

AN ABSTRACT OF THE RESEARCH OF

Tuğrul Yanık for the degree of Doctor of Philosophy in
Electrical & Computer Engineering presented on November 21, 2001.

Title: New Methods for Finite Field Arithmetic

Abstract approved: _____

Çetin K. Koç

We describe novel methods for obtaining fast software implementations of the arithmetic operations in the finite field $GF(p)$ and $GF(p^k)$. In $GF(p)$ we realize an extensive speedup in modular addition and subtraction routines and some small speedup in the modular multiplication routine with an arbitrary prime modulus p which is of arbitrary length. The most important feature of the method is that it avoids bit-level operations which are slow on microprocessors and performs word-level operations which are significantly faster. The proposed method has applications in public-key cryptographic algorithms defined over the finite field $GF(p)$, most notably the elliptic curve digital signature algorithm. The new method provides up to 13 % speedup in the execution of the ECDSA algorithm over the field $GF(p)$ for the length of p in the range $161 \leq k \leq 256$.

In the finite extension field $GF(p^k)$ we describe two new methods for obtaining fast software implementations of the modular multiplication operation with an arbitrary prime modulus p , which has less bit-length than the word-length of a microprocessor and an arbitrary generator polynomial. The second algorithm is a significant improvement over the first algorithm by using the same concepts introduced in $GF(p)$ arithmetic.

©Copyright by Tuğrul Yanık

November 21, 2001

All Rights Reserved

New Methods for Finite Field Arithmetic

by

Tuğrul Yanık

A THESIS submitted

to

Oregon State University

in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

Completed November 21, 2001

Commencement June 2002

Doctor of Philosophy Thesis of Tuğrul Yanık presented on November 21, 2001

APPROVED:

Major Professor, representing Electrical & Computer Engineering

Head of Electrical & Computer Engineering Department

Dean of Graduate School

I understand that my Thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my Thesis to any reader upon request.

Tuğrul Yanık, Author

ACKNOWLEDGMENTS

I am grateful to my advisor Prof. Çetin K. Koç for his reviews, guidance and encouragement in conducting this research and producing the papers which formed a basis for this thesis.

I am thankful to my committee members for their comments and reviews, particularly to Dr. Thomas A. Schmidt from the Mathematics Department.

At various stages in the writing of this papers, a number of people have given me invaluable technical help, advise and comments. In this regard I owe a debt of gratitude to the Information Security Lab researchers of the Oregon State University.

In particular, thanks to Serdar Erdem for his interest and feedback, Erkay Savaş and Murat Aydos for providing valuable support during various research phases.

Finally, I acknowledge financial support for this work from SITI and rTrust.

Tuğrul Yanık

Corvallis, OREGON

November 2001

TABLE OF CONTENTS

	<u>Page</u>	
1	INTRODUCTION	1
1.1	Motivation	2
1.2	Thesis Outline	3
2	CRYPTOGRAPHIC SYSTEMS	5
2.1	Introduction	5
2.2	Private Key Cryptography	6
2.3	Public Key Cryptography	7
2.3.1	Trapdoor Knapsacks	9
2.3.2	RSA Cryptosystem	10
2.3.3	ElGamal Cryptosystem	11
2.3.4	Elliptic Curve Cryptosystems	12
3	ELLIPTIC CURVES	14
3.1	Introduction	14
3.2	Definition	14
3.3	Addition Formula	15
3.4	Group Structure	16
3.5	Counting Elliptic Curves	17
3.6	Isomorphisms of Elliptic Curves	17
3.7	Implementation of Elliptic Curve Cryptosystems	19
3.7.1	ECC Arithmetic Using Projective Coordinates	19
3.7.2	ECC Arithmetic Using Modified Jacobian Coordinates	21
3.7.3	Elliptic Scalar Multiplication	23
4	THE ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)	25

TABLE OF CONTENTS (CONTINUED)

