}1\bar{1}0 \\ \bar{1}010 \end{matrix}\right\} \rightarrow 00\bar{3}0 \quad \left.\begin{matrix} \bar{1}10\bar{1} \\ \bar{1}1\bar{1}1 \end{matrix}\right\} \rightarrow 000\bar{5} \quad \left.\begin{matrix} \bar{1}001 \\ \bar{1}01\bar{1} \end{matrix}\right\} \rightarrow 000\bar{7}$$

In general when the look-up table width *w* is used, then the signed-digit set D = $\{0, \pm1, \pm3, \ldots, \pm(2^{w-1} - 1)\}$ which is minimal as wNAF. Therefore, the scheme

requires only 2^{w-2} precomputed elements. Now we give the definition of *w*MOF as in [27].

Definition 3.11 *A sequence of signed digits is called wMOF iff the following three properties hold:*

1. *The most significant non-zero bit is positive.*
2. *All but the least significant non-zero digit x are adjoint by w-1 zeros as follows:*

   – *in case of* $2^{s-1} < |x| < 2^s$ *for an integer* $2 \le s \le w - 1$ *the pattern*

   – *equals* $x\underbrace{0\ldots0}_{s}\ \underbrace{0\ldots0}_{w-s-1}$

   – *in case of* $|x| = 1$, *either the pattern equals* $x\underbrace{0\ldots0}_{w-1}$ *and the next lower non-zero digit has opposite sign from x or the pattern equals* $0x\underbrace{0\ldots0}_{w-2}$ *and the next lower non-zero digit has the same sign as x.*

If x is the least significant non-zero digit, it is possible that the number of right-hand adjacent zeros is smaller than stated above. In addition, it is not possible that the last non-zero digit is a 1 following any non-zero digit.

3. *Each non-zero digit is odd and less than* 2^{w-1} *in absolute value.*

The following algorithm is proposed by [27] to generate *w*MOF.

Algorithm 3.10 Generation *w*MOF from MOF [27]

INPUT: width w, t-bit exponent k in its MOF $(k_{t-1}, k_t, \ldots, k_0)$

OUTPUT : wMOF of k = $(\delta_t, \delta_{t-1}, \ldots \delta_0)$

1. $k_{-1} \leftarrow 0$; $i \leftarrow t$
2. While $i \ge w - 1$ do

if $k_i = k_{i-1}$ then

2.1.1 $k_i \leftarrow 0$; $i \leftarrow i-1$

else {The MOF window begins with a non-zero digit left hand}

2.2.1 $(\delta_i, \delta_{i-1}, \ldots \delta_{i-w+1}) \leftarrow$ Table $_{wSW}(k_{i-1} - k_i, k_{i-2} - k_{i-1}, \ldots, k_{i-w} - k_{i-w+1})$

2.2.2 $i \leftarrow i-w$

3. if $i \geq 0$ then

3.1 $(\delta_i, \delta_{i-1}, \ldots \delta_0) \leftarrow$ Table $_{i+1SW}(k_{i-1} - k_i, k_{i-2} - k_{i-1}, \ldots, k_0 - k_1, -k_0)$

4. return $(\delta_t, \delta_{t-1}, \ldots \delta_0)$.

Algorithm 3.10 generates a *w*MOF of exponent from most significant bit by applying sliding window left-to-right and using the conversion table, for example, exponent k = 619 has binary the representation

$$k = (1,0,0,1,1,0,1,1,1,1) \tag{3.6}$$

with window size $w = 3$, is converted to 3 MOF representation:

$$k = (0,1,0,0,0,3,0,0,3,0,\bar{1}) \tag{3.7}$$

Okeya and Takagi [27] proved that every non-negative integer k has a representation as wMOF, which is unique except for the number of leading zeros

Theorem 3.12 *For $t \to \infty$, the average non-zero density of wMOF is asymptotically 1/(w+1)* [27].

Proof The AHd is the average density of non-zero digits of a randomly chosen wMOF with bit length $t \to \infty$. This density is given as the average number of non-zero digits divided by the average number of digits written out by algorithm 3.10 Two cases exist:

$k_i = 0$: In this case only one digit is written out, which is zero.

$k_i \neq 0$: In this case w digits are written out, one non-zero and w - 1 zero.

Since AHd(MOF) = 1/2, both cases appear each with a probability of 1/2.

Therefore, the AHd of the wMOF is given as

$$\text{AHd}(w\text{MOF}) = \frac{\frac{1}{2}.0+\frac{1}{2}.1}{\frac{1}{2}.1+\frac{1}{2}.w} = \frac{1}{w+1} \qquad \square$$

Avanzi [3] proved that the Hw of a exponent given in its *w*MOF is minimal for any choice of *w*. This implies, that the AHd of the *w*MOF is minimal amongst all D-representations which use the digit set D = $\{0,\pm1,\pm3, \ldots ,\pm2^{w-1} - 1\}$.

Finally, we compare the characterizing properties for wMOF and previous two schemes sliding window applied on NAF, and *w*NAF. These properties are size of precomputed table (i.e. $\#\{|d| : d \in D^*\}$, and the nonzero density. Table 3.3 shows the comparison of these characterizing properties, where SW is an abbreviation for sliding window.

Scheme	Table Size	1/ nonzero density
SW+ NAF [33]	2^{w-2}	$w + \frac{4}{3} - \frac{(-1)^w}{3.2^{w-2}}$
wNAF [17][18]	$\frac{1}{3}(2^w + (-1)^{w+1})$	$w+1$
wMOF [27]	2^{w-2}	$w+1$

Table 3.3 General comparison of table size and non-zero density

3.2.4.3 Exponentiation with *w*MOF, w >2

All algorithms for generating *w*NAF need carry-overs, as result the recoding is restricted to be done right-to-left. In the context of memory constraint devices, a small digit set D is even more valuable, because fewer precomputed elements have to be stored. Although none of the preceding methods is a left-to-right scheme, each one requires additional memory $O(n)$ to store the recoded string before starting the left-to-right evaluation of the exponent product. The

advantages of exponentiation with *w*MOF, the digit set of *w*MOF is the same as for *w*NAF, and turns out as a complete left-to-right scheme. The evaluation stage can be performed left-to-right, and the recoding into *w*MOF proceed left-to-right, this due to no additional memory required for performing the exponentiation, since algorithm 13 requires only O(*w*) bits memory for generating *w*MOF[27].

Okeya and Takagi [27] constructed an algorithm to compute table look-up for any *w* in an efficient way, and less memory usage. This table enables to merge the evaluation stage that can be performed left-to-right, and the recoding into *w*MOF.

1. Computation Table look-up

The table computation algorithm 3.11 has to compute γ and ξ which fit the equation $c = \gamma * 2^{\xi}$, and the converted one *w*MOF δ is obtained from γ and ξ as:

$$\delta = (\underbrace{0,2,3,0}_{w-\xi-1}), \gamma, \underbrace{0,2,3,0}_{\xi} \tag{3.8}$$

Algorithm 3.11 Table Computation with Width *w* [27]

INPUT: width w.

OUTPUT: arrays $\gamma_{0...tw}$ and $\xi_{0...tw}$ where $tw = 2^{w-1}$.

1. For $k \leftarrow 2^{w-1}$ to $3 * 2^{w-1} - 1$ do the following
   1.1 $c \leftarrow (k \,\&\, (2^{w-1})) - (k >> 1)$
   1.2 $\xi_{k-2}{}^{w-1} \leftarrow 0$
   1.3. While $(c \,\&\, 1) = 0$ do the following
      1.3.1 $\xi_{k-2}{}^{w-1} \leftarrow \xi_{k-2}{}^{w-1} + 1$
      1.3.2 $c \leftarrow c >> 1$
   1.4. $\gamma_{k-2}{}^{w-1} \leftarrow c$
2. return $\gamma_{0...tw}$ and $\xi_{0...tw}$

We can observe from algorithm that ξ is in {0, 1, ...,*w* - 1} which are *w* different values,$\lceil log_2 w \rceil$ bits are required to store ξ and each element of γ is in {±1, ±3, ..., ±(2^{w-1} - 1)} which has the cardinality of 2^{w-2} and requires w - 2 pre-commuted points.

2. Computation of Precomputed points

Further, the precomputations (2^{w-2}- 1) of elliptic pints are required, since wMOF is signed binary representations. Those points are all points of the form γP, where γ ∈ signed digit set D = {0,±1, ±3, ..., ±(2^{w-1} - 1)}, and *P* is an elliptic point .

3. On the Fly Multiplication for w > 2

Finally the table look-up created in algorithm 3.12 computed and precomputed points are used to merge the recoding and evaluation stages for any *w*.

Algorithm 3.12 Exponentiation with *w*MOF [27]

INPUT a non-zero t-bit binary string k, a point P and the multiple of the point P, $\gamma_{0...tw}$ and $\xi_{0...tw}$, the precomputed table look-up .

OUTPUT exponentiation kP.

