A HIGH PERFORMANCE ARITHMETIC LIBRARY TO IMPLEMENT VARIOUS CRYPTOGRAPHIC ALGORITHMS

by

Bayram KULIYEV

June 2006

A HIGH PERFORMANCE ARITHMETIC LIBRARY TO IMPLEMENT VARIOUS CRYPTOGRAPHIC ALGORITHMS

by

Bayram KULIYEV

A thesis submitted to

the Graduate Institute of Sciences and Engineering

of

Fatih University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

June 2006
Istanbul, Turkey

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

_____
Prof. Dr. Kemal FİDANBOYLU
Head of Department

This is to certify that I have read this thesis and that in my opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____
Assist. Prof. Dr. Tuğrul YANIK
Supervisor

Examining Committee Members

Assist. Prof. Dr. Tuğrul YANIK _____

Assist. Prof. Dr. Veli HAKKOYMAZ _____

Assist. Prof. Dr. Tahsin UĞURLU _____

It is approved that this thesis has been written in compliance with the formatting rules laid down by the Graduate Institute of Sciences and Engineering.

_____
Assist. Prof. Dr. Nurullah ARSLAN
Director

Date
June 2006

A HIGH PERFORMANCE ARITHMETIC LIBRARY TO IMPLEMENT VARIOUS CRYPTOGRAPHIC ALGORITHMS

Bayram KULIYEV

M. S. Thesis - Computer Engineering
June 2006

Supervisor: Assist. Prof. Tuğrul YANIK

ABSTRACT

A finite field is an algebraic structure that plays an important role in theoretical foundation of cryptography. Almost all cryptographic algorithms are based on the properties of finite fields. In particular, elliptic curves combined with finite fields form a new type of cryptosystem called an elliptic curve cryptosystem. A cryptographic system can be regarded as a set of facilities implemented at hardware or software level and satisfies predefined requirements for information security. Security is the most obvious quality of any cryptosystem. However, an efficient implementation of a cryptosystem is also important in order to achieve a high performance.

A prime field $GF(p)$ is a finite field with prime number of elements that are represented as integers between 0 and the prime number p with respect to modular addition and multiplication operations. This fact makes it possible to implement prime field arithmetic operations efficiently on a general-purpose computer since elements of a prime field can be represented on a general-purpose computer as an array of integers. In this thesis, we concentrate on prime fields. We implemented multiprecision algorithms performing prime field arithmetic suitable for a general-purpose computer. For some operations multiple algorithms were considered and implemented. Their performance was measured and compared. In addition, we implemented arithmetic operations defined on points of elliptic curves over finite fields and the elliptic curve digital signature algorithm.

Keywords: Finite Field Arithmetic, Elliptic Curve Cryptography, Multiprecision Arithmetic, Public Key Cryptography, Digital Signature.

YÜKSEK PERFORMANSLI BİR ARİTMETİK İŞLEM KÜTÜPHANESİNİN GELİŞTİRİLMESİ VE BU KÜTÜPHANENİN ÇEŞİTLİ KRİPTOGRAFİK ALGORİTMALARIN YAZILMASINDA KULLANILMASI

Bayram KULIYEV

Yüksek Lisans Tezi – Bilgisayar Mühendisliği
Haziran 2006

Tez Yöneticisi: Yrd. Doç. Dr. Tuğrul YANIK

ÖZ

Sonlu cisimler kriptolojinin temeleni oluşturan önemli cebirsel yapılardır. Tüm kriptografik algoritmalar sonlu cisimlerin cebirsel özellikleri üzerine kuruludur. Özellikle, bu cebirsel yapılar üzerinde tanımlı eliptik eğriler, eliptik eğriler kriptosistemi adlı çok yaygın bir kriptosistem türü oluşturmaktadır. Kısaca kriptografik sistem yzılım veya donanım düzeyinde gerçekleştirilen ve belli bir öntanımlanmış güvenlik koşuluna uygun bir servis türüdür. Güvenlik herhangi bir kriptogrfik sistemin en önemli kriteridir. Fakat, bu sistemin yüksek performans sergileyebilmesi ve kullanışlı hale gelebilmesi için sistemin yazılım veya donanım düzeyinde etkin bir şekilde gerçekleştirilmesi gerekmektedir.

Bir p asal sayı için, üzerinde modüler toplama ve çarpma işlemleri tanımlanmış olan 0 ve $p - 1$ arasındaki sayılar kümesi asal cisim oluşturmaktadır. Bu sayılar genel amaçlı bilgisayrda kolayaca ifade edilebileceği gibi de cisim üzerinde tanımlanmış aritmetik işlemler için de hızlı kod geliştirilebilir. Bu tezde biz asal cisimler üzerinde yogunlaşıyoruz. Asal cisimler aritmetiğini gerçekleştiren çok-duyarlıklı genel-amaçlı bilgisayar için kod geliştirilmiştir. Bazı işlemler için birden fazla algoritmalar için kod gelişririlmiştir ve bunların zamanlaması kıyaslanmıştır. Ayrıca, asal cisimler üzerinde tanımlı eliptik eğriler aritmetiği için kod geliştirilmiştir. Son olarak bu kodlar kullanarak eliptik eğri dijital imza algoritması için kod geliştirilmiştir.

Anahtar Kelimeler: Sonlu Cisim Aritmetiği, Eliptik Eğrilere Dayalı Kriptografi, Büyük Sayı Aritmetiği, Açık Anahtar Kriptografi, Dijital İmza.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Assist. Prof. Dr. Tuğrul YANIK for his immense help in planning and executing the works and insight throughout the research.

I also give special thanks to Prof. Dr. Bariş KENDİRLİ and Assist. Prof. Dr. Tevfik BİLGİN for their valuable suggestions and comments during my research.

I express my thanks and appreciation to my family for their understanding, motivation and patience. Lastly, but in no sense the least, I am thankful to all colleagues and friends who made my stay at the university a memorable and valuable experience.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

A vast deal of digital information such as business transactions, private medical records, military and diplomatic actions are transmitted everyday over various public communication channels and stored on computers. With increased computerization of human life, information security has become of great importance to governmental and private organizations.

Cryptography is a branch of computer science that deals with design of cryptographic systems, which addresses issues concerning information security. The basic principle of any cryptosystems is to transform the original text to a sequence of unintelligible symbols thereby hiding the contents of the original text. This transformation process is carried out by means of an additional parameter called a key. Depending on the type of key used for transformation of text there are two types of widely used cryptographic systems: symmetric key cryptosystems and public key cryptosystems.

All cryptographic systems until 1970s were based on the concept of symmetric key. The practical application of the public-key cryptography started in 1976 when Whitfield Diffie and Martin Hellman introduced the idea of public-key cryptography and described key exchange algorithm. The idea of the public key cryptography is based on difficulty of solving a computational problem.

Since then a great deal of research has been conducted concerning its security level and efficient implementation and many public-key cryptosystems were described. For

example, Neal Koblitz and Victor Miller proposed a different type of public key cryptosystem, which exploits elliptic curves. However many of the proposed cryptosystems were shown to be insecure. There are three types of widely used public-key cryptosystems, which were proven to be conditionally secure: those that based on the difficulty of solving integer factorization problem, those that based on the difficulty of solving discrete logarithm problem and those that based on the difficulty of solving the elliptic curve discrete logarithm problem.

In this thesis we develop a high performance arithmetic library that is used to implement various public-key cryptographic algorithms. Since the mathematical foundation of cryptography is using the algebraic structures called finite fields, we concentrate specifically on the implementation of operations on prime fields, i.e., fields with a prime number of elements. These basic operations are addition, subtraction, multiplication and inversion.

Furthermore, elliptic curves over finite fields form another algebraic structure called group, which is used to design elliptic curve cryptosystems. We implement basic operations on these groups, i.e. addition and using them implement the elliptic curve digital signature algorithm.

Chapter 2 introduces basic algebraic structures, i.e., groups and fields. Definition of a cyclic group and a prime field are given. Chapter 3 provides a short background of elliptic curves. We give the definition of an elliptic curve over a finite field and outline its group structure together with algebraic operations on the elements of the group.

In Chapter 4 we give basic facts about cryptographic systems. We start with statement of basic computational problems used to design cryptosystems. We give definitions of symmetric key cryptosystems and public key cryptosystems together with their properties. Finally, a brief description of well-known public key cryptosystems is given.

Chapter 5 explains representation of prime fields followed by various multiprecision algorithms for implementing prime field arithmetic operations. We describe addition and subtraction operations. Three modular reduction algorithms, classical algorithm, Barrett's reduction algorithm and Montgomery algorithm are introduced. Modular multiplication is implemented using Montgomery's multiplication algorithm. We also describe different exponentiation algorithms: binary method, *m*-ary method, and *m*-ary recoding method. The classical inversion operation is implemented according to the Montgomery's inversion method. The elliptic curve digital signature algorithm is described in Chapter 6.

CHAPTER 2

FINITE FIELDS

2.1 INTRODUCTION

Groups, rings and fields are fundamental algebraic structures of abstract mathematics. These structures play an important role in theory of cryptography and cryptanalysis. Almost all cryptographic algorithms are based on the computational properties of groups, rings and fields. For example, the set of integer numbers forms a ring whose elements can be factorized uniquely into prime numbers. This property lays a foundation for the RSA cryptosystem.

In abstract algebra we are concerned with a set of abstract elements together with a well-defined binary operation on the elements of the set. Using this operation we combine any two elements to obtain a third element of the set. For example, let **Z** be a set of all integer numbers. Then an ordinary addition operation (+) is well defined on **Z**, that is, we can take two elements from the set and add them up to obtain a third element of the set. We should note that for any element a and b of **Z**, $a + b$ is always in **Z**. This means that the addition operation is *closed* over the set of all integer numbers.

Formally, a *group* $(G, *)$ is a set of elements with a binary operation $*$ such that the elements of G closed over the operation $*$, and the following rules hold:

1. For any a, b, c in G, $(a * b) * c = a * (b * c)$, the *associative law*;
2. There exists a distinguishing element e, called the *identity* element, in G such that for any a in G, $a * e = e * a$;

3. For any a in G, there exists a unique element a^{-1} in G, called the *inverse of a*, such that $a^{-1} * a = a * a^{-1} = e$.

Traditionally for any group $(G, *)$ the binary operation $*$ is called a multiplication operation. The exponentiation operation a^n for an arbitrary element a in G and an integer n is defined as $a * \ldots * a$, multiplication of a n times. Further, we define $a^0 = e$ and a^{-n} as $(a^{-1})^n$.

A group is called an abelian or commutative group if for any two elements of group G, we have $a * b = b * a$. A group with a finite number of elements is called a *finite* group. The number of elements in a group is said to be the *order* of the group and denoted by $|G|$. Finite groups have finite orders, while groups with infinite number of elements have an infinite order. Any subset G' of a group $(G,*)$ is said to be a *subgroup* of $(G,*)$ if $(G',*)$ is a group. This fact is denoted by $G' \leq G$.

For example, the set **Z**, of all integer numbers, is an abelian group with respect to the ordinary addition operation (+). The identity element of the **Z** is *0* (zero) since for any element a in Z, $a + 0 = 0 + a = a$. Also, $-a$ is the inverse of a as $a + (-a) = (-a) + a = 0$. Furthermore, the order of **Z** is infinite. The set of all even integers, denoted by **2Z**, is a subgroup of **Z**, i.e., **2Z** ≤ **Z**.

2.2 CYCLIC GROUPS

Let $(G, *)$ be a group and g be a distinguished element of G where for any element a in G there exists an integer power n such that $a = g^n$. The element g is said to *generate* the group G and is called a *generator* of G. This is denoted by $\langle g \rangle = G$.

A group $(G, *)$ that can be generated by a single element g in G is said to be a *cyclic* group generated by g. All cyclic groups are abelian and may have finite or infinite order.

For example, let us consider a set of positive integers $Z_p^* = \{1, 2, \ldots, p-1\}$ where p is a prime integer. Let us define a binary operation $\otimes$ on the set Z_p^* as follows. For any a and b in Z_p^* define $a \otimes b$ as (ab) mod p. Then $(Z_p^*, \otimes)$ is a cyclic group. In particular, $\langle 3 \rangle = Z_7^*$.