		<u>Page</u>
4.1	Introduction	25
4.2	Domain Parameters	26
4.3	Elliptic Curve Digital Signature Algorithm	26
5	INCOMPLETE MODULAR ARITHMETIC IN $GF(p)$	28
5.1	Introduction	28
5.2	Representation of the Numbers	28
	5.2.1 Incompletely Reduced Numbers	30
	5.2.2 Positive and Negative Numbers	31
	5.2.3 A Representation Example	31
5.3	Modular Addition	32
5.4	Addition Examples	34
5.5	Modular Subtraction	35
5.6	Subtraction Examples	37
5.7	Montgomery Modular Multiplication	38
5.8	Multiplication Examples	43
5.9	Implementation Results and Conclusions	46
6	DOUBLE MONTGOMERY MULTIPLICATION IN $GF(p^k)$	49
6.1	Introduction	49
6.2	Previous Work	50
6.3	Polynomial Representation	51
6.4	Double Montgomery Multiplication in $GF(p^k)$	52
6.5	Incomplete Reduction in $GF(p^k)$	60
6.6	Implementation Results	65

TABLE OF CONTENTS (CONTINUED)

	<u>Page</u>
6.7 Comparing $GF(p)$ and $GF(p^k)$ Arithmetic	67
6.8 Conclusions and Further Research	68
7 MODULAR INVERSION IN $GF(p^k)$	69
7.1 Introduction	69
7.2 Inversion Algorithms	70
7.2.1 OEF Inversion	70
7.2.2 Extended Euclidian Algorithms	71
7.2.3 Almost Inverse Algorithms	74
7.3 Conclusion	76
8 CONCLUSIONS	77

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
2.1. RSA Algorithm		10
2.2. ElGamal Public Key Encryption		11

LIST OF TABLES

<u>Table</u>	<u>Page</u>
5.1. Incomplete and complete arithmetic timings	46
5.2. ECDSA over $GF(p)$ signature generation timings.	48
6.1. The operation counts for the Double Montgomery multiplication algorithm.	59
6.2. The operation counts for the Incomplete Double Montgomery multiplication algorithm.	64
6.3. Timings of the DMM and IDMM in two different platforms. . .	66
6.4. $GF(p^k)$ vs. $GF(q)$ modular multiplication timings.	67
7.1. Complexity for inverse algorithms.	74

To my wife and my children...

for their endless love and support

New Methods for Finite Field Arithmetic

CHAPTER 1 INTRODUCTION

"Research is what I'm doing when I don't know what I'm doing."

Wernher Von Braun (1912-1977)

The dawning of the information age revealed the necessity of information security by making many enemies like viruses, hackers, electronic eavesdropping and electronic fraud. Sensitive information like financial transactions, health and legal reports, credit ratings are transmitted over electronic medium which is open to public use. This fact has heightened the awareness of the need to protect the confidentiality and integrity and to guarantee the authenticity of the source and information.

To provide a secure environment where communication and electronic commerce can be conducted without fear, we rely on the disciplines of cryptography and network security. We need to implement the necessary cryptographic functions in very efficient ways, keeping in mind that high performance and practicality are of utmost importance. On servers that deal with hundreds of transactions each minute, time spend in cryptographic functions can be critical for the availability of services.

In this thesis, we concentrate on developing high-speed algorithms which can be used in a number of cryptographic systems. Specifically, we are focusing on elliptic curve cryptosystems because of their high security with short key lengths. This property has made the use of Elliptic Curve Digital Signature Algorithm (ECDSA) very popular. ECDSA generates shorter signatures providing the same amount of security as other cryptosystems (RSA, ElGamal, etc.) with longer signatures. Elliptic curve cryptosystems require space and time efficient implementations in arithmetic operations conducted over finite fields.

We are proposing new methods for arithmetic operations over the finite fields $GF(p)$ and $GF(p^k)$. Since arithmetic operations in finite fields have applications in

coding theory and computer algebra the proposed methods in this thesis might have applications in those areas.

1.1 Motivation

Arithmetic in finite fields is the base of many public-key algorithms, including the Diffie-Hellman key exchange algorithm [7], elliptic curve cryptography [15, 24], and the Elliptic Curve Digital Signature Algorithm (ECDSA) [27]. The overall performance of these cryptographic algorithms depend on the efficiency of the arithmetic performed in the underlying finite field.

Specifically, the motivation of this research was to speed up the ECDSA signature and verification time. The profiling data obtained from an ECDSA implementation demonstrated that 85 % of the overall time was spend in modular multiplication, addition and subtraction. Speedup in those routines is crucial to speedup the overall signature time. We introduced the incomplete modular arithmetic concept which is a practical and effective approach to speedup the ECDSA signature and verification procedures.