1. i ← t
2. Q ← *O*
3. While i ≥ 1 do the following
   - 3.1. if (k_i XOR k_{i-1}) = 0, then do the following
     - 3.1.1. Q ←ECDBL(Q)
     - 3.1.2. i ← i - 1
   - 3.2. else do the following
     - 3.2.1. index ← ((k >> (i - w)) & (2^{w+1} - 1)) - 2^{w-1}
     - 3.2.2. For j = 1 to *w* - ξ_{index} do the following
       1. Q ← ECDBL(Q)
       2. i ← i - 1

3.2.3. $Q \leftarrow ECADD(Q, \gamma_{index}P)$

3.2.4. For $j = 1$ to ξ_{index} do the following

1. If $i \geq 0$ then $Q \leftarrow ECDBL(Q)$
2. $i \leftarrow i - 1$

4. If $i = 0$ do the following

4.1. $Q \leftarrow ECDBL(Q)$

4.2. If $k_0 = 1$ then $Q \leftarrow ECADD(Q,-P)$

5. return Q

CHAPTER 4

4 Contribution of This Thesis

Chapter 3 presents typical methods for exponentiation, where addition of two points and doubling of two points are performed repeatedly. These methods can speed up exponentiation by reducing addition, but the doublings are quite costly.

This chapter introduces new formula works with *w*MOF for speed up exponentiation on an elliptic over $\mathbf{F}_p$. This formula can increase the speed of doubling by trading inversion for multiplication. In addition, we show the actual performance of the newly introduced algorithm and how this formula can improve *w*MOF method.

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

On method to increase the speed of doublings is direct computation of several doublings, which can compute 2^nP directly from $P \in E(\mathbf{F}_q)$, without computing the intermediate points $2P, 2^2P, \ldots, 2^{n-1}$.[28].

Guajardo and Paar[11] 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[28] proposed formulae for computing 2^nP directly ($\forall n \geq 1$) on $E(\mathbf{F}_p)$ in terms of affine coordinates.

These formulas require only one inversion for computing 2^nP instead of n inversions in regular add-double method. Therefore direct computation of several

doublings may be effective in elliptic curve exponentiation because modular inversion is more expensive than multiplication.

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

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

As an example Let $n_1 = 2$, $n_2 = 1$, let $P_1 = (x_1, y_1)$, 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 computed as the following

1) Computing $4P_1$ as in [28]

Let

$$C_0 = y_1$$

$$A_0 = x_1$$

$$B_0 = 3x_1^2 + a$$

$$A_1 = B_0^2 - 8A_0C_0^2$$

$$C_1 = -8C_0^4 - B_0(A_1 - 4A_0C_0^2)$$

$$B_1 = 3A_1^2 + 16aC_0^4$$

$$A_2 = B_1^2 - 8A_1C_1^2$$

$$C_2 = -8C_1^4 - B_1(A_2 - 4A_1C_1^2)$$

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

$$x_4 = \frac{A_2}{(4C_0C_1)^2} \tag{4.1}$$

$$y_4 = \frac{C_2}{(4C_0C_1)^3} \tag{4.2}$$

2) Computing $(4P_1+Q)$

Assume $4P_1 = (x_4, y_4) \neq -Q$. Recall from section 2.3, the point addition $P_1' = (x_1', y_1') = (4P_1+Q)$ in term of affine coordinates, can be computed as follows:

$$x_1' = \lambda^2 - x - x_4 \tag{4.3}$$

$$y_1' = \lambda (x - x_1') - y \tag{4.4}$$

$$\lambda = \frac{(y_4 - y)}{(x_4 - x)} \tag{4.5}$$

Substituting x_4, y_4 by equations (4.1) and (4.2) respectively into the expression for λ we readily find

$$\lambda = \frac{(\frac{C_2}{(4C_0C_1)^3} - y)}{(\frac{A_2}{(4C_0C_1)^2} - x)} \tag{4.6}$$

After simplification equation (4.6) we get

$$\lambda = \frac{C_2 - (4C_0C_1)^3 y}{(4C_0C_1)(A_2 - (4C_0C_1)^2 x)} \tag{4.7}$$

Now let

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

$$\lambda = \frac{T}{(4C_0C_1)S} \tag{4.8}$$

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

$$x'_1 = \frac{T^2}{(4C_0C_1)^2S^2} \text{ - x- } \frac{A_2}{(4C_0C_1)^2} \tag{4.9}$$

After simplification equation (4.9) we get:

$$x'_1 = \frac{T^2 - S^2(A_2 + (4C_0C_1)^2 x}{(4C_0C_1)^2S^2} \tag{4.10}$$

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

$$x'_1 = \frac{T^2 - MS^2}{(4C_0C_1)^2S^2} \tag{4.11}$$

Let $A'_0 = T^2 - MS^2,$ we get:

$$x'_1 = \frac{A'_0}{(4C_0C_1)^2S^2} \tag{4.12}$$

Substituting λ, and x'_1 from equation (4.12) into the expression for $y'_1,$ we find

$$y'_1 = \frac{T}{(4C_0C_1)S}\left(x - \frac{A'_0}{(4C_0C_1)^2S^2}\right) - y \tag{4.13}$$

After simplification we get:

$$y'_1 = \frac{-(4C_0C_1)^3 yS^3 - T(A'_0 - (4C_0C_1)^2 xS^2)}{(4C_0C_1)^3S^3} \tag{4.14}$$

Let $C'_0 = -(4C_0C_1)^3 yS^3 - T(A'_0 - (4C_0C_1)^2 xS^2),$ we get:

$$y'_1 = \frac{C'_0}{(4C_0C_1)^3S^3} \tag{4.15}$$

3) Computing $2(4P_1+Q) = 2P'_1$

Recall from section 2.2, the point doubling $2P'_1 = P'_2 = (x'_2, y'_2)$ in term of affine coordinates, can be computed as follows:

$$x'_2 = \lambda^2 - 2x'_1 \tag{4.16}$$

$$y'_2 = \lambda(x'_1 - x'_2) - y'_1 \tag{4.17}$$

$$\lambda = \frac{3x_1'^2 + a}{2y_1'} \tag{4.18}$$

Substituting x_1', y_1' by equations (4.12) and (4.15) respectively into the expression for λ we readily find

$$\lambda = \frac{3\left(\frac{A_0'}{(4C_0C_1)^2S^2}\right)^2 + a}{2\left(\frac{C_0'}{(4C_0C_1)^3S^3}\right)} \tag{4.19}$$

After simplification we get:

$$\lambda = \frac{3A_0'^2 + a(4C_0C_1)^4S^4}{2C_0'(4C_0C_1)S} \tag{4.20}$$

Now let $B_0' = 3A_0'^2 + a(4C_0C_1)^4S^4$, we get:

$$\lambda = \frac{B_0'}{2C_0'(4C_0C_1)S} \tag{4.21}$$

Substituting λ, and x_1' into the expression for x_2', we find

$$x_2' = \frac{B_0'^2}{(2C_0')^2(4C_0C_1)^2S^2} - 2\left(\frac{A_0'}{(4C_0C_1)^2S^2}\right) \tag{4.22}$$

After simplification we get:

$$x_2' = \frac{B_0'^2 - 8A_0'C_0'^2}{(2C_0')^2(4C_0C_1)^2S^2} \tag{4.23}$$

Let $A_1' = B_0'^2 - 8A_0'C_0'^2$, we get:

$$x_2' = \frac{A_1'}{(2C_0')^2(4C_0C_1)^2S^2} \tag{4.24}$$

Substituting λ, y_1', x_1' and x_2' into the expression for y_2', we find

$$y_2' = \frac{B_0'}{2C_0'(4C_0C_1)S}\left(\frac{A_0'}{(4C_0C_1)^2S^2}\right) - \left(\frac{A_1'}{(2C_0')^2(4C_0C_1)^2S^2}\right) - \frac{C_0'}{(4C_0C_1)^3S^3} \tag{4.25}$$

After simplification we get:

$$y'_2 = \frac{-8C'^4_0 - B'_0(A'_1 - 4A'_0C'^2_0)}{(2C'_0)^3(4C_0C_1)^3S^3} \tag{4.26}$$

Let $C' = -8C'^4_0 - B'_0(A'_1 - 4A'_0C'^2_0)$, we get finally:

$$y'_2 = \frac{C'_1}{(2C'_0)^3(4C_0C_1)^3S^3} \tag{4.27}$$

Algorithm 4.1 shows the general formulae that allow direct computing $2^{n_2}(2^{n_1}P + Q)$ for $n_1 \geq 1$.

Algorithm 4.1 Direct Computation of $2^{n_2}(2^{n_1}P + Q)$ in affine coordinate, where $n_1 \geq 1$, and P, Q $\in$ E($\mathbf{F}_p$) .