2.3 PRIME FIELDS

A *field* is a set F with two binary operations + and $*$, traditionally called addition and multiplication operations such that both $(F, +)$ and $(F^*, *)$ are abelian groups. F^* denotes a subset of nonzero elements of F, i.e. all elements of F except the identity element of the group $(F, +)$, where the operations + and $*$ are associated with each other by means of the *distributive law* defined as follows:

1. For any a, b, c in F, $(b + c) * a = (b * a) + (c * a)$;
2. For any a, b, c in F, $a * (b + c) = (a * b) + (a * c)$.

For example, sets of rational and real numbers are fields with infinite number of elements.

A field containing a finite number of elements is said to be a *finite field*. The number of element in a finite field is always a power of a prime p. A finite field of order p^n is denoted by F_p or $GF(p^n)$. *GF* means Galois field in honor of the mathematician who first studied finite fields. For any finite field $GF(p^n)$, the prime integer p is called the *characteristic* of the finite field. The characteristic of fields with infinite number of elements is zero. It should be noted that an important property of fields is that two fields with equal number of elements have the same structure, i.e. they are *isomorphic*.

Let $GP(p^n)$ be a finite field such that $n = 1$. Then we have $GF(p)$ that is said to be a *prime field*. In other words, a finite field whose order is a prime integer is called a prime field. Because of the isomorphism property of fields any prime field of order p can be

represented using a set of positive integers $\{0, 1, \ldots, p - 1\}$ together with binary operations $\otimes$ and $\oplus$ defined for any elements a and b as follows: $a \oplus$ b is $(a + b)$ mod p, and $a \otimes$ b is (ab) mod p. These operations are modular operations over the set $\{0, 1, \ldots, p - 1\}$ where the modulus is p.

Any prime field $GF(p)$ is a cyclic group with respect to the operation $\oplus$, i.e. $\langle 1 \rangle = GF(p)$ for any prime p. Similarly, nonzero elements of $GF(p)$ forms a cyclic group with respect to the operation $\otimes$. A generator of the last group is called a *multiplicative generator* of the field $GF(p)$. For example, 3 is a multiplicative generator of $GF(7)$.

CHAPTER 3

ELLIPTIC CURVES

3.1 DEFINITION

An elliptic curve (Koblitz N., 1994) over some field F is a set of points in a two dimensional plane which satisfies a cubic equation of the general form $y^2 + axy + by = x^3 + cx^2 + dx + e$ where a, b, c, d and e belong to F. For cryptographic purpose it is sufficient to consider elliptic curves over finite fields with characteristic other than 2 and 3. Then the cubic equation is of the form $y^2 = x^3 + ax + b$.

An *elliptic curve* $E(F_p)$ over a finite field F_p of characteristic neither 2 or 3 is a set of points (x, y) on a two dimensional plane with x, y in F_p satisfying the equation $y^2 = x^3 + ax + b$, where a, b in F_p and $\Delta = -16(4a^3 + 27b^2) \neq 0$, together with a distinguished element O. O is called the "point at infinity". Δ is said to be the discriminant, where $\Delta = 0$ means that $x^3 + ax + b$ has no multiple roots.

3.2 GROUP STRUCTURE OF ELLIPTIC CURVES

There are two basic mathematical operations on an elliptic curve over a finite field F_p: negation and addition. Combining points on an elliptic curve by means of these operations we can obtain new points on the elliptic curve. Geometrically, negation and addition are defined as follows:

1. Negation rule: for $P = (x, y)$ in $E(F_p)$, $-P = (x, -y)$, i.e. the negative of the point P is the point with the same x coordinate but the negative y coordinate;
2. For $P = (x, y)$ in $E(F_p)$, $P + O = O + P = P$, $P + (-P) = (-P) + P = O$;
3. $O + O = O$, $-O = O$;
4. For $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $P \neq \pm Q$, $P + Q$ is obtained as follows. We draw a chord between points P and Q. Then we find the third point of intersection of the line passing through the points P and Q with the curve. The point S symmetric to this point relatively to the x-axis is the sum of P and Q. The figure 1 depicts the addition operation.
5. For $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ such that $P = Q$, $P + Q = 2P$ is found similarly. Instead of chord we draw the tangent line at the point P (figure 2). This operation is called *doubling*.

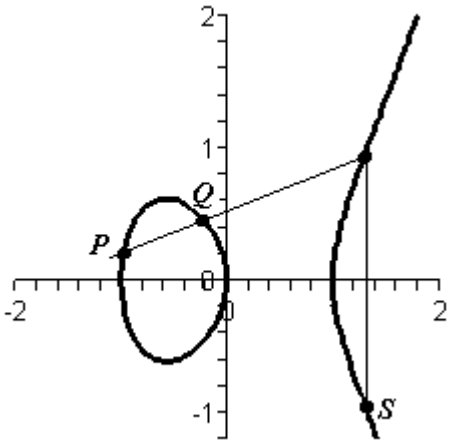


Figure 3.1 Geometric addition of two distinct points on an elliptic curve.

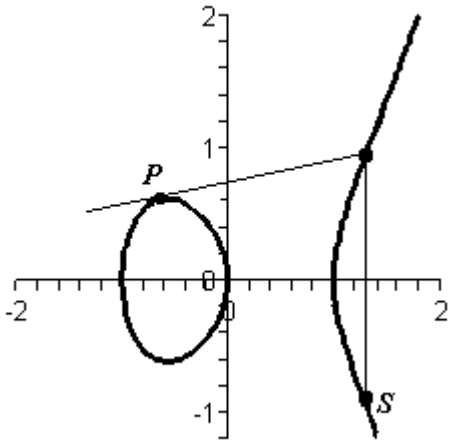
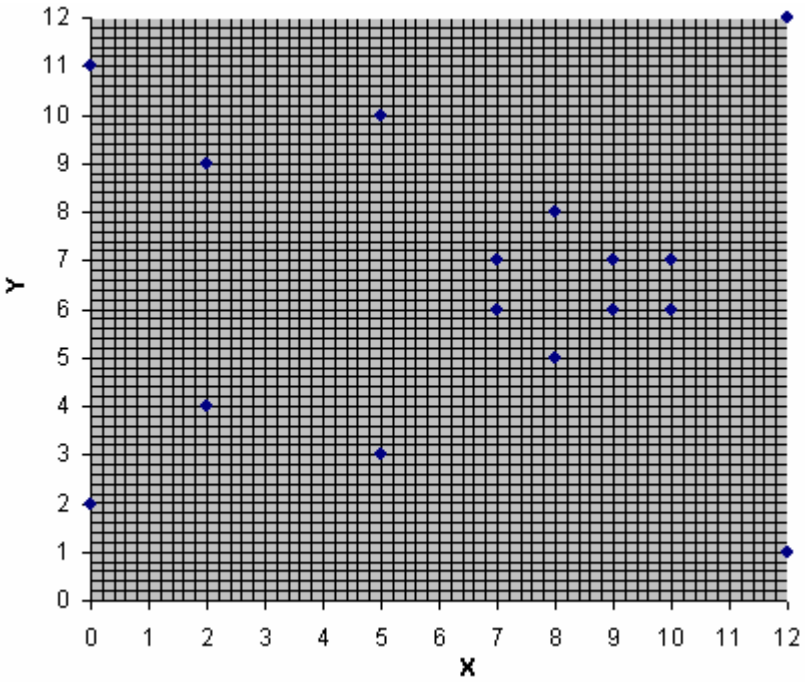


Figure 3.2 Geometric doubling of a point on an elliptic curve.

Points of an elliptic curve together with the operation of addition, as we have defined, forms an abelian group where the "point at infinity" serves as the identity element of the group and the inverse of an elliptic curve point P is its negation $-P$. Scalar multiplication of an elliptic curve point P by a positive integer k is defined as the sum of k copies of P and denoted by kP. Similarly, $-kP$ is defined as $k(-P)$, i.e, the sum of k copies of $-P$.

For example, the figure 3 depicts points of the elliptic curve $y^2 = x^3 + 2x + 4$ over the field F_{13}.



Points on $E(F_{13})$

(0, 2)
(0, 11)
(2, 4)
(2, 9)
(5, 3)
(5, 10)
(7, 6)
(7, 7)
(8, 5)
(8, 8)
(9, 6)
(9, 7)
(10, 6)
(10,7)
(12, 1)
(12, 12)
O

Figure 3.3 Elliptic curve points in $E(F_{13})$, a=2 and b = 4.

There are several coordinated systems to represent elliptic curves (Cohen H. et al., 1998). Three basic well-known coordinate systems are affine, projective and Jacobian coordinate systems. We will give addition and doubling algebraic formulas in affine and projective coordinate systems.

3.3 CURVE ADDITION AND DOUBLING FORMULAS

Let $y^2 = x^3 + ax + b$ be the equation of an elliptic curve over F_p, where $a, b \in F_p$ and $4a^3 + 27b^2 \neq 0$. For any elliptic curve points P and Q we will define $P + Q$ in two different ways: for affine and projective coordinate systems (Silverman J. H.).

Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $P + Q = (x_3, y_3)$. In the affine coordinate system addition and doubling formulas are defined as following.

- Addition formulas ($P \neq \pm Q$): $x_3 = \lambda^2 - x_1 - x_2$, $y_3 = \lambda(x_1 - x_3) - y_1$, where $\lambda = (y_2 - y_1)/(x_2 - x_1)$;
- Doubling formulas ($P = Q$): $x_3 = \lambda^2 - 2x_1$, $y_3 = \lambda(x_1 - x_3) - y_1$, where $\lambda = (3x_1^2 + a)/2y_1$.

In the projective coordinate system we substitute variable *x* and *y* such that $x = X/Z$ and $y = Y/Z$ hence obtaining a new elliptic curve equation $Y^2Z = X^3 + aXZ^2 + bZ^3$. Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_3)$ and $P + Q = (x_3, y_3)$. Then addition and doubling formulas take the following form:

- Addition formulas ($P \neq \pm Q$):
  $X_3 = vA, Y_3 = u(v^2X_1Z_2 - A) - v^3Y_1Z_2$, $Z_3 = v^3Z_1Z_2$, where $u = Y_2Z_1 - Y_1Z_2$, $v = X_2Z_1 - X_1Z_2$, $A = u^2Z_1Z_2 - v^3 - 2v^2X_1Z_2$.
- Doubling formulas ($P = Q$): $X_3 = 2hs$, $Y_3 = w(4B - h) - 8Y_1^2s^2$, $Z_3 = 8s^3$, where $w = aZ_1^2 + 3X_1^2$, $s = Y_1Z_1$, $B = X_1Y_1s$, $h = w^2 - 8B$.

Unlike in affine coordinate systems in projective coordinates we do not perform inversion operation to compute curve addition and doubling.

CHAPTER 4

CRYPTOGRAPHIC SYSTEMS

4.1 INTRODUCTION

Today we live in a highly networked society where the communication is an important part of human activity. All kind of sensitive information such as business transactions, diplomatic or military actions, and commercial information is transmitted over various public or secret communication channels. Organizations that handle secret information need to protect it in an efficient way. This is the reason for the need of a reliable cryptographic system that detects and prevents from any action which compromises the security of secret information owned by an organization.

Any system that provides secure information transmission and safeguarding of information is referred as a *cryptographic system*. Basically, a cryptographic system provides following four basic services (Menezes A. J. et al., 1997):

- **confidentiality:** this service prevents unauthorized disclosure of sensitive information;
- **authentication:** service that provides proof of identity of the sender to the recipient, so that the recipient can be assured that the sender is who he or she claims to be;
- **data integrity:** service that ensures that an unauthorized person does not modify the contents of message;
- **non-repudiation:** this service prevents the denial of a previous action.

One of the essential ingredients of a cryptographic system is a transformation of original text, called a *plaintext*, to an illegible version of the plaintext called a *ciphertext*. Modern cryptographic systems do not depend on the secrecy of an algorithm used to encrypt or decrypt the plaintext. Encryption process uses a key to encrypt or decrypt information.

Formally, encryption and decryption processes are defined as follows. Let P and C be two sets of messages, i.e. sets of strings, defined over sets of symbols (alphabets) A and B respectively. In other words P is a set of all plaintexts while C is a set of all ciphertexts. Furthermore, let e and d belong to a set of keys K. Then the encryption process associated with the key e is defined as a one-to-one onto function E_e: $P \rightarrow C$. The decryption process associated with the key d is defined similarly as a one-to-one onto function D_d: $C \rightarrow P$.