We continued to work on $GF(p^k)$ arithmetic to produce a fast arithmetic library to implement ECDSA. Previous work of Bailey and Paar exploits the fact that software implementations of $GF(p^k)$ have advantages over $GF(p)$ for performing multiprecision arithmetic without carry propagation, and over $GF(2^k)$ for performing word size calculations supported by the microprocessor. Bailey and Paar propose to use special forms for p and the generator polynomial to construct the extension field $GF(p^k)$ [2, 1]. This choice results in a dramatic tradeoff between the performance of arithmetic in the extension field and the number of the extension fields available to use. Most of the research conducted in this subject has advanced on that direction.

Our motivation was to implement the finite field arithmetic without limiting the the prime characteristic p and the generator polynomial and still obtain comparable timing results to the arithmetic implemented by Bailey and Paar.

1.2 Thesis Outline

Chapter 2 provides a short background of cryptographic systems. The definitions of private key cryptosystems and public key cryptosystems are given and their properties are explained. In addition, some of the well known public key cryptosystems are explained.

In Chapter 3, an overview of elliptic curves and their properties that make it possible to use them in cryptographic systems are given.

Chapter 4 explains the Elliptic Curve Digital Signature Algorithm (ECDSA). We are giving the algorithms to generate the private and public keys, generate the digital signature and verify it.

In Chapter 5, we are focusing on finite field arithmetic in $GF(p)$. The basic arithmetic operations (i.e., addition, subtraction, and multiplication) in the finite field $GF(p)$ have several applications in cryptography, such as decipherment operation of the RSA algorithm [28], the Diffie-Hellman key exchange algorithm [7], elliptic curve cryptography [15, 24], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm (ECDSA) [27]. These applications demand high-speed software implementations of the arithmetic operations in $GF(p)$ for $160 \leq \lceil \log_2(p) \rceil \leq 2048$. In Chapter 5, we introduce the incomplete modular arithmetic concept and describe a new method for obtaining high-speed software implementations of the arithmetic operations on the ARM microprocessor and general-purpose computers.

In Chapter 6, we investigate to finite field arithmetic in $GF(p^k)$. We are proposing two new methods for modular multiplication in $GF(p^k)$. The first method is called the Double Montgomery Multiplication where we use the Montgomery multiplication algorithm both for the polynomial multiplication in the extension field and the coefficient multiplications in the subfield $GF(p)$. We obtained fast timing results using this method. The second method is an improvement over the Double Montgomery Multiplication method. We are using the incomplete arithmetic concept introduced in Chapter 5 to reduce the complexity for the subfield multiplications

performed in $GF(p)$. The second method provides us a 18–30 % speedup in the ARM7TDMI microprocessor implementation and 46–69 % speedup in a Pentium II processor implementation over the first method. Both methods have no restriction on the generator polynomial except it is monic and irreducible as expected. In addition, both algorithms restrict the prime p to have less bit-length than the word-length of the microprocessor.

Chapter 7 explains the modular inversion operation in $GF(p^k)$. It gives several inversion algorithms from different papers which are suitable to use with the multiplication algorithms introduced in Chapter 6.

CHAPTER 2

CRYPTOGRAPHIC SYSTEMS

”The art of war teaches us to rely not on the likelihood of the enemy’s not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather of the fact that we have made our position unassailable.”

The Art of War, Sun Tzu

2.1 Introduction

In the past, cryptography has been used to secure military and diplomatic communications. Most governments exercise control over cryptographic devices if not over cryptographic research. But the information age has brought urgent need of cryptography into the private sector. Sensitive information such as financial transactions, health and legal reports, commercial agreements are transmitted over public communication mediums. This type of information needs to be protected from any attack that endangers its confidentiality, integrity and authenticity. In general, cryptographic systems have four basic services which can be expressed as:

- Confidentiality: a service used to keep the content of information hidden from unauthorized parties.
- Authentication: a service related to identification. Authentication applies to both identification and information itself. Parties entering into a communication should identify each other. Information exchanged over a channel should be authenticated as to origin, data content, time sent, etc.
- Data Integrity: a service which deals with unauthorized alteration of data. The system should be capable of detecting manipulation in data by unauthorized parties. Data can be manipulated by insertion, deletion and substitution.

- Non-repudiation: a service which prevents parties from denying previous commitments and actions. This can be achieved by ensuring that the previous transactions are nonrevocable so that they are legally binding. Neither the sender nor the receiver of a message should be able to deny the transaction.

To achieve these goals two general methods were developed which were named as private key cryptography and public key cryptography. In this thesis, we are working on applications related to public key cryptography. But we will make definitions for both methods.