INPUT: $P_1 = (x_1, y_1)$, $Q = (x, y) \in E(\mathbf{F}_p)$

OUTPUT: $P'_{2^4} = 2^4P' = 2^4(2P_1 + Q) = (x'_{2^4}, y'_{2^4}) \in E(\mathbf{F}_p)$

1. Compute A_0 and C_0 and B_0

   $C_0 = y_1$

   $A_0 = x_1$

   $B_0 = 3x_1^2 + a$

2. For i from 1 to n_1 Compute A_i, C_i, for i from 1 to $n_1 - 1$ Compute B_i

   $A_i = B_{i-1}^2 - 8A_{i-1}C_{i-1}^2$

   $C_i = -8C_{i-1}^4 - B_{i-1}(A_i - 4A_{i-1}C_{i-1}^2)$

   $B_i = 3A_i^2 + 16^i a(\prod_{j=0}^{i-1} C_j)^4$

3. Compute the N, V, W, Z then A'_0, C'_0

   $N = A_2 - (2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2 x$

   $V = A_2 + (2^{n_1}\prod_{i=0}^{n_1-1} C_i)^2 x$

$$W=C_2-(2^{n_1}\prod_{i=0}^{n_1-1}C_i)^3 y$$

$$Z=(2^{k_1}\prod_{i=0}^{k_1-1}C_i)N$$

$$A'_0=W^2-VN^2$$

$$C'_0=-Z^3 y-W(A'_0-Z^2 x)$$

4. *if* $(n_2>0)$

Compute B'_0

$$B'_0=3A'^2_0+aZ^4$$

For i from 1 to n_2 Compute A'_i, C'_i , for i from 1 to n_2-1 Compute B'_i

$$A'_i=B'^2_{i-1}-8A'_iC'^2_i$$

$$C'_i=-8C'^4_{i-1}-B'_{i-1}(A'_i-4A'_iC'^2_i)$$

$$B'_i=3A'^2_{i-1}+16^i aZ^4(\prod_{j=0}^{i-1}C'_j)^4$$

Compute Z

$$Z=Z(2^{n_2}\prod_{i=0}^{n_2-1}C'_i)$$

5. Compute $x'_{2^{k_2}}, y'_{2^{k_2}}$

$$x'_{2^{n_2}}=\frac{A'_{n_2}}{Z^2}$$

$$y'_{2^{n_2}}=\frac{C'_{n_2}}{Z^3}$$

Theorem 4.1 describes the computational complexity of this formula.

Theorem 4.*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 [28, Algorithm1] 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 $(2^{n_1} \prod_{i=0}^{n_1-1} C_i)^2 x$. Then we obtain N, and V. Next, we perform two multiplications, one multiplication to compute $(2^{n_1} \prod_{i=0}^{n_1-1} C_i)^2 y$ and other to compute $(2^{n_1} \prod_{i=0}^{n_1-1} C_i)(2^{n_1} \prod_{i=0}^{n_1-1} C_i)^2 y = (2^{n_1} \prod_{i=0}^{n_1-1} C_i)^3 y$. 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'. Forth, 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 1 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 4.1 involve M + S and the complexity of step 2 involves $(2M + 3S)n_2 + (M+S)(n_2 - 1)$ as step 2.

In step 3 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 sub-step 3 involves $n_2 M$. Hence, the complexity of step 4 involves $4n_2 M + 4n_2 S$.

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 $C_{n_2}' T^2 T$. Then we obtain $y'_{2^{n_2}}$. The complexity of step 5 involves $3M + S + I$. Therfore the complexity of above computations involve $(4(n+2) +2) M, (4(n+1) + 2)S$, *where* $n = n_1 + n_2$. □

4.1.1 The Break-Even Point

For application in practice it is highly relevant to compare the complexity of our newly derived formulae for direct computing of *n* doublings separated with one addition and individual *n* doublings. 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 [11], 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. Table 4.1 shows the number of squarings *S*, multiplications *M*, and inversions *I* in $\mathbf{F}_p$.

Calculation with n	Method	Complexity			Break-Even Point
		S	*M*	*I*	
4	DECDBL(*4*)	22	26	1	$6.6\,M < I$
	4 doublings + 1 addition	10	9	5	
5	DECDBL(*5*)	26	30	1	$6\,M < I$
	5 doublings + 1 addition	12	11	6	
w	DECDBL(*w*)	4*w*+6	4*w*+10	1	$\frac{(3.6\,w+12)}{w}M$
	w doublings + 1 addition	2*w*+1	2*w*+2	w+1	

Table 4.1 Complexity comparison: Individual doublings and one addition vs. direct computation of several doublings with one addition.

In general let $n = n_1 + n_2$, and 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:

$$\text{Cost(separate } n \text{ ECDBL}_A + \text{ECADD}_A) > \text{Cost(DECDBL}(n)\text{)}$$

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

$$(2n\,M + 2n\,S + n\,I + 2M + S + I) > (4(n+2)M + 4(n+1)S + 2M + 2S + I)$$

We define r = *I/M* (the ratio of speed between a multiplication and inversion), and assume that one squaring has complexity $S = 0.8\,M$[28]. We also assume that the cost of field addition and multiplication by small constants can be ignored. One can rewrite the above expressions as:

$$n \text{ r } M > (2nM + 8M + 1.6n\,M + 4M)$$

Solving for r in terms of M one obtains:

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

As we can see from Table 4.1, if a field inversion has complexity I > 7.6 M, direct computation of 3 doublings with one addition may be more efficient than 3 separate doubling and one adding.

4.2 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 $n_1 \geq 1$, and P,Q $\in$ E($\mathbf{F}_p$), we can improve algorithm 3.12 for elliptic curve exponentiation with wMOF by change step 3.2 of algorithm 3.12 with a new step that compute $2^{n_2}(2^{n_1}P + Q)$ directly as in the following algorithm.

Algorithm 4.2 Exponentiation with wMOF Using Direct Computation of $2^{n_2}(2^{n_1}P + Q)$

INPUT a non-zero t-bit binary string k, a point P and the multiple of the point P, $\gamma_{0...tw}$ and $\xi_{0...tw}$, the precomputed table look-up .

OUTPUT exponentiation kP.

1. i ← t
2. Q ← O
3. While i ≥ 1 do the following
   - 3.1. if (k_i XOR k_{i-1}) = 0, then do the following
     - 3.1.1. Q ←ECDBL(Q)
     - 3.1.2. i ← i - 1
   - 3.2. else do the following
     - 3.2.1. index ← ((k >> (i - w)) & (2^{w+1} - 1)) - 2^{w-1}

3.2.2. if (i < w) $Q \leftarrow 2^{i-(w-\xi index)+1}(2^{w-\xi index} Q + \gamma_{index} P)$

3.2.3 else if (i ≥ w) $Q \leftarrow 2^{\xi index}(2^{w-\xi index} Q + \gamma_{index} P)$

3.2.4. i ← i - w

4. If i = 0 do the following

4.1. Q ← ECDBL(Q)

4.2. If $k_0 = 1$ then Q ← ECADD(Q,-P)

5. return Q

In algorithm 4.2, for each window width w of wMOF, Step 3.2.1 performs direct computation of $2^{i-(w-\xi index)+1}(2^{w-\xi index} Q + \gamma_{index} P)$ if (i < w) otherwise Step 3.2.2 performs direct computations of $2^{\xi index}(2^{w-\xi index} Q + \gamma_{index} P)$ if (i ≥ w), where $\xi_{index} = 0,1,\ldots w\text{-}1$, $\gamma_{index} P = \{\pm 1, \pm 3, \ldots, \pm(2^{w-1} - 1)\}$.

4.2.1 Complexity Analysis of the wMOF Method

In this subsection, we perform an analysis of wMOF method when it used in conjunction with the $2^{n_2}(2^{n_1} P + Q)$ formula. In addition, we compare the complexity of wMOF method, with and without formula. 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 4.2 describes the complexity of algorithm 3.12 for computing exponentiation with wMOF.

Theorem 4.2 *In terms of affine coordinate, Let* $P \in E(\mathbf{F}_p)$*, t-digits exponent in* wMOF*, then the complexity of algorithm 3.12 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 3.12 performs an ECADD operation each time the current digit δ is non-zero, recall from theorem 3.12 that the average non-zero density of *w*MOF is asymptotically $\frac{1}{w+1}$ also, one ECDBL operation is performed in each iteration (where i ≥ 0) to double the intermediate result. Then on average, algorithm 3.12 for computing exponentiation with wMOF requires

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

Recall from table 2.2, the computational costs for doublings and additions operations in affine coordinate. Then we can rewrite previous expression as:

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

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

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

Now Theorem 4.3 describes the complexity algorithm 4.2 for computing exponentiation with wMOF by using $2^{n_2}(2^{n_1}P + Q)$.

Theorem 4.3 *In terms of affine coordinate, Let* $P \in E(\mathbf{F}_p)$*, and t-digits exponent in w*MOF*, then the complexity of algorithm 4.2 for computing kP requires on*

average $\frac{4(w+3)t}{w+1}\mathrm{M}+\frac{4(w+2)t}{w+1}\mathrm{S}+\frac{2t}{w+1}\mathrm{I}$, *where M, S and I denote multiplication, squaring and inversion respectively*.