4.2 COMPUTATIONAL PROBLEMS FOR CRYPTOSYSTEMS

Public key cryptographic systems rest on computational intractability of mathematical problems (Koblitz N., 1998). Theoretically, a problem is intractable if there is no algorithm that solves the problem in polynomial time as a function of input length. There are many mathematical problems that are intractable, however, only three of them are shown to be most efficient and secure. These are the integer factorization problem, the discrete logarithm problem for $GF(p)$ and the elliptic curve discrete logarithm problem.

The integer factorization problem is, given a positive integer n, to find all prime factors of the integer n (Adleman L. M. et al., 1994). To define the discrete logarithm problem let $GF(p)$ be a prime field with a multiplicative generator g. Then the discrete logarithm problem is determining for an arbitrary non-identity element a of $GF(p)$ an exponent x so that $a = g^x$. Similarly, let $E(F_p)$ be an elliptic curve over a finite field F_p. Suppose Q and P are points on the elliptic curve such that $Q = xP$ for some integer x. Then, given the points Q and P, the elliptic curve discrete logarithm problem is to find the integer x.

These problems are currently considered intractable, i.e. they are widely believed to be intractable. There is no known proof that claims whether or not there is a polynomial time algorithm that solves any of these problems.

4.3 SYMMETRIC-KEY CRYPTOSYSTEMS

Let (e, d) be encryption and decryption keys, respectively. A cryptosystem system is said to be a symmetric-key cryptosystem if from knowing d it is computationally easy to determine e and vise versa (Menezes A. J. et al., 1997). A symmetric-key cryptosystem is sometimes referred as a private key cryptosystem since in practice we take $e = d$, i.e. the same key is used to encrypt and decrypt information. The private key is kept secret.

Schematically, symmetric-key encryption/decryption process can be described as follows. Let P and C be respectively sets of all possible plain and cipher texts, K be a set of all possible keys and M be the set of all messages. Then a symmetric-key cryptosystem is defined as a pair of functions E_e: $P \rightarrow C$ and D_d: $C \rightarrow P$ such that e and d belong to K. Since the correspondence between plain and cipher text is one-to-one onto, we have also $D_d(E_e(m)) = m$, for some m in M.

Two main reasons for symmetric-key cryptosystems to be the method of choice are that key sizes are relatively short and fast hardware and software implementations exist. However, in a large network the key management becomes inefficient since each group in a network must have their individual pair of keys (e, d). Examples of common symmetric-key cryptosystems are The One-Time Pad, DES, AES, RC5.

4.4 PUBLIC-KEY CRYPTOSYSTEMS

A notion for the public key cryptosystems was proposed in 1976 by Whitfield Diffie and Martin Hellman. Unlike in a symmetric-key cryptosystem, there is no need to keep an encryption key secret for a sender to make any secret arrangement with the recipient.

Let M be the set of all possible messages and (e, d) be respectively encryption and decryption keys together with encryption and decryption functions E_e and D_d. If for a pair of plain and cipher text (m, c), we encrypt as $E_e(m) = c$, it is computationally infeasible to determine d from e and e from d, then the functions E_e and D_d form a public key cryptosystem, and e and d are called respectively a public key and a secret key.

4.4.1 The RSA Cryptosystem

RSA cryptosystem was first proposed in 1977 by Rivest, Shamir and Adlemann. RSA encryption algorithm is based on the difficulty of factoring integer numbers. RSA public key encryption scheme works as follows:

1. Select randomly two large prime numbers p and q, where $p \neq q$;
2. Compute $n = pq$;
3. Select an odd integer e relatively prime to $(p - 1)(q - 1)$;
4. Compute d as a the multiplicative inverse of e modulo $(p - 1)(q - 1)$;
5. Publish the pair (e, n) as an RSA public key, also called encryption key;
6. Keep secret the pair of integers (d, n) as an RSA secret key, also called decryption key.

Assume that a message is partitioned into smaller blocks and each message block is identified with integer m such that $0 \leq m < n$. The transformation E_e of a plaintext m to a ciphertext c using an RSA public key (e, n) is performed as

$$E_e(m) = m^e \bmod n.$$

The transformation D_d of a ciphertext c to a plaintext m using an RSA secret key (d, n) is performed as

$$D_d(c) = c^d \bmod n.$$

We justify the correctness of the RSA algorithm by considering n as a product of an arbitrary number of prime numbers instead of two. Let n be a product of k prime integers

$p_1, p_2, \ldots, p_k$, and m is a positive integer. Further, suppose that e and d are positive integers such that $ed \equiv 1 (\mathrm{mod}\, \varphi(n))$. Since $\varphi(n) = (p_1 - 1)\cdots(p_k - 1)$, it follows that $ed \equiv 1 (\mathrm{mod}(p_i - 1))$ for $i = 1, \ldots, k$. If m is divisible by p_i then it is trivially true that $m^{ed} \equiv m (\mathrm{mod}\, p_i)$, otherwise $\gcd(p_i, m) = 1$ and hence by Fermat's little theorem $m^{p_i - 1} \equiv 1 (\mathrm{mod}\, p_i)$. On the other hand, we have $ed \equiv 1 (\mathrm{mod}(p_i - 1))$ meaning that $ed = 1 + t(p_i - 1)$ for some integer t and hence we have $m^{ed} \equiv m^{(p_i - 1)t} m \equiv m (\mathrm{mod}\, p_i)$. Thus we obtain $m^{ed} \equiv m (\mathrm{mod}\, p_i)$ for $i = 1, \ldots, k$. Since $\gcd(p_i, p_j) = 1$ for $i, j = 1, \ldots, k$ and $i \neq j$, it directly follows that $m^{ed} \equiv m (\mathrm{mod}\, p_1 p_2 \cdots p_k)$, where $n = p_1 p_2 \cdots p_k$. Now, suppose that $m^{ed} \equiv m (\mathrm{mod}\, n)$ and $m^{ed} \equiv m' (\mathrm{mod}\, n)$. Then it follows $m \equiv m' (\mathrm{mod}\, n)$ implying $m = m'$ since both m and m' are less than n. Therefore, the function D_d does recover the original message and for any plaintext m there is exactly one ciphertext $c = E_e(m)$.

Since the RSA cryptosystem rests on the difficulty of factoring integers, it would be easy to break the RSA cryptosystem if factoring integers could be performed in polynomial time.

4.4.2 The ElGamal Cryptosystem

The ElGamal cryptosystem takes advantage of intractability of the discrete logarithm problem. Assume we have defined a large prime field $GF(p)$ and an element g such that g is a generator of the field $GF(p)$. We partition the original message into smaller blocks and identify each message block with integer m such that $0 \leq m < p - 1$. The ElGamal cryptosystem works as following:

1. We choose an integer a in the range $(0, p - 1)$ and keep it secret;
2. We publish the integer g^a as the public key;
3. To encrypt a message m associated with the public key g^a we choose an integer k at random and send the pair (g^k, mg^{ak}) to a recipient;
4. To recover the message m the recipient multiplies mg^{ak} by $(g^{ak})^{-1}$.

To justify the correctness of the scheme it is enough to note that both integers m and g^{ak} are elements of $GF(p)$ and hence their product mg^{ak} is also an unique element of $GF(p)$. Multiplying mg^{ak} by $(g^{ak})^{-1}$ we obtain the original message m since $mg^{ak}(g^{ak})^{-1} = m$. The uniqueness of the encryption and decryption operations follows from the fact that the multiplication operation in finite fields is defined uniquely, i.e., for any elements $x, y \in GF(p)$ if $xy = z$ and $xy = z'$ then $z = z'$.

Breaking the ElGamal cryptosystem is equivalent to solving the discrete logarithm problem for $GF(p)$. Hence anyone who can solve the discrete logarithm in $GF(p)$ can break the ElGamal cryptosystem. Theoretically, if there were a way to compute g^{ab} knowing only g^a and g^b the cryptosystem also might be broken without solving the discrete logarithm problem.

4.4.3 The Diffie-Hellman Key Exchange System

Whitfield Diffie and Martin Hellman in 1976 introduced the idea of public key cryptography that addressed the key management problem known as the key exchange protocol. The protocol provides a way to securely accomplish securely a key establishment process whereby a shared key becomes available for two parties for subsequent cryptographic use.

Let $GF(p)$ be a prime field and g a generator of $GF(p)$, which is public. Suppose users A and B want to agree upon a key. The protocol works as follows:

1. A chooses a random positive integer a and keeps it secret;
2. A computes g^a and publishes it as his or her public key;
3. B chooses a random positive integer b and keeps it secret;
4. B computes g^b and publishes it as his or her public key;

5. The secrete key, therefore, is g^{ab} that can be computed easily by both users for later usage.

The Diffie-Hellman key exchange system is based on the intractability of the discrete logarithm problem meaning that by solving the discrete logarithm problem we can break the Diffie-Hellman key exchange system.

4.4.4 Digital Signatures

A digital signature serves the same purpose as a handwritten signature, which is used to provide authentication, data integrity and non-repudiation. The Digital Signature Standard (DSS) was proposed in 1991 by the U.S. government's National Institute of Standard and Technology (NIST). The standard is based on the Digital Signature Algorithm (DSA) and provides a standard signature method. The DSA consists of three parts: setup scheme, signing scheme and a verification scheme. It works as follows:

- **Setup scheme**

  1. Choose two prime integers p and q of size 512 and 160 bits respectively, where $p \equiv 1 \bmod q$;
  2. For a random element g_0 in $GF(p)$ different from zero compute a generator $g = (g_0)^{(p-1)/q}$ of nontrivial subgroup of $GF(p)^*$ – a set of nonzero elements of the prime field $GF(p)$;
  3. Take a random integer d such that $1 < d < q$ and make it a secrete key;
  4. Publish $e = g^d$ as the public key.

- **Signing scheme**

  1. Apply a hash function H to a message m: $0 < H(m) < q$;
  2. Take a random integer k and compute g^k that is also in $GF(p)^*$;
  3. Set $r = g^k \bmod q$;
  4. Find an integer s such that $s \equiv (k^{-1}(H(m) + dr)) \bmod q$;
  5. Set a signature as a pair (r, s).

- **Verification scheme**
  1. Compute $x = s^{-1}H(m) \bmod q$;
  2. Compute $y = s^{-1}r \bmod q$;
  3. Compute $t = g^x e^y$ in $GF(p)$;
  4. If $r \equiv t \bmod q$ then the verification is certifiable.

The DSA algorithm is originally based on ElGamal and Schnorr's work and rests on the intractability of the discrete logarithm problem. To break the system one needs to solve the discrete logarithm problem.

4.4.5 Elliptic Curve Cryptosystem

In recent years elliptic curve cryptosystems have been used more and more in public-key cryptography. A vast amount of work has been done on applications of elliptic curve cryptography. The idea of using an elliptic curve over a finite field $GF(p)$ in public key cryptosystems was independently proposed in 1985 by N. Koblitz and V. Miller. Elliptic curve cryptosystems provide a high level of security with small key sizes that makes them more appropriate for many cryptographic applications such as mobile communication and smart cards. Also, recently an extensive and careful study of the software implementation on workstations of the NIST-recommended elliptic curves over prime fields has been made. A typical and popular application of elliptic curves is an analogue of DSA – the Elliptic Curve Digital Signature Algorithm, which was accepted in 2000 as NIST and IEEE standards.

The security of elliptic curve cryptosystems is based on the intractability of the discrete logarithm problem for points of an elliptic curve (Menezes A. J., 1993). Unlike the discrete logarithm and integer factorization problems no algorithm with sub-exponential running time solves the elliptic curve discrete logarithm problem.

CHAPTER 5

PRIME FIELD ARITHMETIC

5.1 INTRODUCTION

An efficient software implementation of prime field arithmetic is a crucial factor in cryptographic applications such as RSA, (elliptic curve) digital signature algorithms and Diffie-Hellman key exchange system. The basic arithmetic operations in the prime field $GF(p)$ are addition, subtraction, multiplication and inversion. In order to obtain cryptographic high-speed software implementation for embedded systems or general-purpose computers, theses operations should be implemented thoroughly in a manner suitable for systems having 32-bit architecture. The algorithms proposed in this chapter perform word-level (32-bit) multiprecision prime field arithmetic that is much faster with comparison to bit-level implementation. For some arithmetic operations we consider several algorithms. Their timing results of our implementations are provided as well.