2.2 Private Key Cryptography

The private key cryptography is also referred as the conventional encryption model. This method was the only method used before public key cryptography was invented. It is widely used due to the performance it provides when huge amount of information is encrypted.

The original message referred to as plaintext is encrypted to a random, meaningless message referred to as ciphertext by applying an algorithm that uses a key independent from the plaintext. Different keys produce different ciphertexts. The same key is used to decrypt ciphertext to plaintext applying the decryption algorithm. The below notation is giving a better idea how this method works.

Let \mathcal{M} denote the set of all possible plaintext messages, \mathcal{C} the set of all possible ciphertext messages and \mathcal{K} the set of all possible keys.

A *private key cryptosystem* consists of a family of pairs of functions

$$E_k : \mathcal{M} \longrightarrow \mathcal{C} \quad \text{and} \quad D_k : \mathcal{C} \longrightarrow \mathcal{M} \quad k \in \mathcal{K} ,$$

such that

$$D_k(E_k(m)) = m \quad \text{for all } m \in \mathcal{M} \text{ and } k \in \mathcal{K} .$$

E_k is the encryption function. D_k is the decryption function. If two or more parties want to use this cryptographic system, they have to agree on one or more

keys and solve the key distribution and key management problems. This can be a serious problem if a secure medium to transmit the keys doesn't exist.

Another shortcoming of private key cryptology is its inability to support *digital signature* schemes. A *digital signature* is an electronic analogue of a hand-written signature that allows a receiver to convince a third party that the message is in fact originated from the sender.

Classical examples to private key cryptography are the Playfair cipher, the Hill cipher and rotor machines. Examples to modern techniques are the Data Encryption standard (DES) and the Advanced Encryption Standard (AES) which was announced on February 2001 by the National Institute of Standards and Technology (NIST).

2.3 Public Key Cryptography

The invention of of public key cryptography in 1976 by W.Diffie and M.Hellman [7] was a real breakthrough in the history of cryptography. W.Diffie and M.Hellman were trying to find a method which could overcome the two major shortcomings of private key cryptography which were inefficient and insecure key distribution and inability to sign a digital message. They came up with a method that addressed both problems.

The public key cryptosystem they proposed has keys come in inverse pairs and each pair of keys has two properties:

- Anything encrypted with one key can be decrypted with the other corresponding key.
- Given one of the keys, the public key, it is infeasible to discover the other key, the secret key.

The encryption and decryption is separated which makes it possible to name one of the keys as the public key and publish it. The following protocols explain how public key cryptosystems work.

- One can send a private message to someone by encrypting the message with

the receiver's public key. Only the receiver can decrypt the message with his secret key.

- One can sign a message by encrypting it with his own secret key. Others having access to the specific public key can verify that the message was encrypted with the corresponding secret key. No one else can forge a message with the same property.

The first aspect simplifies the key management which was a serious problem in large networks for the private key cryptography. The second aspect makes it possible to sign and verify a digital message.

Diffie and Hellman introduced a key exchange protocol in their first publication [7] along with their ideas of public key cryptography. Their protocol is known as *Diffie-Hellman key exchange*. And in terms of an arbitrary group it can be described as:

1. (Setup) A and B publicly select a (multiplicatively written) finite group G and an element $\alpha \in G$.
2. A generates a random integer a , computes α^a in G , and transmits α^a to B over a public communications channel.
3. B generates a random integer b , computes α^b in G , and transmits α^b to A over a public communications channel.
4. A receives α^b and computes $(\alpha^b)^a$.
5. B receives α^a and computes $(\alpha^a)^b$.

A and B now share the common group element α^{ab} . Note that an eavesdropper knows G, α, α^a and α^b , and his task is to use this information to reconstruct α^{ab} . This problem is commonly referred to as *Diffie-Hellman problem*.

The problem of computing a , given G, α and α^a is called the *discrete logarithm problem*. It has not been proven but widely believed that the discrete logarithm problem and the Diffie-Hellman problem are computationally equivalent [24].

2.3.1 Trapdoor Knapsacks

Later in the same year of 1976, Ralph Merkle began to work on his best-known contribution to public key cryptography. He build trapdoors into the knapsack one-way function to develop the trapdoor knapsack public key cryptosystem. Given a cargo vector of integers $a = (a_1, a_2, \dots, a_n)$ it is easy to add together the elements of a specified subvector. It is not easy to figure out a subvector given a sum S of some of the elements of the cargo vector. This problem is known to be a NP-complete problem. The cargo vector a can be used to encrypt an n-bit message $x = (x_1, x_2, \dots, x_n)$ by applying the dot product $S = a \cdot x$, where S is the ciphertext.