Proof From theorem 3.12, for t-digits exponent k in its *w*MOF, if $t \to \infty$ the average non-zero density of *w*MOF is asymptotically $\frac{1}{w+1}$ and *w*MOF 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 $0 \le i \le w - 1$;

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 *w*MOF – 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 4.2 performs $\frac{1}{w+1}t$ blocks b_1 and step 3.2 performs $\frac{1}{w+1}t$ block b_2 then algorithm 4.2 for computing *kP* requires on average

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

Recall from Table 2.2, the computational costs for doublings and additions operations in affine coordinate. Then we can rewrite previous expression as:

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

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

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

Relative Improvement

Let us denote the times it would take to perform exponentiation by using algorithms 3.12, and 4.2 by symbols $T_{\text{Regular method}}$, $T_{\text{Formula method}}$ respectively. According to theorems 4.2, and 4.3, we can derive expressions for the time it would take to perform a whole exponentiation with *w*MOF as:

$$T_{\text{Regular method}} = \frac{(2w+4)n}{w+1}\mathrm{M} + \frac{(2w+3)n}{w+1}\mathrm{S} + \frac{(w+2)n}{w+1}\mathrm{I} \qquad (4.28)$$

$$T_{\text{Formula method}} = \frac{4(w+3)n}{w+1}\mathrm{M} + \frac{4(w+2)n}{w+1}\mathrm{S} + \frac{2n}{w+1}\mathrm{I} \qquad (4.29)$$

Notice that from equations 4.28, and 4.29, one can readily derive the relative improvement by defining r = I/M (the ratio of speed between a multiplication and inversion) as:

$$\text{Relative Improvement} = \frac{T_{\text{Regular method}} - T_{\text{Formula method}}}{T_{\text{Regular method}}} \qquad (4.30)$$

By using (4.28) and (4.29)

$$\text{Relative Improvement} = \frac{wI - [(2w+8)M + (2w+5)S]}{(w+2)I + [(2w+4)M + (2w+3)S]} \qquad (4.31)$$

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

$$\text{Relative Improvement is} = \frac{4(r) - 29}{6(r) + 23} \qquad (4.32)$$

$$\text{Relative Improvement is} = \frac{4(12.6) - 29}{6(12.6) + 23} \cdot 100 = 21.7\ \% \qquad (4.33)$$

4.3 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 have several choices, these include selection of elliptic curve domain parameters, platforms [6].

4.3.1 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 [24]. 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 parameters of these curves are in Appendix B.

The ECC is implemented on a Pentium 4 personal computer (PC) with 2 GHz processor and 512 MB of RAM. Programs were written in Java language for multi-precision integer operations, and are ran under Windows XP.

We used jBorZoi Library[4] in this implementation. jBorZoi is a Java Elliptic Curve Cryptography which implements cryptographic algorithms using elliptic curves defined over binary finite fields. We extended jBorZoi Library to implement cryptographic algorithms using elliptic curves defined over prime finite fields. Complete code listings are provided in Appendix B.

4.3.2 Timings analysis of *w*MOF 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 4.2.

Curves	Average Timing (μsec) for M	Average Timing (μsec) for S	Average Timing (μsec) 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

Table 4.2 The ratio of speed between a multiplication and inversion in prime filed F_p

4.3.2.1 Optimal Window Size

To show the actual improvement of *w*MOF 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 (4.1- 4.4) illustrate the relation among the window size w, the speed of the evaluation and pre-computed processes. We can noticed 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 thesis will be processed for *w* = 4 in 160-bits input and *w*= 5 for 192, 224, or 256-bits.