5.2 REPRESENTATION OF PRIME FIELDS

We perform modular arithmetic operations in $GF(p)$ where the elements can represented as the set of integers $\{0, 1, \ldots, p - 1\}$. To represent any one of these elements in a general-purpose computer, depending on the length of the modulus p, we need multiple 32-bit words. This is actually an array of words. This representation is also known as multiprecision representation. Arithmetic done with these elements is called multiprecision arithmetic. To create a scalable general-purpose cryptographic application we must not

place any restrictions on the modulus p and its length, i.e. the length of the modulus may be 160 or 2048 bits, while requiring that word size be 32-bit length.

Let k be the bit length of the modulus p, i.e., $k = \lceil \log_2 p \rceil$, w be the word size (usually w = 8, 16 or 32), $s = \lceil k/w \rceil$ – the exact number of words to represent the prime modulus p and $m = sw$ the total number of bits of s. Thus an element of $GF(p)$ is represented as a s word array of unsigned integers. We will denote an element in $GF(p)$ as $A = (A[s-1]\, A[s-2] \ldots A[1]\, A[0])$, where $A[i]$ is a one word unsigned integer. $A[s-1]$ is the most significant word (MSW) and $A[0]$ is the least significant word (LSW). Similarly, we represent A using bit-level representation as $A = (a[k-1]\, a[k-2] \ldots a[1]\, a[0])$, where $a[k-1]$ is the most significant bit and $a[0]$ is the least significant bit. If k is not a multiple of w then we will represent A as $s = \lceil k/w \rceil$ words of unsigned integers such that exactly $k \bmod w$ least significant bits of $A[s-1]$ are occupied, and $w - (k \bmod w)$ most significant bits of $A[s-1]$ are all zero.

$A[s-1]$	$A[s-2]$	…	$A[1]$	$A[0]$

Figure 5.1 Representation of a in $GF(p)$ as an array of w-bit unsigned integers.

Alternatively, w-bit representation of an integer can be interpreted as a representation of the integer to the base 2^w, where w is usually 8, 16, or 32. Thus, if $b = 2^w$ then the decimal representation of an element $A = (A[s-1]\, A[s-2] \ldots A[1]\, A[0])$ from $GF(p)$ is equal to $A[s-1]b^{s-1} + A[s-2]b^{s-2} + \cdots + A[1]b + A[0]$.

5.3 MODULAR ADDITION AND SUBTRACTION

Let A and B be elements of $GF(p)$. To compute $C = A + B$ we first perform an ordinary multiprecision addition operation (McEliece R. J.). Since the result must be in $GF(p)$ we check whether or not $C = A + B$ is greater than or equal to p. If $C > p$ we set $C = p - C$. We will use the assignment operation $(C[i], \varepsilon) = A[i] + B[i] + \varepsilon$, which means that

we add up $A[i]$, $B[i]$ and the previous 1-bit carry ε (that is equal to 0 or 1). The result is in $C[i]$, where ε is set to 1 if a carry occurred, 0 otherwise. We will denote the multiprecision subtraction operation $A - B$ by $Sub(A, B)$. The addition algorithm is given bellow:

Modular Addition Algorithm

INPUT: p, $A = (A[s-1]\ldots A[1]\ A[0])$, $B = (B[s-1]\ldots B[1]\ B[0])$ and A, B in $GF(p)$.

OUTPUT: $C = (C[s-1]\ldots C[1]\ C[0]) = A + B$ in $GF(p)$.

Step 1. $(C[0], \varepsilon) = A[0] + B[0]$

Step 2. for $i = 1$ to $s - 1$

Step 3. $(C[i], \varepsilon) = A[i] + B[i] + \varepsilon$

Step 4. if $C \geq p$ then $C = Sub(p, C)$

Step 5. return $(C[s-1]\ldots C[1]\ C[0])$

Similarly, we compute $C = A - B$ (McEliece R. J., 1987). If the difference is less than zero we perform the extra step $C = p + C$. We define a notation for subtraction operation as $(C[i], \varepsilon) = A[i] - B[i] - \varepsilon$, which means we subtract $B[i]$ and the previous 1-bit borrow ε from $A[i]$ such that the difference is in $C[i]$. ε is set to 1 if a 1-bit borrow occurrs, 0 otherwise. Let us denote the multiprecision addition operation $A + B$ by $Add(A, B)$. The subtraction algorithm is outlined bellow:

Modular Subtraction Algorithm

INPUT: p, $A = (A[s-1]\ldots A[1]\ A[0])$, $B = (B[s-1]\ldots B[1]\ B[0])$ and A, B in $GF(p)$.

OUTPUT: $C = (C[s-1]\ldots C[1]\ C[0]) = A - B$ in $GF(p)$.

Step 1. $(C[0], \varepsilon) = A[0] - B[0]$

Step 2. for $i = 1$ to $s - 1$

Step 3. $(C[i], \varepsilon) = A[i] - B[i] - \varepsilon$

Step 4. if $C < 0$ then $C = Add(p, C)$

Step 5. return $(C[s-1]\ldots C[1]\ C[0])$

5.4 MODULAR REDUCTION

One of the basic frequently used arithmetic operations in public key cryptography is the modular reduction. The modular reduction operation is a time critical step in the implementation of the modular multiplication and exponentiation operations. The efficiency of the algorithms mainly depends on a good implementation of the modular reduction.

There are three basic well-known algorithms for the modular reduction operation, which may be implemented to run on general-purpose computers: the classical algorithm, Barrett's algorithm and Montgomery's algorithm (Bosselaers A. et al., 1994). Unlike in the classical algorithm in Barrett's and Montgomery's algorithms expensive divisions are replaced with less-expensive operations. We will describe the all three algorithms, but we implemented only the classical algorithm and Montgomery's algorithm and measured their running time.

We will assume that $b = 2^w$, where $w = 8, 16$ or 32. Further, X and M are two integers such that $X \geq M$. We will express these integers in radix b notation as follows:

$$X = \sum_{i=0}^{l-1} X[i]b^i \text{, where } 0 < X[l-1] < b \text{ and } 0 \leq X[i] < b \text{ for } i = 0,1,\ldots,l-2,$$

$$M = \sum_{i=0}^{k-1} M[i]b^i \text{, where } 0 < M[k-1] < b \text{ and } 0 \leq M[i] < b \text{ for } i = 0,1,\ldots,k-2.$$

Thus our problem is to compute X mod M.

5.4.2 The Classical Reduction Algorithm

The classical algorithm (Knuth D. E.) imitates the well-known ordinary pencil-and-paper method. At each step we divide (k+1)-digit number Z by k-digit modulus M obtaining the one-digit quotient Q and k-digit remainder R. Since $R < M$, we set $Z = Rb + (\text{next digit of } X)$ as the next step of the algorithm. We repeat these operations $l - k$ times. To estimate Q as accurately as possible we normalize M by shifting as many bits to the left as necessary to make the most significant bit of M equal to 1, which ensures

that $M[k-1] \geq \lfloor b/2 \rfloor$. Hence by dividing the most significant digits of Z by $M[k-1]$ we obtain the quotient Q that may be smaller than its correct value at most two. At the end we obtain the correct value of the remainder by shifting it to the right the same number of bits as M was shifted to the left during normalization. The algorithm is given below:

Classical Reduction Algorithm

INPUT: b, $X = (X[l-1] \ldots X[1]\ X[0])$, $M = (M[k-1] \ldots M[1]\ B[0])$.

OUTPUT: X mod M.

Step 1. $R = X$

Step 2. If $R > Mb^{l-k}$ then $R = R - Mb^{l-k}$

Step 3. for $i = l - 1$ to k by -1

Step 4. if $R[i] = M[k-1]$ then $Q = b - 1$ else $Q = (R[i]b + R[i-1])/M[k-1]$

Step 5. while $Q(M[k-1]b + M[k-2]) > R[i]b^2 + R[i-1]b + R[i-2]$

Step 6. $Q = Q - 1$

Step 7. $R = R - QMb^{i-k}$

Step 8. if $R < 0$ then $R = R + QMb^{i-k}$

Step 9. return R

5.4.3 Barrett's Reduction Algorithm

In the Barrett's algorithm (Bosselaers A. et al., 1994) the quotient X/M is estimated using the less-expensive multiplication operation and division by b^t for some t that depends only on the modulus M. When b is a power of 2 the division by b^t is performed efficiently in general-purpose computers by shifting t bits to the right. The Barrett's algorithm requires an expensive modulus dependent precalculation $\mu = \lfloor b^{2k}/M \rfloor$ and hence is applicable if we perform many reductions using a single modulus for a fixed base. However, for a fixed modulus M and base b there is a restriction on X, i.e., X must be less that b^{2k}. The algorithm is outlined bellow:

Barrett's Reduction Algorithm

INPUT: b, $X = (X[l-1] \ldots X[1]\ X[0])$, $M = (M[k-1] \ldots M[1]\ M[0])$.

PRECOMPUTATION: $\mu = \lfloor b^{2k} / M \rfloor$

OUTPUT: $X \bmod M$.

Step 1. $R = X$

Step 2. $Q = \left(\left(R / b^{k-1} \right) \mu \right) / b^{k+1}$

Step 3. $R = R \bmod b^{k+1} - (QM) \bmod b^{k+1}$

Step 4. if $R < 0$ then $R = R + b^{k+1}$

Step 5. while $R \geq M$

Step 6. $R = R - M$

Step 7. return R

5.4.4 Montgomery's Reduction Algorithm

Montgomery's reduction algorithm reduces X modulo M by replacing expensive division by M with a multiplication operation followed by division by a power of b (Bosselaers A. et al., 1994). This is performed using the Montgomery's reduction method (Montgomery P. L).

Let $R = b^k$ be an integer greater than M such that $\gcd(R, M) = 1$ meaning that M must be an odd integer. The M-residue of an integer $Y < M$ with respect to R is defined as $(YR) \bmod M$. We will denote the M-residue of an integer $Y < M$ by $\hat{Y}$. The Montgomery reduction of an integer Y is defined as $(YR^{-1}) \bmod M$, where R^{-1} is the inverse of B modulo M, which is the inverse operation of the M-residue transformation.

The basic idea of Montgomery's reduction is to make X divisible by R by adding multiples of M. Let $M' = -M^{-1} \bmod R$. Then for any integer Y, $(Y + TM)/R$ is an integer congruent to YR^{-1} modulo M, where $T = (M'X) \bmod R$. This property allows us to perform a Montgomery reduction $YR^{-1} \bmod M$ for integers $0 \leq Y < RM$ in the same time as

multiplication. The reduction process proceeds word by word hence we perform this operation by computing one word T_i at a time to and adding $T_i M b^i$ to X that repeats k times. This allows us to compute $M'[0] = M[0]^{-1} \bmod b$ instead of M'. The algorithm is given below:

Montgomery's Reduction Algorithm

INPUT: $b,\ X = X[l-1]\ldots X[1]X[0],\ M = (M[k-1]\ldots M[1]\ M[0])$.

OUTPUT: $X \bmod M$.

PRECOMPUTATION: $M'[0] = M[0]^{-1} \bmod b,\ \hat{X} = XR \bmod M$.

Step 1. $\hat{r} = \hat{X}$

Step 2. for $i = 0$ to $k - 2$

Step 3. $T[i] = (\hat{r}[i]M'[0]) \bmod b$

Step 4. $\hat{r} = \hat{X} + T[i]Mb^i$

Step 5. $\hat{r} = \hat{X} / b^k$

Step 6. if $\hat{r} \geq M$ then $\hat{r} = \hat{r} - M$

POSTCOMPUTATION: $r = \hat{r}R^{-1} \bmod M$

return r

5.4.5 Implementation Results

The classical reduction algorithm and Montgomery's reduction algorithm was implemented in C and on 700 MHz Pentium II computer running Windows 2000 operating system with 128 megabytes of memory. To measure execution times we fixed a 5-word prime modulus and generated random numbers of length between 5 and 9 words. For each word length we run codes 1000 times. The execution times of the routines of the classical reduction and Montgomery's reduction algorithms are tabulated in Table 5.1 in microseconds.