Merkle choose the cargo vector a such that each element is larger than the sum of the preceding elements which is called superincreasing. We name this vector as a' . It is easy to find out the message x if a' and S' are given and $S' = a' \cdot x$ holds. Therefore a random superincreasing vector a' with hundreds of components is chosen and kept secret. We further need a random integer m which should be larger than $\sum a'$ and a random integer w , relatively prime to m . The inverse $w^{-1} \text{ mod } m$ is used in decryption.

The cargo vector a is obtained by multiplying each component of a' by $w \text{ mod } m$.

$$a = a' \cdot w \text{ mod } m$$

Alice's public key is a permuted version of a . She keps the permutation, the simple cargo vector a', w, w^{-1} and the modulus m secret as her private key. If Bob wants to send a message x to Alice, he computes $S = a \cdot x$ and sends S .

$$\begin{aligned} S' &= w^{-1} S \text{ mod } m \\ &= w^{-1} \sum a_i x_i \text{ mod } m \\ &= w^{-1} \sum (w a'_i \text{ mod } m) x_i \text{ mod } m \\ &= \sum (w^{-1} w a'_i \text{ mod } m) x_i \text{ mod } m \\ &= \sum a'_i x_i \text{ mod } m \\ &= a' x \end{aligned}$$

When $m > \sum a'_i$, Alice can use her secret information $w^{-1} \text{ mod } m$ to transform any message S to S' and solve $S' = a' \cdot x$ for x which is easy to compute.

In 1982 the single iteration knapsack cryptosystem was first broken by Adi Shamir and others followed. Two years later multi iteration knapsack cryptosystems were broken which was the end of the knapsack systems.

2.3.2 RSA Cryptosystem

The RSA cryptosystem was invented in 1977 by Rivest, Shamir and Adleman [29] and was the first realization of Diffie-Hellman's abstract model for public key cryptography. The security of the RSA cryptosystem rests on the integer factorization problem. The key generation, encryption and decryption methods of the RSA algorithm are give as below.

Figure 2.1. RSA Algorithm

KEY GENERATION	
• Select p, q	p and q both prime
• Calculate $n = p \times q$	
• Calculate $\phi(n) = (p - 1)(q - 1)$	
• Select integer e	$\text{gcd}(\phi(n), e) = 1; 1 < e < \phi(n)$
• Calculate d	$d = e^{-1} \text{ mod } \phi(n)$
• Public Key	$\{e, n\}$
• Private Key	$\{d, n\}$
ENCRYPTION	
• Plaintext	$M < n$
• Ciphertext	$C = M^e \text{ (mod } n)$
DECRYPTION	
• Plaintext	C
• Ciphertext	$M = C^d \text{ (mod } n)$

2.3.3 ElGamal Cryptosystem

In 1985, T. ElGamal [9] proposed a public key scheme based on the discrete logarithm problem. The scheme is given in Figure 2.2.

Figure 2.2. ElGamal Public Key Encryption

KEY GENERATION	
• Generate p	a large random prime
• Generate α	a generator of the multiplicative group G of integers modulo p
• Select random integer a	$1 \leq a \leq p - 2$
• Compute $\alpha^a \text{ mod } p$	
• Public Key	(p, α, α^a)
• Private Key	a
ENCRYPTION	
• Obtain the receivers authentic public key	(p, α, α^a)
• Represent message as integer m	$0 \leq m \leq p - 1$
• Select random integer k	$1 \leq k \leq p - 2$
• Compute $\gamma = \alpha^k \text{ mod } p$	
• Compute $\delta = m \cdot (\alpha^a)^k \text{ mod } p$	
• Send the ciphertext $c = (\gamma, \delta)$ to receiver	
DECRYPTION	
• Compute $\gamma^{p-1-\alpha} \text{ mod } p$	$\gamma^{p-1-\alpha} = \gamma^{-a} = \alpha^{-\alpha k}$
• Recover m by $m = (\gamma^{-a}) \cdot \delta \text{ mod } p$	

The problem of breaking the ElGamal encryption scheme, i.e., recovering m given $p, \alpha, \alpha^a, \gamma$ and δ , is equivalent to solving the Diffie-Hellman problem. The ElGamal encryption scheme can be viewed as comprising a Diffie-Hellman key exchange to determine a session key α^{ak} , and then encrypting the message multiplying with the session key. ElGamal [9] also designed a signature scheme, which makes use of the

group G [24]. It is critical that different random integers k are used to encrypt different messages. If the same k is used to encrypt two messages m_1 and m_2 and the resulting ciphertext pairs are (γ_1, δ_1) and (γ_2, δ_2) , then $\delta_1/\delta_2 = m_1/m_2$. m_2 could be computed if m_1 were known.