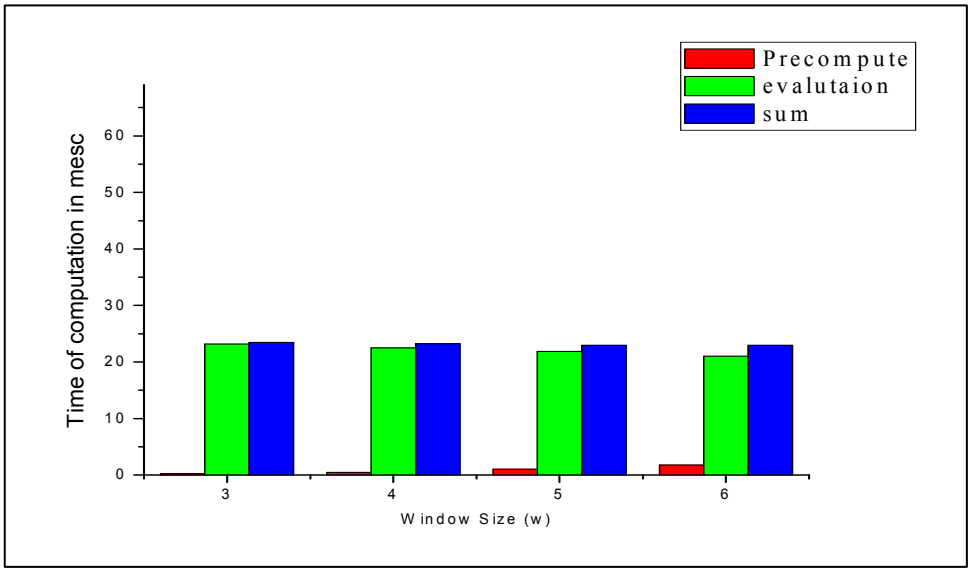

Figure 4.1 Pre-compute and evaluation with 160-bits input

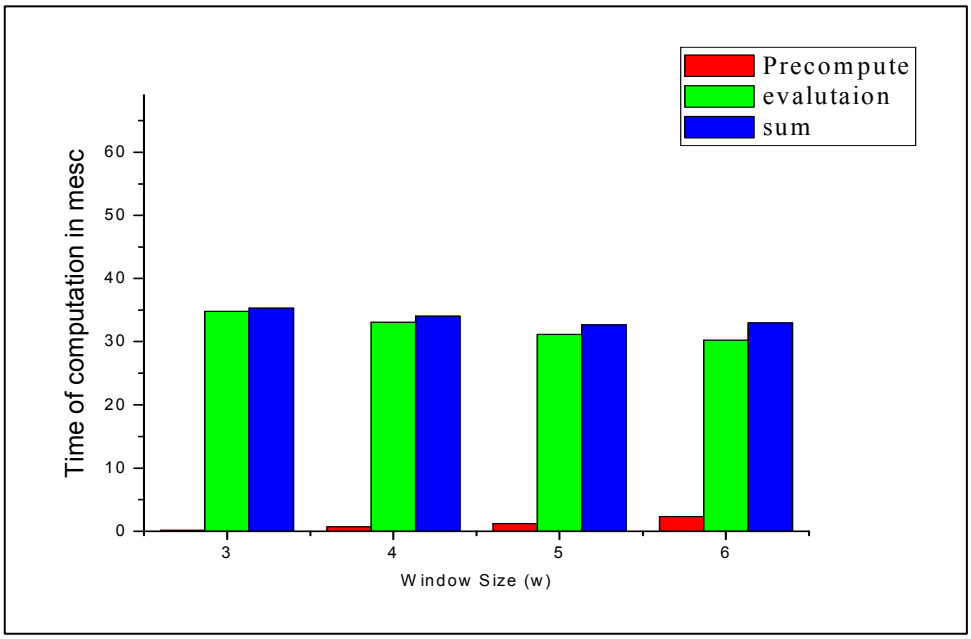

Figure 4.2 Pre-compute and evaluation with 192-bits input

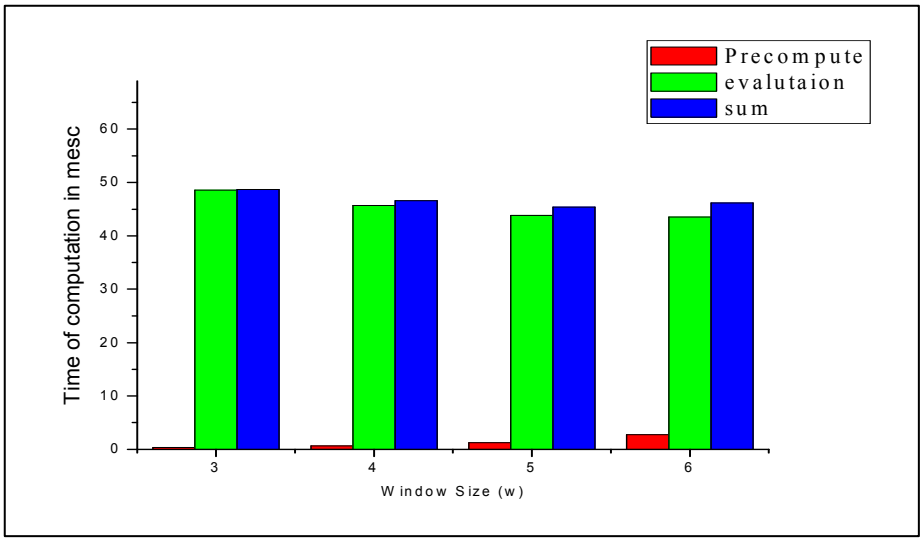

Figure 4.3 Pre-compute and evaluation with 224-bits input

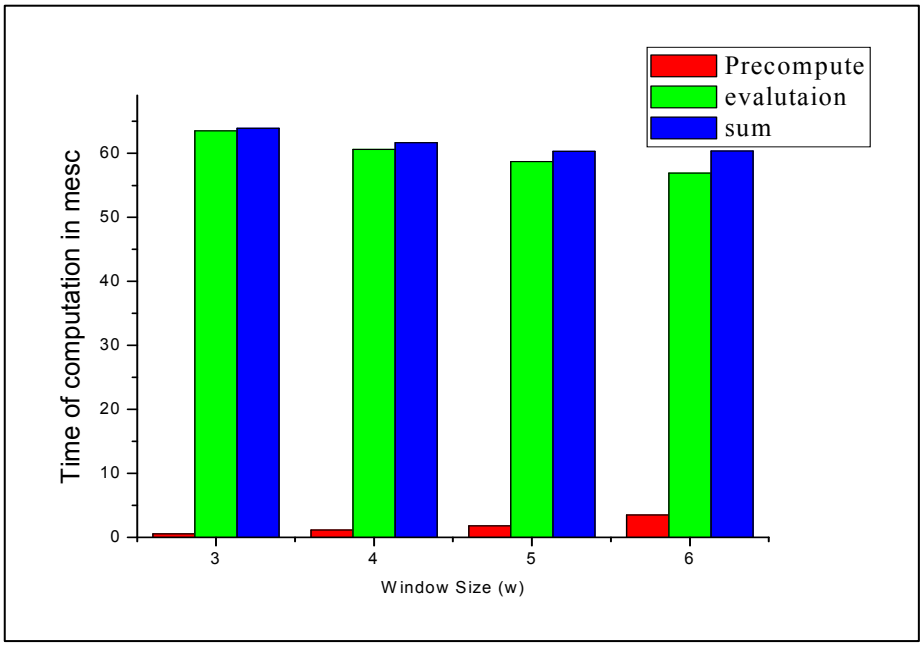

Figure 4.4 Pre-compute and evaluation with 256-bits input

4.3.2.2 The performance of *w*MOF method

Table 4.3 shows how the wMOF can outperform the binary method (Add-Double) by taking the optimal window size *w*.

Curves	Time in mesec		# Pre- computed points	
	Add- Double	wMOF	Add- Double	wMOF
P 160	27.1	22.6	0	3
P 192	41.5	33.15	0	7
P 224	54.8	46.2	0	7
P 256	73.3	60.3	0	7

Table 4.3 Comparison of add-double method vs. wMOF method to perform an exponentiation

4.3.2.3 The performance of improved *w*MOF method

Using Table 4.2, we can readily predict that the timings for performing a exponentiation with and without the formulae presented in Algorithm 4.1. In addition, using the complexity shown in theorems (4.2, 4.3) and the timings shown in Table 4.2 we can make estimates as to how long an exponentiation with wMOF will take using both doublings with formulae and individual doublings.

Curves	Method	Predicted Timing	Measured Timing	% Improvement	
				Predicted	Measured
P 160	wMOF with formula (w = 4)	17.4	18.3	21.62	21.7
	wMOF (w = 4)	22.2	23.4		
P 192	wMOF with formula (w = 5)	23.8	24.3	25.62	25.7
	wMOF (w = 5)	32	32.6		
P 224	wMOF with formula (w = 5)	31.7	33.9	24.52	24.6
	wMOF (w = 5)	42	45		
P 256	wMOF with formula (w = 5)	43.8	47.4	23.5	23.3
	wMOF (w = 5)	57.3	61.8		

Table 4.4 Average time comparison required to perform an exponentiation without pre-computations stage of a random point in mesc (Pentium IV 2.0 GHz).

CHAPTER 5

5 Conclusion

This thesis presented several methods which can be taken to efficiently implement cryptosystems on smart cards.

As explained in Chapter 1, one of the most critical issues concerning cryptosystems is the security of the secret key which is used for signing and decrypting messages. Due to their tamper resistance and mobility, smart cards are a good choice to serve as host for the secret keys and the cryptosystems. However, since the computational power and the available memory on smart cards are very limited, efficient implementations are needed.

The first measure to reduce the memory and computational power required is to use cryptosystems that are based on the additive group of points on an elliptic curve. The main advantage of elliptic curves over commonly used groups is, that the same level of security can be achieved with much smaller key sizes, i.e. 160-bit instead of 1024-bit.

As it turned out, exponentiation is the most basic operation used in elliptic curve cryptosystems. We construct 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 have showed in what way the formula for computing $2^{k_2}(2^{k_1}P+Q)$ can improve the speed of the

exponentiation with wMOF. 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. 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.

Appendix A Mathematical Background

A.1 Basic Algebra

We provide here the essential algebraic terminologies and concepts required for the understanding of the studies on elliptic curves.

Definition A.1 A nonempty set of elements G is said to form a *group* $(G, \bullet)$ if in G there is defined an operation, called the product and denoted by $\bullet$, such that

1. $a, b \in G$ implies that $a \bullet b \in G$ (closure).

2. $a, b, c \in G$ implies that $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ (associative law).

3. There exists an element $e \in G$ such that $a \bullet e = e \bullet a = a$ for all $a \in G$ (the existence of an identity element in G).

4. For every $a \in G$ there exists an element $a\text{-}1 \in G$ such that $a \bullet a\text{-}1 = a\text{-}1 \bullet a = e$ (the existence of inverses in G).

Definition A.2 A group G is said to be *abelian* (*commutative*) if $a \bullet b = b \bullet a$ for all $a, b \in G$.

Definition A.3 A *field* is a nonempty set of elements **F** with two operations, *addition* "+" and *multiplication* "×", such that

1. (**F** , +) is an abelian additive group.

2. (**F**\{**0**}, ×) is an abelian multiplicative group, where **0** denotes the additive identity element.

3. The distributive laws hold in **F**.

From now on, **F** will always denote a field and $\mathbf{F}^*$ will denote the group of nonzero elements of **F**, unless otherwise stated.

Definition A.4 *The characteristic of **F**, denoted by char(**F**), is defined to be the smallest positive integer p such that pa = 0 for all a* $\in$ ***F**. If such an integer does not exist, char(**F**) is zero.*

Definition A.5 *A finite field* $\mathbf{F}_q$ *is a field that has a finite number q of elements. In particular, for a prime p,* $\mathbf{F}_p$ *is the field of equivalence classes of integers modulo p and thus has a finite number p of elements.*

Definition A.6 *A field **K** is said to be an extension of **F** if **K** contains **F**.*

Definition A.7 *The ring of polynomials* **F** [x] *in x over* **F** *is the set of all formal expressions* $f(x) = a_0 + a_1x + \cdots + a_nx^n \geq 0$, $a_i \in \mathbf{F}$ *for all i = 0, 1, …, n.*

Definition A.8 *Let* $f(x) \in \mathbf{F}[x]$, *If* $f(x) \neq 0$ *and* $a_n \neq 0$, *then the degree of f(x), written as deg f(x), is n.*

Definition A.9 **F** *is said to be algebraically closed if for every* $f(x) \in \mathbf{F}[x]$ *of deg* $f(x) \geq 1$, *f(x) has a root in* **F**.

Definition A.10 *Let* **F** *be a field and let V be an additive abelian group. V is called a vector space over* **F** *if an operation* $\mathbf{F} \times V \square V$ *is defined so that the following conditions are satisfied*:

1. $a(u + v) = au + av$

2. $(a + b)u = au + bu$

3. $a(bu) = (a \cdot b)u$

4. $1u = u$

The elements of V are called vectors and the elements of **F** are called scalars.

Definition A.11 Let V be a vector space over a field **F** and let $v_1, v_2, \ldots, v_m \in V$. Any vector in V of the form

$$c_1v_1 + c_2v_2 + \cdots + c_mv_m$$

where $c_i \in \mathbf{F}$ ($i = 1,\ldots,m$) is a linear combination of $v_1, v_2,\ldots, v_m$. The set of all such linear combinations is called the linear span of $v_1, v_2,\ldots, v_m$ and it is denoted by span($v_1, v_2,\ldots, v_m$). The vectors $v_1, v_2,\ldots, v_n$ are said to span or generate V if V = span($v_1, v_2,\ldots, v_n$).

Definition A.12 Let V be a vector space over a field **F**. The vectors $v_1, v_2,\ldots, v_m \in$ V are said to be linearly independent over **F** if there are no scalars $c_1, c_2,\ldots, c_m \in \mathbf{F}$ (not all 0) that satisfy

$$c_1v_1 + c_2v_2 + \cdots + c_mv_m = 0$$

Definition A.13 A set S = $\{u_1, u_2,\ldots, u_n\}$ of vectors is a basis of V if and only if $u_1, u_2,\ldots, u_n$ are linearly independent and they span V. If S is a basis of V, then every element of V is uniquely represented as a linear combination of the elements of S. If a vector space V has a basis of a finite number of vectors, then any other basis of V will have the same number of elements. This number is called the dimension of V over **F**.

A.2 Projective Space

Definition A.10 *The affine plane* $\mathbf{A}^2(\mathbf{F})$ *over* **F** *is* the usual plane, $\mathbf{A}^2(\mathbf{F}) = \{(x, y) | x, y \in \mathbf{F}\}$.