Table 5.1 Execution times in microseconds for reduction algorithms.

Length of X (in words)	The Classical Algorithm	Montgomery's Algorithm	$\frac{\text{Classical}}{\text{Montgomery's}}$ %
5	0.31	0.92	33.7
6	1	0.81	123.4
7	1.21	0.94	128.7
8	1.64	1	164
9	2.1	1.14	184.2

For each word length the relative ratio between the classical reduction algorithm and Montgomery's reduction algorithm was obtained. As can be seen from Table 5.1, for 5-word integers Montgomery's reduction algorithm demonstrates poor performance comparing to the classical, however. When the length of X is equal to the length of the modulus the classical algorithm demonstrates better performance since *for* loop in Step 3 of the classical reduction algorithm is skipped. As the length of X's increases the Montgomery's algorithm runs at least 1.2 times faster than the classical reduction algorithm.

5.5 MODULAR MULTIPLICATION

Modular multiplication as well as the modular reduction operation has significant impact on performance of implementation of cryptographic algorithms. There are several algorithms to perform modular multiplication operation. The simplest way to implement modular multiplication for any elements A and B of $GF(p)$ is to compute the product AB and reduce the product modulo p. The Montgomery multiplication algorithm is more suitable when one performs several modular multiplication operations with respect to the same modulus. For example, the RSA algorithm and the Diffie-Hellman Key Exchange scheme require modular exponentiation operation. The modular exponentiation algorithms perform modular reduction and squaring operations at each step of exponentiation. In particular, Montgomery's multiplication algorithm is suitable for implementation on general-purpose computers. We will consider two algorithms: standard modular multiplication algorithm and the Montgomery's multiplication algorithm.

The following algorithm (Knuth D. E.), based on the separated operand scanning method, performs integer multiplication of A and B followed by reduction modulo M operation.

Standard Modular Multiplication Algorithm

INPUT: $A = (A[s-1]\ldots A[1]\ A[0]),\ B = (B\ [s-1]\ldots B\ [1]\ B[0]),\ M$.

OUTPUT: $AB \bmod M$.

Step 1. for $i = 0$ to $s-1$

Step 2. $\varepsilon = 0$;

Step 3. $(\varepsilon, S) = C[i+j] + A[j]B[i] + \varepsilon$

Step 4. $C[i+j] = S$

Step 5. $C[i+s] = \varepsilon$

Step 6. return $C \bmod M$

Montgomery's multiplication algorithm is based on the Montgomery product. The Montgomery product of integers X and Y less that M is $(XYR^{-1}) \bmod M$, where R is an integer relatively prime to M. Since Montgomery's multiplication algorithm performs division by powers of 2, in practice R is chosen to be a power of 2. The division by powers of 2 can be implemented efficiently on general-purpose computers by shifting to the right necessary number of bits of a dividend.

Formally, let M be a prime modulus and k is the minimum number of words needed to represent M in radix $b = 2^w$ notation, where $w = 8, 16 \text{ or } 32$. Further, assume that $R = b^k$ and X and Y are integers less that M. The Montgomery product of M-residues is defined as $\hat{Z} = (\hat{X}\hat{Y}R^{-1}) \bmod M$, where R^{-1} is the inverse of B modulo M. In its general form the Montgomery multiplication algorithm takes two M-residues $\hat{X}$ and $\hat{Y}$, computes the product $\hat{X}\hat{Y}$ and performs the Montgomery reduction $(\hat{X}\hat{Y}R^{-1}) \bmod M$. The result $\hat{Z}$ is the M-residue of the product XY.

There are many ways to implement the Montgomery multiplication algorithm (Koç Ç. K. et al., 1996). The coarsely integrated operand scanning method comparing to other methods demonstrates better performance in terms of time and space requirements. In this method the multiplication $\hat{X}\hat{Y}$ and reduction $(\hat{X}\hat{Y}R^{-1}) \bmod M$ are performed simultaneously that results in the better performance. The reduction operation is performed in the same manner as in the Montgomery's reduction algorithm, i.e., we precompute $M'[0]$ instead of M' and process one word at a time. The Montgomery multiplication algorithm is outlined below.

The Montgomery Multiplication Algorithm
(coarsely integrated operand scanning method)

INPUT: $\hat{X} = \hat{X}[s-1] \ldots \hat{X}[1]\hat{X}[0]$, $\hat{Y} = \hat{Y}[s-1] \ldots \hat{Y}[1]\hat{Y}[0]$, M, b.

OUTPUT: $(\hat{X}\hat{Y}R^{-1}) \bmod M$.

Step 1. for $i = 0$ to $s-1$
Step 2. $\varepsilon = 0$
Step 3. for $j = 0$ to $s-1$
Step 4. $(\varepsilon, S) = T[i+j] + \hat{X}[j]\hat{Y}[i] + \varepsilon$
Step 5. $T[i] = S$
Step 6. $(\varepsilon, S) = T[s] + \varepsilon$
Step 7. $T[s] = S$
Step 8. $T[s+1] = \varepsilon$
Step 9. $\varepsilon = 0$
Step 10. $m = T[0]M'[0] \bmod b$
Step 11. $(\varepsilon, S) = T[0] + mM[0]$
Step 12. for $j = 0$ to $s-1$
Step 13. $(\varepsilon, S) = T[j] + mM[j] + \varepsilon$
Step 14. $T[j-1] = S$
Step 15. $(\varepsilon, S) = T[s] + \varepsilon$
Step 16. $T[s-1] = S$
Step 17. $T[s] = T[s+1] + \varepsilon$
Step 19. return T

The Montgomery multiplication algorithm is used to compute a modular multiplication of two integers X and Y with respect to a prime modulus as follows. We first perform M-residues transformations $\hat{X}$ and $\hat{Y}$ by multiplying with R, i.e., $\hat{X} = XR \bmod M$ and $\hat{Y} = \hat{Y}R \bmod M$. Then we compute $\hat{Z} = (\hat{X}\hat{Y}R^{-1}) \bmod M$ using the Montgomery multiplication algorithm. Finally, to change the residue domain we compute $Z = \hat{Z}R^{-1} \bmod M$. However, for a single modular multiplication the Montgomery multiplication algorithm should not be used because of relatively expensive M-residue transformations. To measure efficiency of both standard modular multiplication and the Montgomery multiplication algorithm we implemented the exponentiation algorithms based on these algorithms and run 5 times on 700 MHz Pentium II each time generating 1000 numbers. As a result we determined average-running times as 55.81 microseconds for Montgomery's multiplication and 122.83 microseconds for the classical algorithm, i.e., Montgomery's multiplication algorithm is at least two times faster than the classical algorithm.

5.6 MODULAR EXPONENTIATION

The simplest way to compute the modular exponentiation $A^e \bmod M$ is to multiply A by itself e times (Knuth D. E.). At each step of multiplication we reduce the product modulo M and so the whole process requires $e-1$ multiplications and $e-1$ reductions modulo M. For example, to compute $A^7 \bmod M$ we proceed as following:

Step 1. $C = A^2 \bmod M$

Step 2. $C = (CA) \bmod M = A^3 \bmod M$

Step 3. $C = (CA) \bmod M = A^4 \bmod M$

Step 4. $C = (CA) \bmod M = A^5 \bmod M$

Step 5. $C = (CA) \bmod M = A^6 \bmod M$

Step 4. $C = (CA) \bmod M = A^7 \bmod M$

This method computes all powers of A less that e to find A^e and hence is not applicable for large exponents. The Montgomery multiplication algorithm performs the modular multiplication without expensive division by the modulus and so is more suitable whenever several multiplication operations are required. The exponentiation operation can be performed faster by replacing the ordinary multiplication followed by reduction with the Montgomery multiplication. There are several methods that improve the exponentiation algorithm. The next section will focus on these algorithms. We will fix the multiplication algorithm to the Montgomery multiplication algorithm and describe binary, *m*-ary and recoding methods. We will denote the Montgomery product of A and B by *MonProc*(A, B).

5.6.2 The Binary Method

The binary method performs exponentiation by scanning bits of the exponent starting from the most significant bit to the lowest significant bit (Knuth D. E.). Let $f(A,e)$ be a function that computes A^e. Then $f(A,e)$ is defined recursively as $f(A,0)=1$, $f(A,e)=\left(A^{e/2}\right)^2$ if e is even and $f(A,e)=\left(A^{(e-1)/2}\right)^2 A$ if A is odd, i.e., subsequent multiplication by A is performed if the scanned bit is different form zero. Thus we obtain the exponentiation algorithm based on the previous definition.

Exponentiation Algorithm (binary method)

INPUT: A, M, $e=(e[k-1]\ldots e[1]\ e[0])$.

OUTPUT: $A^e \bmod M$.

PRECOMPUATTION: $\hat{A}=AR \bmod M$ (M-residue of A).

Step 1. if $e[k-1]=1$ then $\hat{C}=\hat{A}$ else $\hat{C}=\hat{1}$

Step 2. for $i=k-2$ to 0

Step 3. $\hat{C}=MonProc(\hat{C},\hat{C})$.

Step 4. if $e[i]=1$ then $\hat{C}=MonProc(\hat{C},\hat{A})$

POSTCOMPUTATION: $C=\hat{C}R^{-1} \bmod M$

Step 6. return C

For example, let $e = 53_{10} = (110101)_2$. Then $k = 6$ and so we start with $\hat{C} = \hat{A}$ since $e[5] = 1$. We compute $A^{37} \bmod M$ using the binary method as follows.

i	e_i	Step 3	Step 4	Computed value
4	1	$\hat{C} = MonProc(\hat{C}, \hat{C})$	$\hat{C} = MonProc(\hat{C}, \hat{A})$	$A^3 \bmod M$
3	0	$\hat{C} = MonProc(\hat{C}, \hat{C})$	skipped	$A^6 \bmod M$
2	1	$\hat{C} = MonProc(\hat{C}, \hat{C})$	$\hat{C} = MonProc(\hat{C}, \hat{A})$	$A^{13} \bmod M$
1	0	$\hat{C} = MonProc(\hat{C}, \hat{C})$	skipped	$A^{26} \bmod M$
0	1	$\hat{C} = MonProc(\hat{C}, \hat{C})$	$\hat{C} = MonProc(\hat{C}, \hat{A})$	$A^{53} \bmod M$

5.6.3 The *m*-ary Method

The *m*-ary method is regarded as a generalization of the binary method (Koç Ç. K., 1994). Instead of scanning one bit of the exponent at a time in the *m*-ary method we scan $\log_2 m$ bits at a time. Assume that *m* is a power of 2 and $r = \log_2 m$. The exponent *e* can be written using base *m* notation as $e = d_0 + d_1 m + \ldots + d_{n-1} m^{n-1}$. This representation allows us compute A^e as a product $A^{d_0} A^{d_1} \cdots A^{d_{n-1}}$. As an example, let $e = 1231_{10} = (10011001111)_2$ and $m = 16$. Since $\log_2 16 = 4$ we partition bits of the exponent into 4-bit groups obtaining base-16 representation $15 + 12(16) + 4(16^2) = 1231$. Thus we have $A^{1231} = A^{15} \left(A^{12}\right)^{16} \left(A^4\right)^{256}$. The exponentiation algorithm based on *m*-ary method is outlined below:

Exponentiation Algorithm (*m*-ary method)

INPUT: $A, M, m = 2^r, e$.

OUTPUT: $A^e \bmod M$.

PRECOMPUATTION: $\hat{A} = AR \bmod M$. Compute and store $\hat{A}^n$ for $n = 1, \ldots, m-1$``.
Compute $(d[0]\, d[1] \ldots d[s-1])_m$, base-*m* representation of *e*.