2.3.4 Elliptic Curve Cryptosystems

Elliptic curves have been studied by mathematicians for more than a century. Besides their recent cryptographic applications they are used in primality testing and integer factorization. Elliptic curves were first suggested in 1985 independently by N. Koblitz [15] and V. Miller [25] for implementing public key cryptosystems. The majority of the products and standards that use public key cryptography for encryption and digital signatures use RSA. But the bit length for secure RSA use has increased over recent years, and this has increased the processing load on applications using RSA. Recently, the Elliptic Curve Cryptology (ECC) has begun to challenge RSA.

The security of ECC rests on the discrete logarithm problem over the points of an elliptic curve. When the curve is defined over a finite field the points of the curve form an abelian group. The addition of the points can be implemented efficiently both in software and hardware.

As it is in the case with the integer factorization problem and the discrete logarithm problem modulo p , no efficient algorithm is known to solve the elliptic curve discrete logarithm problem. Of the three problems, the integer factorization and the discrete logarithm problem modulo p both admit general algorithms that run in sub-exponential time. This means the problem is still hard, but not as hard as those problems that admit only fully exponential algorithms. The best general algorithm for the elliptic curve discrete logarithm problem is fully exponential time. Therefore, cryptosystems that rely on the elliptic curve discrete logarithm problem provide higher strength-per-bit than the other cryptosystems that rely on the integer factorization problem and the discrete logarithm problem modulo p .

Having shorter key lengths mean smaller bandwidth and memory requirements,

which is a crucial factor in some applications such as design of smart cards, where both memory and processing power are limited. Another advantage of using the elliptic curves is that each user may select a different curve, even though the underlying field is the same for all. That means each user can change his curve periodically (for extra security) without changing the hardware [24].

CHAPTER 3

ELLIPTIC CURVES

"I asked my mother [a mathematician] whether mathematics was a difficult topic. She said to be sure to learn all the formulas and be sure you know them. The second thing to remember is if you need more than five lines to prove something, then you're on the wrong track."

Edsger W. Dijkstra

3.1 Introduction

After elliptic curves cryptosystems were invented by Neal Koblitz [15] and Victor Miller [25] they became one of the most popular cryptosystems due to the difficulty of the mathematical problem their security relies on. The elliptic curve discrete logarithm problem (ECDLP) appears to be harder than the integer factoring and discrete logarithm problem [24]. Therefore the strength-per-key-bit is substantially greater than the cryptosystems relying on the integer factoring and discrete logarithm problem. The advantages gained from smaller parameters are speed and smaller keys. These are important in environments where processing power, storage space or power consumption is constrained. In this chapter we will give an overview of elliptic curves and how they are used in cryptographic systems.

3.2 Definition

For cryptographic purposes we are only interested in elliptic curves defined over finite fields. An elliptic curve E over a field F_q , where $q = p^k$ and p is a prime greater than 3, is the group formed by \mathcal{O} and the solutions in F_q to

$$y^2 = x^3 + ax + b \tag{3.1}$$

for $a, b \in F_q$ such that $\Delta = -16(4a^3 + 27b^2)$ is non-zero. \mathcal{O} is a special point, called the point of infinity. Δ is called the discriminant and equals to zero if $f(x) = x^3 + ax + b$ has a double root. We say that E of equation $y^2 = x^3 + ax + b$ is non-singular if $\Delta \neq 0$.

3.3 Addition Formula

There is a rule for adding two points on an elliptic curve $E(F_q)$ to obtain a third point on the curve which forms an abelian group with the identity element \mathcal{O} . Using this group we can construct elliptic curve cryptosystems.

The addition rule is best explained geometrically as follows. Let $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ be two distinct points on an elliptic curve E . Then the sum of P and Q is found by drawing a line through P and Q which intersects the curve at a third point. The result R is the reflection of this third point in the x-axis.