Definition A.11 *Define an equivalence relation on the triples over* **F***, not all components zero, as follows*: *(X, Y, Z)* : (X', Y', Z') *if and only* $(X', Y', Z') = \lambda(X, Y, Z)$ *for some* λ *in* $\mathbf{F}^*$. *Then each equivalence class (X, Y, Z) is called a projective*

point and the numbers X, Y, Z are called the homogeneous coordinates of that point. For instance, $(\frac{1}{4}, \frac{1}{2}, \frac{2}{3})$ *is equivalent to (3, 6, 8) (use* $\lambda = 12$*).*

The set of equivalence classes with respect to : is called 2-dimensional projective space over **F** and is denoted $\mathbf{P}^2$. The equivalence class of (*X, Y, Z*) in $\mathbf{P}^2$ is typically written [X, Y, Z] to avoid confusion with affine space.

Definition A.12 *The projective plane* $\mathbf{P}^2(\mathbf{F})$ *over* **F** *is the set of all projective points.*

$$\mathbf{P}^2(\mathbf{F})=\{ [X, Y, Z] \mid X, Y, Z \text{ not all zero}\}$$

For each projective point has $Z \neq 0$, a typical element [X, Y, Z] is equivalent to [x, y, 1], where x = X/Z, y =Y/Z. The set of these points is a copy of $\mathbf{A}^2(\mathbf{F})$.

For each projective point has Z = 0, a typical element like [X, Y, 0]. Note that either X or Y is nonzero if $X \neq 0$, then the [X, Y, 0] is equivalent to $[1, \frac{Y}{X}, 0]$ which is essentially a copy of $\mathbf{A}^1(\mathbf{F})$. If X = 0, then the typical point has the form [0, Y, 0], which is equivalent to [0, 1, 0] since Y is nonzero. Thus the set of these points is union of $\mathbf{A}^1(\mathbf{F})$ and the point [0, 1, 0], which is essentially a copy of $\mathbf{P}^1(\mathbf{F})$. This set is often referred to as the “line at infinity".

Therefore the projective plane $\mathbf{P}^2(\mathbf{F})$ can be thought of as a disjoint union of the affine plane $\mathbf{A}^2(\mathbf{F})$ with the line at infinity.

Appendix B

B.1 Recommended NIST Elliptic Curves over Prime Fields

The following parameters are given for each elliptic curve:

P The order of the prime field $\mathbf{F}_p$.

a,b The coefficients of the elliptic curve $y^2 = x^3 + ax + b$ *where a, b* $\in \mathrm{F}_p$ and $(4a^3 + 27b^2) \neq 0$. The selection a = -3 was made for reasons of efficiency;

xG, yG The x and y and coordinates of the base point G.

n The (prime) order of G

h The co-factor.

P-160: $p = 2^{160} - 2933$, $a = -3$, $h = 1$,
b = 0x 1C97BEFC 54BD7A8B 65ACF89F 81D4D4AD C565FA45
xG = 0x 4A96B568 8EF57328 46646989 68C38BB9 13CBFC82
yG = 0x 23A62855 3168947D 59DCC912 04235137 7B6B24F2
n = 0x 0100000000000000 0001F4C8 F927AED3 CA752257
P-192: $p = 2^{192} - 264 - 1$, $a = -3$, $h = 1$,
b = 0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1
XG = 0x 188DA80E B03090F6 7CBF20EB 43A18800 F4FF0AFD 82FF1012
yG = 0x 07192B95 FFC8DA78 631011ED 6B24CDD5 73F977A1 1E794811
n = 0x FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831

P-224: $p = 2^{224} - 2^{96} + 1$, $a = -3$, $h = 1$,
b = 0x B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4
xG = 0x B70E0CBD 6BB4BF7F 321390B9 4A03C1D3 56C21122 343280D6

115C1D21 yG = 0x BD376388 B5F723FB 4C22DFE6 CD4375A0 5A074764 44D58199 85007E34 *n* = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D
P-256: $p = 2^{256} - 2224 + 2^{192} + 2^{96} - 1$, $a = -3$, $h = 1$, *b* = 0x 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B xG= 0x 6B17D1F2 E12C4247 F8BCE6E5 63A440F2 77037D81 2DEB33A0 F4A13945 D898C296 yG= 0x 4FE342E2 FE1A7F9B 8EE7EB4A 7C0F9E16 2BCE3357 6B315ECE CBB64068 37BF51F5 *n* = 0x FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551