Step 1. $\hat{C} = \hat{A}^{d[s-1]}$

Step 2. for $i = s - 2$ to 0

Step 3. $\hat{C} = \hat{C}^m$

Step 4. if $d[i] \neq 0$ then $\hat{C} = MonProc(\hat{C}, \hat{A}^{d[i]})$

POSTCOMPUTATION: $C = \hat{C}R^{-1} \bmod M$

Step 6. return C

5.6.4 The *m*-ary Recoding Method

The recoding method converts the exponent to a nonstandard representation using signed digits 1, 0 and –1 (–1 will be denoted by $\bar{1}$). By converting binary representation of exponent to the signed-digit representation this technique reduces the number of 1s in order to obtain sparse representation of the exponent. The recoding technique exploits the following identity

$$2^{n+m-1} + 2^{n+m-2} + \cdots + 2^{n} = 2^{n+m} - 2^{n}$$

to obtain a signed-digit representation of an exponent. For example, let $e = 243 = (1011101)_2$. The signed-digit representation of this number is $(10\bar{1}00\bar{1}01)$, i.e., $(10\bar{1}00\bar{1}01) = 2^7 - 2^5 - 2^2 + 1 = 243$. Hence A^{243} is equal to the product $A^{128}A^{-32}A^{-4}A$. Once signed-digit representation of an exponent has been obtained, *m*-ary method can be applied. Additionally, in this method we need to precompute and store values $\hat{A}^{-n}$ for $n = 1, \ldots, m-1$. The *m*-ary recoding method is given below:

Exponentiation Algorithm (*m*-ary recoding method)

INPUT: A, M, $m = 2^r$, e.

OUTPUT: $A^e \bmod M$.

PRECOMPUATTION: $\hat{A} = AR \bmod M$. Compute and store $\hat{A}^n$ and $\hat{A}^{-n}$ for $n = 1, \ldots, m-1$. Compute $(\bar{d}[0]\,\bar{d}[1] \ldots \bar{d}[s-1])_m$, base-*m* signed-digit representation of *e*.

Step 1. $\hat{C} = \hat{A}^{\bar{d}[s-1]}$

Step 2. for $i = s - 2$ to 0

Step 3. $\hat{C} = \hat{C}^m$

Step 4. if $\bar{d}[i] = 1$ then $\hat{C} = MonProc(\hat{C}, \hat{A}^{\bar{d}[i]})$
else if $\bar{d}[i] = -1$ then $\hat{C} = MonProc(\hat{C}, \hat{A}^{-\bar{d}[i]})$

POSTCOMPUTATION: $C = \hat{C}R^{-1} \bmod M$

Step 6. return C

To convert an exponent e to a signed-digit representation we use the canonical recoding algorithm. A signed-digit representation is said to be canonical if it contains no adjacent non-zero bits. Let $e = (e_0 e_1 \ldots e_{k-1})$ be an ordinary binary representation of the exponent. The canonical recoding algorithm converts the exponent to a signed-digit representation by scanning two bits at a time starting from the least significant bit. The algorithm uses an auxiliary variable c. Initially $c_0 = 0$. The signed digits are computed according to the following truth table:

c_i	e_i	e_{i+1}	c_{i+1}	f_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	$\overline{1}$
1	0	0	0	1
1	0	1	1	0
1	1	0	1	$\overline{1}$
1	1	1	1	0

For example, let $e = 90 = (1011010)_2$. The signed-digit representation of e is $(10\overline{1}0\overline{1}010) = 2^7 - 2^5 - 2^3 + 2 = 90$.

5.6.5 Implementation Results

We implemented the modular exponentiation algorithm based on the classical modular exponentiation algorithm (by repeated multiplication and reduction operations) and the Montgomery multiplication algorithm. For each algorithm we implemented the straightforward *m*-ary method and recoding *m*-ary method. The performance of the C codes was measured on a 700 MHz Pentium II computer running Windows 2000 operating system with 128 megabytes of memory.

We fixed 5, 8, 16 and a 32-word moduli and randomly generated 5, 8, 16, and 32 word bases and exponents respectively. For each single modulus the codes run 1000 times and execution times for *m* equal to 2, 4, 16 and 256 were measured. Precomputation timings were measured separately and are displayed in Table 5.2.

Table 5.2 Precomputation times in microseconds for the *m*-ary method.

Length of modulus (in words)	Straightforward *m*-ary			Recoding *m*-ary		
	m = 4	*m* = 16	*m* = 256	*m* = 4	*m* = 16	*m* = 256
5	1352.1	1349.6	1580.7	2962.8	2939.8	2906.7
8	1348.7	1976.7	1910.6	3032.4	3207.3	4161.6
16	1353.2	3243.4	3184	3581.6	3790.1	7296.3
32	1437.5	8556.1	8250.4	5661.4	6492.6	19349.4

The execution times of exponentiation operation for different values of *m* are tabulated in the following table:

Table 5.3 Execution times in microseconds for the straightforward *m*-ary algorithm for different values of *m*.

Length of modulus (in words)	*m* = 2	*m* = 4	*m* = 16	*m* = 256
5	262.46	238.34	211.49	187.81
8	930	856.41	844.12	641.98
16	5603.38	5127.4	4589.71	4137.9
32	40659.22	37239	33352.96	30251.68

Table 5.3 shows that as the value of *m* increases, the straightforward *m*-ary algorithm runs faster. To compare the *m*-ary recoding method with the straightforward *m*-ary algorithm we fixed the value of *m* to 4, 16 and 256 and run the code for fixed moduli of length 5, 8, 16 and 32 words. For each modulus we generated 1000 random bases and exponents of length respectively 5, 8, 16 and 32 words. Execution times are given in Table 5.4.

Table 5.4 Execution times in microseconds for the straightforward *m*-ary and the recoding *m*-ary algorithms.

Length of modulus (in words)	Straightforward *m*-ary			Recoding *m*-ary		
	m = 4	*m* = 16	*m* = 256	*m* = 4	*m* = 16	*m* = 256
5	288.39	239.64	247.43	290.16	252.3	260.88
8	1024.3	774.84	732.04	906.81	777.2	728.78
16	58030.1	5155	4640.3	5629.3	5100.4	4647.9
32	40655	36675	33955	39308	36285	34087

From Table 5.4 we can see that the straightforward *m*-ary algorithm and the recoding *m*-ary algorithm differ only slightly. Moreover, when the length of the modulus is 32 words and *m* is equal to 256, the straightforward *m*-ary algorithm runs faster than the recoding *m*-ary algorithm. The utmost time gained by the use of the *m*-ary recoding algorithm is observed when *m* is equal to 2, i.e., for the binary recoding method. The execution times for the straightforward binary algorithm and the recoding binary algorithm are given below in Table 5.5.

Table 5.5 Execution times in microseconds for the straightforward binary and the recoding binary algorithms.

Length of modulus (in bits)	Straightforward binary	Recoding binary	Straightforward / Recoding
160	294.61	246.81	1.19
192	473.28	409.87	1.15
224	699.42	615.76	1.14
256	910.93	798.25	1.14
512	5918.19	5198.26	1.14
1024	42508.33	37613.74	1.13
2048	324835.51	288665.46	1.13

Since the both straightforward and recoding *m*-ary algorithms require relatively expensive precomputation it is not a good idea to use them in cryptographic algorithms such as RSA. The time needed to perform precomputation may exceed the time gained by the use the straightforward or recoding *m*-ary algorithms.

5.7 INVERSION

The modular inversion operation is one of the basic operations on prime fields that are quite expensive at least as multiplication operation so its implementation will have considerable impact on the performance of cryptographic algorithms. In this section we will describe several inversion algorithms all of which are based on the Extended Euclidean algorithm and on the concept of the Montgomery modular inverse.

5.7.2 The Extended Euclidean Algorithm

The Extended Euclidean Algorithm rests on the well-known Euclidean Algorithm that was invented originally to compute the greatest common divisor of two integers A and B (Knuth D. E.). We will denote the greatest common divisor of A and B by $\gcd(A, B)$. This algorithm exploits the simple property of integers, i.e. for any integers A and B $\gcd(A, B) = \gcd(B, R)$, where R is the least positive remainder of the division A by B, and $\gcd(A, B) = B$ if A is divisible by B. This algorithm can be modified to find two integers x and y for any integers A and B so that $\gcd(A, B) = xA + yB$. This property can be used easily to compute a^{-1} for an element a of $GF(p)$ since $\gcd(a, p) = 1 = xa + yp$ means that $xa \equiv 1 \bmod p$ and hence x is the inverse of a. Given A and B the Extended Euclidean Algorithm computes integers x and y such that $\gcd(A, B) = xA + yB$:

The Extended Euclidean Algorithm

INPUT: A, B.

OUTPUT: $d = \gcd(A, B)$, x, y

Step 1. $u = A,\ v = B,\ x_1 = 1, y_1 = 0, x_2 = 0, y_2 = 1$

Step 2. while $u \neq 0$

Step 3. $q = \lfloor v/u \rfloor,\ r = v - qu,\ x = x_2 - qx_1,\ y = y_2 - qy_1$

Step 4. $v = u,\ u = r,\ x_2 = x_1,\ x_1 = x,\ y_2 = y_1,\ y_1 - y$

Step 6. $d = v,\ x = x_2,\ y = y_2$

Step 7. return (d, x, y)

The Extended Euclidean Algorithm requires computationally expensive division operations at each step of the loop and hence is not practical in cryptographic applications. To avoid this drawback the binary inversion algorithm exploits the following simple property of integer numbers. For any integers A and B, if A is even then $\gcd(A, B)$ is equal to $\gcd(A/2, B)$ (Jebelean T., 1993). Division by two is intrinsically fast operation on general-purpose computers, so the binary inversion algorithm is more applicable to compute modular inversion.

The Binary Inversion Algorithm

INPUT: A, p such that $0 < A < p$.

OUTPUT: $A^{-1} \bmod p$

Step 1. $u = A$, $v = p$, $t_1 = 1$, $t_2 = 0$

Step 2. while $u \neq 1$ and $v \neq 1$

Step 3. while u is even

Step 4. $u = u/2$

Step 6. if t_1 is even then $t_1 = t_1/2$ else $t_1 = (t_1 + p)/2$

Step 7. while v is even

Step 8. $v = v/2$

Step 9. if t_2 is even then $t_2 = t_2/2$ else $t_2 = (t_2 + p)/2$

Step 10. if $u \geq v$ then $\{u = u - v,\ t_1 = t_1 - t_2\}$ else $\{v = v - u,\ t_2 = t_2 - t_1\}$

Step 11. if $u = 1$ then return $t_1 \bmod p$ else return $t_2 \bmod p$

5.7.2 The Montgomery Inversion

The Montgomery inversion, introduced by Kaliski in 1995, is a relatively new approach to compute the modular inversion. Let p be a prime integer and A be an integer such that $1 \leq A \leq p-1$. The Montgomery inverse of the integer A is defined as $A^{-1}2^n \bmod p$, where $n = \lceil \log_2 p \rceil$. The following algorithm computes the Montgomery inverse. This algorithm consists of two phases. The first phase given a prime integer p and A such that $1 \leq A \leq p-1$ computes $r = A^{-1}2^k \bmod p$, where $n \leq k \leq 2n$. The second phase then completes the computation by correcting the output from phase 1, i.e., computes $A^{-1}2^n \bmod p$. The Montgomery Inverse algorithm is given below:

The Montgomery Inverse Algorithm (phase 1)

INPUT: A, p such that $0 < A < p$.

OUTPUT: $A^{-1}2^k \bmod p$ and k, where $n \le k \le 2n$

Step 1. $u = p,\ v = A,\ r = 0,\ s = 1,\ k = 0$

Step 2. while $v > 0$

Step 3. if u is even then $\{u = u/2,\ s = 2s\}$

Step 4. else if v is even then $\{v = v/2,\ r = 2r\}$

Step 6. else if $u > v$ then $\{u = (u - v)/2,\ r = r + s,\ s = 2s\}$

Step 7. else if $v \ge u$ then $v = (v - u)/2,\ s = s + r,\ r = 2r\}$

Step 8. $k = k + 1$

Step 9. if $r \ge p$ then $r = p - r$

Step 10. $r = p - r$

Step 11. return (r, k)

The Montgomery Inverse Algorithm (phase 2)

INPUT: (r, k) from phase 1

OUTPUT: $A^{-1}2^n \bmod p$

Step 1. for i = 1 to $k - n$

Step 2. if r is even then $r = 2r$ else $r = (r + p)/2$

Step 3. return r

The above-described algorithm performs bit level operations and hence is not suitable for software implementation on a general-purpose computer. Savaş E. and Koç Ç. K. made an additional change so that the Montgomery inverse algorithm could be used. The basic idea is to replace the bit level operations in phase 2 with word level operations by introducing a new Montgomery radix $R = 2^m$ instead of 2^n such that m is an integer multiple of the wordsize. To achieve the best performance m is selected to be equal to iw such that $(i-1)w < n \le iw = m$ for a positive integer i where w is the wordsize. The new version of phase 2 is obtained by adding two Montgomery multiplication operations such

that for an integer A and a prime modulus p it computes $A^{-1}2^m \bmod p$. As a result the new version of phase 2 computes the inverse of A and converts it to p-residue with respect to R. The modified version of the Montgomery Inverse algorithm is given below.