If we need to add a point $P = (x_1, y_1)$ to itself we need to draw the tangent line to the elliptic curve point which intersects the curve at a third point. The result is the reflection of this third point in the x-axis. The addition rules for all $P, Q \in E$ can be summarized as follows:

1. $\mathcal{O} + P = P$ and $P + \mathcal{O} = P$.
2. $-\mathcal{O} = \mathcal{O}$.
3. If $P = (x_1, y_1) \neq \mathcal{O}$, then $-P = (x_1, -y_1)$.
4. If $P = -Q$ then $P + Q = \mathcal{O}$.
5. If $P \neq \mathcal{O}$, $Q \neq \mathcal{O}$, $Q \neq -P$, then the addition formulas for *affine coordinates* are given as follows [11].

Let $P = (x_1, y_1)$, $Q = (x_2, y_2)$ and $R = P + Q = (x_3, y_3)$ be points on elliptic curve $E(F_q)$.

- *EC addition formulas when $(P \neq \pm Q)$:*

$$x_3 = \lambda^2 - x_1 - x_2$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

where

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$$

- *EC doubling formulas when $(P = Q)$*

Let $P = (x_1, y_1)$ a point on the curve. Then, $2(x_1, y_1) = (x_2, y_2)$, where

$$\begin{aligned} x_2 &= \lambda^2 - 2x_1 \\ y_2 &= \lambda(x_1 - x_2) - y_1 \end{aligned}$$

where

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

3.4 Group Structure

There are q^2 pairs $(x, y) \in F_q \times F_q$. Exactly half of the elements of F_q^* are squares. There can be at most $q/2$ different y coordinates for the elliptic curve $E : y^2 = x^3 + ax + b$. For each y there could be two different x coordinates that satisfying the equation. Then we should expect that the elliptic curve E will have approximately q elements in $E(F_q)$. The following theorem of H.Hasse shows that our expectation is correct. The theorems stated in this chapter can be found in [33, 24].

Theorem 3.1 (Hasse) *Let E be an elliptic curve defined over F_q . Let $\#E(F_q)$ be the number of points of $E(F_q)$, including the single point of infinity. Then*

$$| \#E(F_q) - q - 1 | \leq 2\sqrt{q}.$$

The number of points of an elliptic curve can be computed with Schoof's algorithm [31]. The following theorem gives us a helpful result on the group structure of elliptic curves.

Theorem 3.2 (Cassells) *Let E be an elliptic curve defined over F_q . The group structure of E is either cyclic or is the product of two cyclic groups. Furthermore if it is not cyclic, then the group structure is $Z/mZ \oplus Z/nZ$, with $m|n$ and $m|(q-1)$.*

3.5 Counting Elliptic Curves

We want to use the elliptic curves for cryptographic applications. Therefore, we should have a notion of how many elliptic curves there are for a field F_q . The straightforward guess is that there are q choices for each of a and b in equation (3.1). But some of these will produce singular curves and there are different equations which can represent the same elliptic curve. The following theorems will clarify the number of elliptic curves for a given field.

Theorem 3.3 *Let q be such that the finite field F_q is of characteristic greater than 3. Then there are $q^2 - q$ distinct elliptic curves defined over F_q .*

3.6 Isomorphisms of Elliptic Curves

Two elliptic curves are isomorphic if there is a 1-1 and onto map between them that can be given locally by rational functions. The following is a precise definition.

Definition 3.1 *Let $E : y^2 = x^3 + ax + b$ and $E' : y^2 = x^3 + a'x + b'$ to be two elliptic curves defined over the field F . We say that E is isomorphic to E' over F if there is some $u \in F^*$ such that $a' = u^4a$ and $b' = u^6b$. We denote this by $E \cong E'$. We call the set of all elliptic curves isomorphic to a fixed E its isomorphism class.*

Definition 3.2 *The j -invariant of an elliptic curve $E : y^2 = x^3 + ax + b$ is the quantity*

$$j(E) = -1728 \cdot (4a)^3 / \Delta.$$

Theorem 3.4 *If E and E' are isomorphic elliptic curves, then $j(E) = j(E')$.*

It is interesting that the j -invariant comes close to determining the isomorphism class of E .

Theorem 3.5 *If E and E' are defined over some field F and $j(E) = j(E')$, then E and E' are isomorphic over the algebraic closure of \overline{F} of F .*

Theorem 3.6 *Let j_0 be a value in a field F . Then there exists an elliptic curve E defined over F such that $j(E) = j_0$.*

To develop a notion of maps from an elliptic curve to itself we can focus on isomorphism classes over F_q .