B.2 Complete Java code

```
package com.dragongate_technologies.borZoi;
import java.math.*;
import java.util.Date;
public class ECurveFp
   extends ECurve {
 protected static final BigInteger ZERO = BigInteger.ZERO;
    /**
     *  <code>constant</code> 1
     */
    protected static final BigInteger ONE = BigInteger.ONE;
    /**
     *  <code>constant</code> 2
     */
    protected static final BigInteger TWO = new BigInteger("2");
    /**
```

```
   *  <code>constant</code> 3
   */
  protected static final BigInteger THREE = new BigInteger("3");
  /**
   *  <code>constant</code> 4
   */
  protected static final BigInteger FOUR = new BigInteger("4");
  /**
   *  <code>constant</code> 8
   */
  protected static final BigInteger EIGHT = new BigInteger("8");
  /**
   *  <code>constant</code> 12
   */
  protected static final BigInteger TWELVE = new BigInteger("12");
  /**
   *  <code>constant</code> 16
   */
  protected static final BigInteger SIXTEEN = new BigInteger("16");
 public ECurveFp(Fp a4, Fp a6) {
 this.a4 = (Fp) a4.clone();
 this.a6 = (Fp) a6.clone();
}

public ECPoint doubl(ECPoint P0) {
 BigInteger a, b, lambda, x0, y0, x1, y1, x2, y2;
 a = a4.val;
 b = a6.val;
 x0 = P0.x.val;
 y0 = P0.y.val;
 x1 = P0.x.val;
 y1 = P0.y.val;
 ECPointFp P2 = new ECPointFp();
 if ((P0.isZero())||(P0.y.isZero())) {
  return P0;
 }
 else {
   lambda = Fp.Fp_mul(x0, x0).multiply(BigInteger.valueOf(3));
   lambda = Fp.Fp_add(a, lambda);
   lambda = Fp.Fp_mul(lambda, Fp.Fp_inv(y0.multiply(BigInteger.valueOf(2))));
   x2 = Fp.Fp_add( (x0.negate()).multiply(BigInteger.valueOf(2)),
              Fp.Fp_mul(lambda, lambda));
   y2 = Fp.Fp_mul(Fp.Fp_add(x0, x2.negate()), lambda);
   y2 = Fp.Fp_add(y2, y0.negate());
   P2.x.val = x2;
   P2.y.val = y2;
```

```
  }
  return P2;
}
public ECPoint add(ECPoint P0, ECPoint P1) {
  BigInteger a, b, lambda, x0, y0, x1, y1, x2, y2;
  a = a4.val;
  b = a6.val;
  x0 = P0.x.val;
  y0 = P0.y.val;
  x1 = P1.x.val;
  y1 = P1.y.val;

  ECPointFp P2 = new ECPointFp();
  if (P0.isZero()) {
    return P1;
  }
  if (P1.isZero()) {
    return P0;
  }
  if (P0.x.compareTo(P1.x) != 0) {
    lambda =
        Fp.Fp_mul(
        Fp.Fp_add(y0, y1.negate()),
        Fp.Fp_inv(Fp.Fp_add(x0, x1.negate())));
    x2 = Fp.Fp_add(x0.negate(), Fp.Fp_mul(lambda, lambda));
    x2 = Fp.Fp_add(x2, x1.negate());
    y2 = Fp.Fp_mul(Fp.Fp_add(x0, x2.negate()), lambda);
    y2 = Fp.Fp_add(y2, y0.negate());
    P2.x.val = x2;
    P2.y.val = y2;
  }
  else if (P0.y.compareTo(P1.y) != 0) {
    return P2;
  }
  else if (P1.x.isZero()) {
    return P2;
  }
  else {
    return doubl(P0);
  }
  return P2;

}

  public ECPoint mul(BigInteger n, ECPoint P) {
  ECPoint Q;
```

```
  ECPoint S = new ECPointFp();
  BigInteger k;
  if (n.compareTo(BigInteger.valueOf(0)) == 0) {
    return new ECPointFp();
  }

  if (n.compareTo(BigInteger.valueOf(0)) < 0) {
    k = n.negate();
    Q = P.negate();
  }
  else {
    k = n;
    Q = P;
  }

  for (int j = k.bitLength() - 1; j >= 0; j--) {
    S = doubl(S);
    if (k.testBit(j))
       S = add(S, Q);
   }
  return S;
 }
 public ECPoint[] pre_ECpoints(ECPoint P) {
  int m;
  ECPoint[] G;
  m = Pre_Table.pow(2, Pre_Table.Win - 1);
  G = new ECPoint[m];
  G[1] = P;
  G[2] = doubl(P);
  for (int j = 1; j < m / 2; j++) {
     G[ (2 * j + 1)] = add( (G[ (2 * j - 1)]), (G[2]));
  }
  return G;
}
 public ECPoint Im_wMOF(BigInteger d, ECPoint P) {
  Date d1,d2;
  long x1,x2;
  ECPoint Q = new ECPointFp();
  ECPoint P1 =new ECPointFp();
  ECPoint[] G;
  ECPoint infinty, K;
  BigInteger tk;
  int m, index, i;
  String s;
  char n_bits[];
  m = Pre_Table.pow(2, Pre_Table.Win - 1);
```

```
    int tw = Pre_Table.pow(2, Pre_Table.Win);
    s = new String(" ");
    s = d.toString(2);
    n_bits = new char[s.length() + 1];
    s.getChars(0, s.length(), n_bits, 1);
    n_bits[0] = '0';
    infinty = new ECPointFp();
    Q = infinty;
    i = s.length();
    index = 0;
    int j;
    // precomputation  stage
    G = pre_ECpoints(P);
    while (i >= 1) {
      j = s.length() - i;
      if (n_bits[j] == n_bits[j + 1]) {
        Q = doubl(Q);
        i = i - 1;
      }
      else {
        index = (d.shiftRight(i - Pre_Table.Win)).intValue();
        index = (index & (2 * tw - 1)) - tw / 2;
        if (Pre_Table.gama[index] > 0)
             P1 =G[Pre_Table.gama[index]];
          else if (Pre_Table.gama[index] < 0)
            P1= G[- (Pre_Table.gama[index])].negate();
        if (i < Pre_Table.Win)
          Q =D_Mu(Q,P1,Pre_Table.Win - Pre_Table.zeta[index],i-
(Pre_Table.Win –
                        Pre_Table.zeta[index])+1 );
        else
          Q =D_Mu(Q,P1,Pre_Table.Win -
Pre_Table.zeta[index],Pre_Table.zeta[index] );
        i=i-(Pre_Table.Win);
      }
    }
  if (i == 0) {
     Q = doubl(Q);
     if (n_bits[s.length()] == '1') {
       Q = add(Q, P.negate());
      }
    }
    return Q;
  }

  public ECPoint wMOF(BigInteger d, ECPoint P) {
```

```
Date d1,d2;
long x1,x2;
ECPoint Q = new ECPointFp();
ECPoint P1 = new ECPointFp();
ECPoint[] G;
ECPoint infinty, K;
BigInteger tk;
int m, index, i;
String s;
char n_bits[];
m = Pre_Table.pow(2, Pre_Table.Win - 1);
int tw = Pre_Table.pow(2, Pre_Table.Win);
s = new String(" ");
s = d.toString(2);
n_bits = new char[s.length() + 1];
s.getChars(0, s.length(), n_bits, 1);
n_bits[0] = '0';
infinty = new ECPointFp();
Q = infinty;
i = s.length();
index = 0;
int j;
// precomputation  stage
G = pre_ECpoints(P);
while (i >= 1) {
  j = s.length() - i;
  if (n_bits[j] == n_bits[j + 1]) {
    Q = doubl(Q);
    i = i - 1;
  }
  else {
        index = (d.shiftRight(i - Pre_Table.Win)).intValue();
        index = (index & (2 * tw - 1)) - tw / 2;
        for (int jj = 1; jj <=(Pre_Table.Win - Pre_Table.zeta[index]); jj++) {
              Q = doubl(Q);
              i = i - 1;
              }
        if (Pre_Table.gama[index] > 0) {
          Q = add(Q, G[Pre_Table.gama[index]]);

      }
      else if (Pre_Table.gama[index] < 0) {
        Q = add(Q, G[ - (Pre_Table.gama[index])].negate());

      }
      for (int jj = 1; jj <= Pre_Table.zeta[index]; jj++) {
```

```
        if (i >= 0) {

        Q = doubl(Q);
        }
        i = i - 1;
       }
      }
   }

   if (i == 0) {
     Q = doubl(Q);
     if (n_bits[s.length()] == '1') {
       Q = add(Q, P.negate());
     }
   }
   return Q;
 }

public ECPoint D_exp(ECPoint Q,int k) {
   ECPointFp P2 = new ECPointFp();
   if (Q.isZero()) {
   return P2;
  }
  if (Q.y.isZero()) {
   return P2;
 }
 else {
   BigInteger[] A,B,C;
   BigInteger DK,FB,RB,BK_2,U, BK_4,BK_S,CK_S,BK_T4,T,T_2,A_B;
   DK=ZERO;
   FB=ONE;
   BK_4=ONE;
   A=new BigInteger[k+1];
   B=new BigInteger[k+1];
   C=new BigInteger[k+1];
   U=ONE;
   A[0]=Q.x.val;
   B[0]=(Q.y.val);
  // C[0]=Fp.Fp_add(Fp.Fp_mul( THREE,Fp.Fp_pow(A[0],2)),a4.val);
   for (int jj = 1; jj <= k ; jj++) {
        if (jj == 1)
          U = a4.val;
        else if (jj > 1)
          U = Fp.Fp_mul(U, BK_4);
        C[jj-1] = Fp.Fp_add(Fp.Fp_mul(THREE, Fp.Fp_pow(A[jj-1],
                                2)),Fp.Fp_mul(Fp.Fp_pow(SIXTEEN, jj-1), U));
```

```
            BK_2=Fp.Fp_mul(B[jj-1],B[jj-1]);
            BK_4=Fp.Fp_mul(BK_2,BK_2);
            A_B=Fp.Fp_mul(A[jj-1],BK_2);
            A[jj]=Fp.Fp_add (Fp.Fp_pow(C[jj-
1],2),Fp.Fp_mul(A_B,EIGHT).negate());
            B[jj]=Fp.Fp_add(Fp.Fp_mul(EIGHT,BK_4).negate(),Fp.Fp_mul(C[jj-1],

Fp.Fp_add(A[jj],Fp.Fp_mul(FOUR,A_B).negate())).negate());
          }
        for (int ii = 0;ii<k ;ii++)
           FB= Fp.Fp_mul(FB,B[ii]);
        FB=Fp.Fp_mul(FB,Fp.Fp_pow(TWO,k));
        T= Fp.Fp_inv( FB);
        T_2=Fp.Fp_pow(T,2);
        P2.x.val=Fp.Fp_mul(A[k],T_2) ;
        P2.y.val=Fp.Fp_mul(B[k],Fp.Fp_mul(T_2,T)) ;
     }
     return P2;
  }
  public ECPoint D_Mu(ECPoint Q,ECPoint P1,int k1, int k2 ) {
     ECPointFp P2 = new ECPointFp();
     BigInteger[] A,B,C;
     BigInteger DK,FB,RB,BK_2,U, BK_4,BK_S,CK_S,BK_T4,T,T_2,A_B
        ,t,s,m,x,y,FB_2,s_2,t_B,s_A,d;
     DK=ZERO;
     s_2=ONE;
     s=ONE;
     FB=ONE;
     BK_4=ONE;
     A=new BigInteger[Pre_Table.Win+1];
     B=new BigInteger[Pre_Table.Win+1];
     C=new BigInteger[Pre_Table.Win+1];
     U=ONE;
     if ((Q.isZero())||(Q.y.isZero())) {
      A[0]=P1.x.val;
      B[0]=P1.y.val;
      d = a4.val;
    }
   else {
     x=P1.x.val;
     y=P1.y.val;
     A[0]=Q.x.val;
     B[0]=(Q.y.val);
        for (int jj = 1; jj <= k1 ; jj++) {
            if (jj == 1)
              U = a4.val;
```

```
          else if (jj > 1)
            U = Fp.Fp_mul(U, BK_4);
          C[jj-1] = Fp.Fp_add(Fp.Fp_mul(THREE, Fp.Fp_pow(A[jj-1], 2)),
                              Fp.Fp_mul(Fp.Fp_pow(SIXTEEN, jj-1), U));
          BK_2=Fp.Fp_mul(B[jj-1],B[jj-1]);
          BK_4=Fp.Fp_mul(BK_2,BK_2);
          A_B=Fp.Fp_mul(A[jj-1],BK_2);
          A[jj]=Fp.Fp_add (Fp.Fp_pow(C[jj-
1],2),Fp.Fp_mul(A_B,EIGHT).negate());
          B[jj]=Fp.Fp_add(Fp.Fp_mul(EIGHT,BK_4).negate(),Fp.Fp_mul(C[jj-1],

Fp.Fp_add(A[jj],Fp.Fp_mul(FOUR,A_B).negate())).negate());
        }
        for (int ii = 0;ii<k1 ;ii++)
            FB= Fp.Fp_mul(FB,B[ii]);
        FB=Fp.Fp_mul(FB,Fp.Fp_pow(TWO,k1));
        FB_2=Fp.Fp_mul(FB,FB);
        s_A=Fp.Fp_mul(FB_2,x);
        if (A[k1].compareTo(s_A) != 0){
          s=Fp.Fp_add(A[k1],s_A.negate());
          t_B=Fp.Fp_mul(Fp.Fp_mul(FB_2,FB),y);
          t=Fp.Fp_add(B[k1],t_B.negate());
          s=Fp.Fp_add(A[k1],s_A.negate());
          m=Fp.Fp_add(A[k1],s_A);
          s_2=Fp.Fp_mul(s,s);
         A[0]=Fp.Fp_add(Fp.Fp_mul(t,t),Fp.Fp_mul(m,s_2).negate()) ;

B[0]=Fp.Fp_add(Fp.Fp_mul(Fp.Fp_mul(s,s_2),t_B),Fp.Fp_mul(t,Fp.Fp_add
                        (A[0],Fp.Fp_mul (s_2,s_A).negate()))).negate();
         d = Fp.Fp_mul(Fp.Fp_pow(SIXTEEN, k1), Fp.Fp_mul (Fp.Fp_mul(
                        U,Fp.Fp_mul(s_2,s_2)), BK_4));
        }
        else {
          U = Fp.Fp_mul(U, BK_4);
          C[k1] = Fp.Fp_add(Fp.Fp_mul(THREE, Fp.Fp_pow(A[k1], 2)),
Fp.Fp_mul
                                (Fp.Fp_pow (SIXTEEN, k1), U));
          BK_2=Fp.Fp_mul(B[k1],B[k1]);
          BK_4=Fp.Fp_mul(BK_2,BK_2);
          A_B=Fp.Fp_mul(A[k1],BK_2);
          A[0]=Fp.Fp_add
(Fp.Fp_pow(C[k1],2),Fp.Fp_mul(A_B,EIGHT).negate());
          B[0]=Fp.Fp_add(Fp.Fp_mul(EIGHT,BK_4).negate(),Fp.Fp_mul(C[k1],
                     Fp.Fp_add(A[0],Fp.Fp_mul (FOUR,A_B).negate())).negate());
          s=Fp.Fp_mul(B[k1],TWO);
          d = Fp.Fp_mul(Fp.Fp_pow(SIXTEEN, k1+1),Fp.Fp_mul(U, BK_4));
```

```
        }
    }
        for (int jj = 1; jj <= k2 ; jj++) {
              if (jj == 1)  U = d;
                 else if (jj > 1) U = Fp.Fp_mul(U, BK_4);
           C[jj-1] = Fp.Fp_add(Fp.Fp_mul(THREE, Fp.Fp_pow(A[jj-1],2)),
                                    Fp.Fp_mul(Fp.Fp_pow(SIXTEEN, jj-1), U));
           BK_2=Fp.Fp_mul(B[jj-1],B[jj-1]);
           BK_4=Fp.Fp_mul(BK_2,BK_2);
          A_B=Fp.Fp_mul(A[jj-1],BK_2);
          A[jj]=Fp.Fp_add (Fp.Fp_pow(C[jj-
1],2),Fp.Fp_mul(A_B,EIGHT).negate());
          B[jj]=Fp.Fp_add(Fp.Fp_mul(EIGHT,BK_4).negate(),Fp.Fp_mul(C[jj-1],


           Fp.Fp_add (A[jj],Fp.Fp_mul(FOUR,A_B).negate())).negate());
                }

      for (int ii = 0;ii<k2 ;ii++)
         FB= Fp.Fp_mul(FB,B[ii]);
      FB=Fp.Fp_mul(FB,Fp.Fp_pow(TWO,k2));
      FB=Fp.Fp_mul(FB,s);
      T= Fp.Fp_inv( FB);
      T_2=Fp.Fp_pow(T,2);
      P2.x.val=Fp.Fp_mul(A[k2],T_2) ;
      P2.y.val=Fp.Fp_mul(B[k2],Fp.Fp_mul(T_2,T)) ;
    return P2;
}
  protected Object clone() {
   return new ECurveFp( (Fp) a4, (Fp) a6);
  }
}
```