The Modified Montgomery Inverse Algorithm

INPUT: A, p, n, m such that $1 \le A \le 2^m - 1$

OUTPUT: $A^{-1}2^m \bmod p$

Step 1. Perform phase 1 and obtain (r, k), $r = A^{-1}2^k \bmod p$ and $n \le k \le n + m$

Step 2. if $n \le k \le m$ then $\{r = MonProc(r,\ 2^{2m}),\ k = k + m\}$

Step 3. $r = MonProc(r,\ 2^{2m-k})$

Step 4 return r

Another modification of the Montgomery Inverse algorithm introduced in performs classical inverse operation, i.e., it computes $A^{-1} \bmod p$ for an integer $1 \le A \le p - 1$ and prime modulus p. Phase 2 are modified so that the output $r = A^{-1}2^k \bmod p$ from phase 2 is transformed to $A^{-1} \bmod p$ with at most two Montgomery multiplication operations.

The Classical Inverse Algorithm (based on the Montgomery Inverse)

INPUT: A, p, n, m such that $1 \le A \le 2^m - 1$

OUTPUT: $A^{-1} \bmod p$

Step 1. Perform phase 1 and obtain (r, k), $r = A^{-1}2^k \bmod p$ and $n \le k \le n + m$

Step 2. if $k > m$ then $\{r = MonProc(r, 1),\ k = k - m\}$

Step 3. $r = MonProc(r,\ 2^{m-k})$

Step 4 return r

We implemented the inverse algorithm in C on a 700 MHz Pentium II computer and compared its running time with the running time of the Montgomery multiplication algorithm. The result is tabulated in Table 5.6 below.

Table 5.6 Running times in microseconds for the Montgomery multiplication algorithm and the classical inverse algorithm.

Length of modulus (in bits)	Montgomery Multiplication	Montgomery Inversion
160	1.47	114.87
192	1.65	128.70
224	2.00	158.02
256	2.67	190.99
512	7.92	534.30
1024	32.13	1923.31

From Table 5.6 it us seen that the Montgomery multiplication is several times faster than the inversion operation based on the Montgomery inversion.

CHAPTER 6

THE ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM

The Elliptic Curve Digital Signature Algorithm is the elliptic curve based digital signature algorithm proposed in 1992 by Scott Vanstone. Like the Digital Signature Algorithm (DSA) ECDSA provides key generation, signature and signature verification schemes. The ECDSA is the most popular digital signature algorithm. The International Standard Organization accepted it in 1998 as an ISO standard. It was also accepted in 1999 as an ANSI standard and in 2000 as NIST and IEEE standards. The main advantage of the ECDSA over DSA is that for the same key-length the security of an algorithm using the ECDSA is considerably greater than that using DSA.

Assume that $E(F_p)$ be an elliptic curve over a finite field F_p and P be point of prime order N in $E(F_p)$ such that the length of N is at least 192 bits. Usually, P is chosen as a generator of the subgroup $\langle P \rangle$ and $(p, F_p, E(F_p), P, N)$ is referred as domain parameters.

- **Setup scheme**

  1. Choose a random integer x in the interval $[1, N-1]$ and keep it secret;
  2. Compute $Q = xP$. Q is the public key.

- **Signing scheme**

  1. Apply a hash function H to a message m: $0 < H(m) < N$;

2. Select a random integer k in the interval $[1, N-1]$ and compute $kP = (x_1, y_1)$;
3. Set $r = x_1 \bmod N$;
4. Find an integer s such that $sk \equiv (H(m) + xr) \bmod N$;
5. If s is zero then return to the step 2.
6. Set the signature as a pair (r, s).

- **Verification scheme**

1. Verify that r and s are in the interval $[1, N-1]$;
2. Compute $u_1 = s^{-1}H(m) \bmod N$ and $u_2 = s^{-1}r \bmod N$;
3. Compute $(x_0, y_0) = u_1P + u_2Q$ and set $v = x_0 \bmod N$;
4. The signature is acceptable if and only if $r = v$.

We have implemented three basic algebraic operations for groups formed by elliptic curves over finite fields: point addition, point inversion and scalar multiplication (Cohen et al, Brown et al., 1998). Later using theses functions the ECDSA algorithm was implemented. The codes were written in C on 700 MHz Pentium II computer. We run and tested the ECDSA algorithm for different elliptic curves. Coefficients of the elliptic curves and related finite fields are given in the appendix. The integer numbers are represented in the base 2^{32} using decimal digits. For each elliptic curve we run codes 500 times. Timing results are depicted in the following table.

Table 6.1 ECDSA key generation, signature generation and signature verification timings in microseconds.

Elliptic curve	Key generation	Signature generation	Signature verification
C1_P160	26,786.3	20,219.6	51,422.2
C2_P160	25,660.8	20,185.3	51,690.2
C1_P192	38,770.0	31,229.2	78,591.9
C2_P192	40,151.7	31,520.7	79,807.0
NIST_P192	42,060.0	34,122.3	79,699.5
C1_P224	57,383.2	46,093.1	114,363.1
C2_P224	55,952.5	46,224.5	107,718.6
NIST_P224	60,283.6	50,011.2	114,821.2
C1_P256	78,250.9	64,737.4	156,184.6
C2_P256	79,502.6	67,353.6	157,508.1
NIST_P256	81,855.4	69,323.6	156,693.5
C1_P384	212,485.8	189,312.0	423,973.6
C2_P384	214,420.6	190,491.2	425,676.0
NIST_P384	218,328.8	194,239.3	424,140.8
C1_P512	440,689.3	402,314.0	882,829.3
C2_P512	439,276.1	422,765.0	875,224.7
NIST_P512	499,861.9	486,786.5	939,457.1

CHAPTER 7

CONCLUSION

Groups, rings and fields are fundamental mathematical structures that lay the foundation for cryptography. Mathematical properties of these structures give rise to computationally hard problems that allow us to design one-way functions exploited by cryptographic algorithms. In this thesis we considered finite fields with prime number of elements called as prime fields. A prime field can be represented as a finite set of integer numbers together with modular addition operation and modular multiplication operation. We also considered elliptic curves over prime fields. Elliptic curves are a rich source of finite abelian groups on which elliptic curve cryptography is based.

We implemented multiprecision subtraction, addition, reduction, modular multiplication and inversion algorithms for prime field arithmetic aimed to work on general-purpose computers. For some prime field operations we considered and implemented multiple algorithms. We measured the running times and compared them. Montgomery's theorem gives us a cheaper way to multiply two integers and reduce the product by a modulus. Since the modular exponentiation is the most frequently used arithmetic operation we considered several methods and implemented multiple algorithms for modular exponentiation. We implemented two different modular exponentiation algorithms: the classical algorithm and the algorithm based on the Montgomery product. As a result we observed that the Montgomery's algorithm runs two times faster than the classical algorithm.

Finally, using the prime field arithmetic procedures we implemented elliptic curve point arithmetic and elliptic curve digital signature algorithm for different elliptic curves.

The future research can be conducted in extending the library by developing procedures that perform finite field operation for arbitrary finite field $GF(p^k)$.

APPENDIX

C1_P160	
p	1542587393, 3879432752, 3310093403, 3575984732, 3144840534
a	3719205881, 2456924538, 3061096603, 614799683, 2461125420
b	105343449, 2450485153, 930660859, 2778888033, 744717828
P_x	1498890981, 2436006830, 2633568011, 2311420911, 1353958562
P_y	1022802176, 3454009759, 4166317347, 3977839083, 586077994
N	2663636817, 2297342991, 2313557241, 2351555231, 449262933

C2_P160	
p	3954591183, 3812703439, 4159284655, 2189842651, 934206511
a	2486329882, 1289626620, 64597136, 1632893365, 222300710
b	2425844884, 1180614102, 106200426, 1897439989, 797418535
P_x	1441215752, 473236524, 3447803080, 3566209518, 199039638
P_y	2048565661, 2947385441, 1026415668, 3029059881, 75227164
N	498012111, 2725972077, 4159329146, 2189842651, 934206511

C1_P192	
p	2175053513, 3332204610, 2106319598, 1393860827, 789161253, 2463096058
a	901808635, 3566305214, 1179265647, 179497202, 1170456174, 2460392958
b	2175053513, 3332204610, 2106319598, 1393860827, 789161253, 2463096058
P_x	2858530508, 2465081945, 1490809915, 759538952, 2029996948, 140485898
P_y	555206940, 185421212, 3205774888, 3150824198, 1777417706, 728489572
N	99459063, 1970596714, 4260314513, 1393860828, 789161253, 2463096058

C2_P192	
p	1250306691, 396056278, 2601505881, 534154407, 1362483061, 1595199156
a	867616414, 2027359134, 2896682443, 849552357, 1498913883, 673749683
b	1496723405, 1206459768, 490775308, 2987648780, 2845854059, 61863010
P_x	1157760553, 2161967765, 3036161656, 1729998116, 3999894102, 1043440918
P_y	1461775492, 2161024665, 472558623, 4174651643, 3052845635, 1234129591
N	1755539509, 3037058654, 3112302546, 534154408, 1362483061, 1595199156

NIST_P192	
p	4294967295, 4294967295, 4294967294, 4294967295, 4294967295, 4294967295
a	4294967292, 4294967295, 4294967294, 4294967295, 4294967295, 4294967295
b	3242637745, 4273528556, 1914974281, 262662571, 3852239079, 1679885593
P_x	2197753874, 4110355197, 1134659584, 2092900587, 2955972854, 411936782
P_y	511264785, 945728929, 1797574101, 1661997549, 4291353208, 119090069
N	3033671729, 342608305, 2581526582, 4294967295, 4294967295, 4294967295

C1_P224	
p	725552005, 2845423983, 54069172, 2902997345, 4065895842, 99700918, 1083833297
a	3302838848, 1595115968, 3777041922, 3129691098, 1373563308, 3421245063, 758919852
b	950557985, 1758167159, 1579239137, 3012285341, 2522358170, 1237863700, 793386139
P_x	3149598117, 893375299, 2240762443, 565899189, 553926940, 670842904, 698885573
P_y	1062233727, 2395891116, 322779262, 974859008, 2061791796, 4273376244, 58197125
N	870003411, 3126672323, 2829466493, 2903035980, 4065895842, 99700918, 1083833297

C2_P224	
p	702452983, 1214255019, 878986955, 953379981, 3986309538, 2887575373, 19280865
a	2652357196, 713983010, 1958166667, 3761818879, 1809361211, 1036160428, 1172912
b	461301861, 2535421854, 1850481789, 2447024150, 4284983361, 1444648577, 10320826
P_x	1266926621, 606854806, 2830632472, 3160222785, 3096719789, 3424567889, 957145
P_y	2370583092, 1031196835, 1524110759, 3390450571, 1571481350, 3866392616, 5796543
N	4116426827, 72296103, 101811847, 953388186, 3986309538, 2887575373, 19280865

NIST_P224	
p	1, 0, 0, 4294967295, 4294967295, 4294967295, 4294967295
a	4294967294, 4294967295, 4294967295, 4294967294, 4294967295, 4294967295, 4294967295
b	592838580, 655046979, 3619674298, 1346678967, 4114690646, 201634731, 3020229253
P_x	291249441, 875725014, 1455558946, 1241760211, 840143033, 1807007615,

	3071151293
P_y	2231402036, 1154843033, 1510426468, 3443750304, 1277353958, 3052872699, 3174523784
N	549543997, 333261125, 3770216510, 4294907554, 4294967295, 4294967295, 4294967295

C1_P256	
p	2152041647, 1502198436, 2159380012, 3955240668, 758658574, 4061893134, 269290586, 3582153167
a	2729643541, 1576067140, 3838073187, 2791390472, 3502664604, 2026533070, 1584952649, 2270197294
b	4217995504, 1410746001, 3282962965, 2857765506, 278922748, 4149464640, 4036374731, 3556056163
P_x	4234915459, 2172097435, 3318241100, 3499055396, 3818222353, 1668925924, 3936201909, 2400765551
P_y	716011241, 1113159874, 2822839933, 4242178085, 3292471632, 3431550869, 2021862991, 1273769712
N	338982949, 3157267705, 2123805379, 1188268587, 758658576, 4061893134, 269290586, 3582153167

C2_P256	
p	3461386131, 4242372005, 2053711051, 1408372137, 144184269, 2011048294, 3765518954, 1940372971
a	2733318916, 4128013279, 4235588355, 3679528661, 340097121, 1529638355, 1084502465, 1909983868
b	3592068870, 584748469, 3493513667, 2141770279, 3295224935, 4058628643, 3323203622, 1713217801
P_x	2866596095, 3715780580, 4188731262, 3123373688, 3622859170, 643568062, 2792141271, 1401598310
P_y	2015972219, 3663181186, 434385189, 2869461847, 3386271818, 1452570981, 900395116, 90953084
N	405434131, 2056464943, 2350549605, 250106429, 144184269, 2011048294, 3765518954, 1940372971

NIST_P256	
p	4294967295, 4294967295, 4294967295, 0, 0, 0, 1, 4294967295
a	4294967292, 4294967295, 4294967295, 0 ,0, 0,1, 4294967295
b	668098635, 1003371582, 3428036854, 1696401072, 1989707452, 3018571093, 2855965671, 1522939352
P_x	3633889942,4104206661, 770388896, 1996717441, 1671708914, 4173129445, 3777774151, 1796723186
P_y	935285237, 3417718888, 1798397646, 734933847, 2081398294, 2397563722, 4263149467,1340293858

N	4234356049, 4089039554, 2803342980, 3169254061, 4294967295, 4294967295, 0, 4294967295

C1_P384	
p	1260742493, 3392901779, 2193730719, 3723537569, 681067929, 93611493, 179985939, 2496333200, 168694669, 3977525223, 2250575228, 2474516928
a	238238490, 2872847588, 656779843, 4245783527, 3955437378, 3535159137, 413323322, 1285401686, 4161150900, 3098262617, 3664150091, 996677088
b	92047716, 1667411683, 853680317, 667984548, 2571462521, 1828594300, 3791142788, 3522989130, 2985862837, 2538765003, 426818096, 782451406
P_x	262021186, 2754234735, 117230908, 1905469124, 1401772299, 3102284934, 109505224, 1591880724, 1195235673, 520956396, 3086160451, 956062078
P_y	4107469467, 4105654924, 1233646793, 1082681042, 584143295, 3079929238, 4236614262, 2216251071, 3137564930, 2807923781, 121323002, 864129456
N	3583843023, 2374707721, 2002156273, 1539926627, 3713530899, 737823573, 179985940, 2496333200, 168694669, 3977525223, 2250575228, 2474516928

C2_P384	
p	4275951097, 1692013271, 3158261515, 622347869, 2660549086, 3594360467, 3490844980, 3241518266, 2634326526, 1105183272, 3105075704, 1478868053
a	3139533582, 3461094475, 1568296839, 2573473223, 1239709353, 2498383425, 1971300227, 2107243370, 105233091, 2508790830, 1266526216, 560269212
b	1913816297, 3515765246, 3691390483, 206426652, 1803931059, 2232534887, 2473054830, 888194042, 41873928, 2941934736, 2384767085, 447447435
P_x	609197895, 2340424177, 309128596, 79518755, 936407903, 873578323, 2269331456, 477788943, 1218695926, 3183280702, 3398523180, 522374672
P_y	3926215189, 1888196087, 1017661911, 3066925227, 1924220483, 1956324652, 1967053315, 3498886157, 660318433, 3891340706, 1282520720, 1386147240
N	2438832221, 2798175040, 1507470051, 2109842584, 1088443724, 4241433067, 3490844980, 3241518266, 2634326526, 1105183272, 3105075704, 1478868053

NIST_P384	
p	4294967295, 0, 0, 4294967295, 4294967294, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295
a	4294967292, 0, 0, 4294967295, 4294967294, 4294967295, 4294967295, 4294967295, 4294967295,4294967295, 4294967295,4294967295
b	3555470063, 713410797, 2318324125, 3327539597, 1343457114, 51644559, 4269883666, 404593774, 3824692505, 2559444331, 3795773412, 3006345127
P_x	1920338615, 978607672, 3210029420, 1426256477, 2186553912, 1509376480, 2343017368, 1847409506, 4079005044, 2394015518, 3196781879, 2861025826
P_y	2431258207, 2051218812, 494829981, 174109134, 3052452032, 3923390739, 681186428, 4176747965, 2459098153, 1570674879, 2519084143, 907533898
N	3435473267, 3974895978, 1219536762, 1478102450, 4097256927, 3345173889,

	4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295

C1_P512	
p	3972435479, 1764204733, 2549849560, 1616345901, 1996426875, 3427881837, 2404842145, 787873698, 55755720, 2028808543, 642924886, 2329455697, 3045165639, 2225051090, 2006274842, 3866778103
a	1871084847, 20508022, 1841130766, 929619208, 2012571608, 3975931000, 907334725, 3712928878, 3917512500, 4225685886, 1394484908, 1058652400, 2154779461, 3042976492, 3713322755, 893599195
b	3852712955, 3653545592, 1693078807, 4027242815, 1454672309, 57870769, 2822052966, 1870607459, 591569559, 1942760050, 461529855, 2746052680, 389291775, 4007651061, 1193332821, 1966707121
P_x	2846147534, 1549736485, 3323019829, 2973803838, 1892047013, 1738685115, 2684726614, 749073034, 331541902, 3186182302, 2612376961, 4215291505, 687158746, 2656281191, 1838046819, 1901502262
P_y	1653738872, 1124298423, 1270032033, 2035553925, 4201583735, 1819247152, 1478028487, 3576879187, 377927312, 2353534385, 2365404466, 2403215142, 1935145422, 3808106977, 4004581202, 2947442425
N	1323963577, 965565694, 2954545699, 1561608984, 995136077, 732357913, 1131975229, 1369406340, 55755718, 2028808543, 642924886, 2329455697, 3045165639, 2225051090, 2006274842, 3866778103

C2_P512	
p	3847319715, 2359733666, 2651644093, 2790329907, 1732839427, 1016907134, 355751544, 3712907603, 3393536743, 3535442213, 3007243002, 1070178, 1579032206, 826039325, 3822185121, 851731775
a	3646285476, 51780629, 3657926561, 1106095612, 2097135445, 1577454372, 1694630651, 2033044939, 67635647, 1903214236, 528316243, 3091027082, 4028742985, 1245786525, 635102426, 467237148
b	1090822743, 50553298, 2231495586, 3483325804, 2586926729, 4045609094, 1515004131, 2905866206, 3150553724, 3948097860, 2752107447, 4049648414, 2611983071, 2337174672, 793660679, 654932937
P_x	3875295360, 2960081641, 3348231280, 1722665286, 2825220750, 3337889558, 482704092, 973606566, 1362402576, 664748766, 1989332989, 435082357, 2391128292, 1942891384, 1852575030, 239952643
P_y	1873875067, 2883743132, 2468643188, 3441953944, 3931595445, 3567598761, 3776584087, 592261134, 3595502286, 1506793355, 1007070204, 1726836531, 1380963139, 216454803, 3055455908, 259935842
N	670262609, 4119725683, 2596439261, 3393442064, 1919846775, 1655063829, 2447858785, 1356874882, 3393536744, 3535442213, 3007243002, 1070178, 1579032206, 826039325, 3822185121, 851731775

NIST_P521	
p	4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 511
a	4294967292, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 511
b	1800421120, 4014284756, 1026307313, 896786312, 1001504519, 374522045, 3967718267, 1444493649, 2398161377, 3098839441, 2578650611, 2732225115, 3062186222, 2459574688, 2384239135, 2503915873, 81
P_x	3269836134, 4185816625, 2238333595, 860402625, 2734663902, 4263362855, 4024916264, 2706071159, 1800224186, 4163415904, 88061217, 2623832377, 597013570, 2654915430, 67430861, 2240677559, 198
P_y	2681300560, 2294191222, 2725429824, 893153414, 1068304225, 3310401793, 1593058880, 2548986521, 658400812, 397393175, 1469793384, 2566210633, 746396633, 1552572340, 2587607044, 959015544, 280
N	2436391945, 3144660766, 2308720558, 1001769400, 4144604624, 2144076104, 3207566955, 1367771011, 4294967290, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 4294967295, 511

REFERENCES

- Adleman L. M., McCurley K. S. *Open problems in number theoretic complexity, II*. Algorithmic Number Theory, (LNCS 877), 291–322, 1994.

- Anderson R. *Practical RSA trapdoor*. Electronics Letters, 29 (May 27, 1993), 995.

- Bosselaers A., Govaerts R. and Vandewalle J. *Comparison of three modular reduction functions*. Advances in Cryptology-Crypto '93, LNCS 773, 1994, 175-186.

- Brown M., Hankerson D., Lopez J., and Menezes A. *Software Implementation of the NIST Elliptic Curves Over Prime Fields*. In D. Naccache, editor, Topics in cryptology − CT-RSA 2001, volume LNCS 2020, pages 250-265, Berlin, April 2001. Springer-Verlag.

- Cohen H., Miyaji A. and Ono T. *Efficient elliptic curve exponentiation using mixed coordinates*. Advances in Cryptology - Asiacrypt '98, LNCS 1514, 1998, 51-65.

- Crandall R. E. *Method and apparatus for public key exchange in a cryptographic system*. U.S. Patent Number 5,463,690, October 1995.

- Diffie W. and Hellman M. *New directions in cryptography*. IEEE Trans. Inform. Theory, 22 (1976), pp. 644–654.

- ElGamal T. *A public key cryptosystem and a signature scheme based on discrete logarithms*. IEEE Transactions on Information Theory, 31(4): 469–472, July 1985.

- IEEE. P1363: *Standard specifications for public-key cryptography*. Draft Version 7, September 1998.

- IEEE. P1363: *Standard specifications for public-key cryptography*. Draft Version 13, November 12, 1999.

- Jebelean T. *Comparing several gcd algorithms*. Proceedings of the 11th Symposium on Computer Arithmetic, 180–185, IEEE Press, 1993.

- Kaliski Jr. B. S. *The Montgomery inverse and its applications*. IEEE Transactions on Computers, 44(8): 1064–1065, August 1995.

- Knuth D. E. *The Art of Computer Programming, Volume 2, Seminumerical Algorithms*. Addision-Wesley, Reading, MA, 3d Edition.

- Koblitz N. *A Course in Number Theory and Cryptography*. 2nd ed., Springer-Verlag, New York, 1994.

- Koblitz N. *Algebraic Aspects of Cryptography*. Springer-Verlag, Berlin, Heidelberg, New York, 1998.

- Koç Ç. K. *High-Speed RSA Implementation*. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.

- Koç Ç. K., Acar T., and Kaliski Jr. B. S. *Analyzing and comparing Montgomery multiplication algorithms*. IEEE Micro, 16(3): 26–33, June 1996.

- Lidl R. and Niederreiter H. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, 1994.

- Menezes A. J. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.

- Menezes A. J., P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.

- McEliece R. J. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

- Montgomery P. L. *Modular multiplication without trial division*. Mathematics of Computation, 44(170): 519–521, April 1985.

- National Institute for Standards and Technology. *Digital Signature Standard. FIPS publication 186-2*. January 2000.

- Rivest R. L., Shamir A., and Adleman L. *A method for obtaining digital signatures and public-key cryptosystems*. Communications of the ACM, 21(2): 120–126, February 1978.

- Savaş E. and Koç Ç. K. *The Montgomery modular inverse – revisited*. IEEE Transactions on Computers, 49(7): 763–766, July 2000.

- Silverman J. H. *The Arithmetic of Elliptic Curves*. Springer, Berlin, Germany, 1986.