Bibliography

[1] A Certicom, white Paper. Remarks on the security of the elliptic curve cryptosystem. September 1997. http://www.certicom.com/research/wecc3.html

[2] ANSI X9.62,The elliptic curve digital signature algorithm (ECDSA), draft standard, 1997.

[3] Avanzi, R., A Note on the Signed Sliding Window Integer Recoding and a Left-to-Right Analogue, Selected Areas in Cryptography SAC 2004, LNCS 3357, Springer, 2004, pp. 130-143.

[4] borzoi 1.02 - an open source Elliptic Curve Cryptography Library by Dragongate Technologies Ltd. April 2004. http://www.dragongate-technologies.com

[5] Braden Phillips ,Minimal Weight Digit Set Conversions , Member, IEEE, and Neil Burgess, Member, IEEE

[6] Brown, M. , Hankerson, D. , Lopez, J. , and Menezes, A., Software Implementation of the NIST Elliptic Curves Over Prime Fields, Topics in Cryptology - CT-RSA 2001, LNCS 2020, (2001), 250-265.

[7] Cohen, H., Miyaji, A., Ono, T., Efficient Elliptic Curve Exponentiation Using Mixed Coordinates, Advances in Cryptology – ASIACRYPT ’98, LNCS 1514, Springer, 1998, pp. 51-65

[8] Diffie, W., and Hellman, M., New directions in cryptography, IEEE Transactions on Information Theory, vol. IT-22, no. 6, 1976, pp. 644-654

[9] Enge, A. Elliptic curves and their cryptography. Kluwer Academic Publishers, 1999.

[10] Gordon, D., A survey of fast exponentiation methods, Journal of Algorithms, vol.27, (1998), 129-146.

[11] Guajardo, J., Paar, C., "Efficient Algorithms for Elliptic Curves Cryptosystem", Advances in Cryptography-CRYPTO'97, LNCS, 1294(1997), Springer-Verlage, 342-356.

[12] J. Jedwab and C.J. Mitchell, Minimum Weight Modified Signed-Digit Representations and Fast Exponentiation,” Electronics Letters, vol. 25, no. 17, pp. 1171-1172, 1989.

[13] Joye, M., and Yen, S.-M., Optimal Left-to-Right Binary Signed-digit Exponent Recoding, IEEE Transactions on Computers 49(7), (2000), 740-748.

[14] Koyama, K. and Tsuruoka, Y., Speeding Up Elliptic Curve Cryptosystems using a Signed Binary Windows Method, Advances in Cryptology-CRYPTO '92, LNCS740, (1992), 345-357.

[15] Menezes, A. J., van Oorschot, P. C. and Vanstone. S. A., Handbook of Applied Cryptography. CRC Press,1997.

[16] Miller, V.S., Use of Elliptic Curves in Cryptography, Advances in Cryptology - CRYPTO '85, LNCS 218, Springer, 1986, pp. 417-426.

[17] Miyaji, A. , Ono, T., and Cohen , H., Efficient Elliptic Curve Exponentiation, Information and Communication Security - ICICS 1997, LNCS 1334, Springer, 1997, pp. 282-291.

[18] Moller, B. , Improved Techiques for Fast Exponentiation Information Security and Cryptology - ICISC 2002, LNCS 2587, Springer, 2003, pp.298-312.

[19] Monico, C., Semirings and Semigroup Actions in Public-Key Cryptography. PhD thesis, University of Notre Dame, May 2002

[20] Morain, F., Olivos, J., Speeding Up the Computations on an Elliptic Curve using Addition-Subtraction Chains, Theoretical Informatics and Applications, vol. 24, no. 6, 1990, pp.531-543.

[21] Muir, J., Stinson, D., Alternative Digit Sets for Nonadjacent

Representations, University of Waterloo,

[22] Muir, J., Stinson, D., Minimality and Other Properties of the Width-w Nonadjacent Form, University of Waterloo, Technical Report CORR 2004-08, 2004, available at http://www.cacr.math. uwaterloo.ca.

[23] Muir, J., Stinson, D., New Minimal Weight Representations for Left-to-Right Window Methods, University of Waterloo, Technical Report CORR 2004-19, 2004, available at http://www.cacr.math. uwaterloo.ca.

[24] National Institute of Standard and Technology, Digital Signature Standard, FIPS Publication 186-2, February 2000.

[25] NIST,FIPS PUB $180-2$: Secure Hash Standard, Aug. 2002.

[26] Okeya, K., Schmidt-Samoa, K., Semay, O., Takagi, T., Analysis of Some Efficient Window Methods and their Application to Elliptic Curve Cryptosystems.2004.

[27] Okeya, K., Schmidt-Samoa, K., Spahn, C., Takagi, T., Signed Binary Representations Revisited, Advances in Cryptology – CRYPTO 2004, LNCS 3152, Springer, 2004, pp. 123-139,

[28] Sakai, Y., Sakurai, K., Efficient Scalar Multiplications on Elliptic Curves with Direct Computations of Several Doublings. IEICE

Tranc.Fundamentals, E84-A No.1 (2001), 120-129.

[29] Schneier, B., Applied Cryptography: Protocols, Algorithms, and Source Code in C, Wiley, 1995.

[30] Smart N. P., A comparison of different finite fields for use in elliptic curve cryptosystems, June 2000

[31] Solinas, J. A. An improved algorithm for arithmetic on a family of elliptic curves. In Advances in Cryptology - CRYPTO '97 (1997), B. S. Kaliski, Jr., Ed., vol. 1294 of Lecture Notes in Computer Science, pp. 357- 371.

[32] Stallings.W., Cryptography and Network Security - Principles and Practice. Prentice Hall, 2nd edition, 1999.

[33] Win, E., Mister, S., Preneel, B., and Wiener, M., On the Performance of Signature Schemes Based on Elliptic Curves, Algorithmic Number Theory, ANTS-III, LNCS 1423, (1998), 252-266