Definition 3.3 *Let $E : y^2 = x^3 + ax + b$. Then $u \in F^*$ defines an automorphism from E to itself if $a = u^4a$ and $b = u^6b$. We denote the set of all automorphisms of E by $Aut(E)$.*

Theorem 3.7 *Let q be such that the finite field F_q is of characteristic greater than 3. Let E be an elliptic curve defined over F_q . Then the number of automorphisms of E is*

$$\#Aut(E) = \begin{cases} 4 & \text{if } j(E) = 1728 \text{ and } 4 \text{ divides } q - 1; \\ 6 & \text{if } j(E) = 0 \text{ and } 3 \text{ divides } q - 1; \\ 2 & \text{otherwise.} \end{cases}$$

Theorem 3.8 (Waterhouse) *Let q be such that the finite field F_q is of characteristic greater than 3. Then the number of isomorphism classes of elliptic curves over F_q is*

$$\begin{cases} 2q + 6 & \text{if } q \equiv 1 \pmod{12}; \\ 2q + 2 & \text{if } q \equiv 5 \pmod{12}; \\ 2q + 4 & \text{if } q \equiv 7 \pmod{12}; \\ 2q & \text{if } q \equiv 11 \pmod{12}. \end{cases}$$

The following proof is a simplified version given by A.J.Menezes and was taken from lecture notes of Thomas A. Schmidt from the Oregon State University. By Theorem (3.3) there are $q^2 - q$ distinct elliptic curves defined over F_q . This means the sum of elliptic curves within distinct isomorphism classes must equal $q^2 - q$. If there were no automorphisms there would be $q - 1$ curves in the isomorphism classes, because we would simply find all $a' = u^4a$ and $b' = u^6b$. But actually, there

are $(q - 1)/\#Aut(E)$ curves in the isomorphism class. The following equation is summing over the isomorphism classes.

$$\sum_{isom.class \ rep \ E} (q - 1)/\#Aut(E) = q^2 - q \quad (3.2)$$

Canceling the factor $q - 1$ we get,

$$\sum_{isom.class \ rep \ E} 1/\#Aut(E) = q. \quad (3.3)$$

If all of the $\#Aut(E)$ are equal to 2, then the number of classes would become $N = 2q$. This is true when neither 3 nor 4 divides $q - 1$ (Theorem (3.7)). That is when $q \equiv \text{mod } 12$ according to Theorem (3.8).

If $j = 0$, then $a = 0$. There are $q - 1$ choices for $b \in F_q^*$. All of these curves will have the same order of the automorphism group. If there is an isomorphism class with an automorphism group of order 4, then there will be $(q - 1)/4$ such classes. Similarly, if $j = 1728$, then $b = 0$. If there is an isomorphism class with an automorphism group of order 6, then there will be $(q - 1)/6$ such classes.

Suppose that 3 divides $q - 1$, but 4 does not. Then equation (3.3) becomes $6 \cdot (1/6) + (N - 6) \cdot (1/2) = q$. Solving this we find $N = 2q + 4$. This is the case of $q \equiv 7 \text{ mod } 12$. The remaining cases can be treated similarly.

3.7 Implementation of Elliptic Curve Cryptosystems

3.7.1 ECC Arithmetic Using Projective Coordinates

The addition operation of affine coordinates requires a field inversion which is very expensive compared to field multiplication. This inverse operation can be eliminated by using *projective coordinates*. In [5] different coordinate systems are explained and compared regarding speed and memory requirements. The *projective coordinates* are expressed as X , Y , and Z . We will give the addition and doubling formulas for $GF(p)$, the formulas for $GF(p^k)$ are very similar and given in [22]. Converting

affine coordinates to projective coordinates is trivial. The affine coordinate (x, y) is converted as $(X = x, Y = y, Z = 1)$. Converting projective coordinates to affine coordinates requires the following computations.

$$x = \frac{X}{Z^2}, \quad (3.4)$$

$$y = \frac{Y}{Z^3} \quad (3.5)$$

The addition formulas are given as follows [10, 11]. Let $P = (X_1, Y_1, Z_1)$, $Q = (X_2, Y_2, Z_2)$ and $K = P + Q = (X_3, Y_3, Z_3)$ be points on elliptic curve $E(F_p)$.

- *EC projective point addition formulas when $(P \neq \pm Q)$:*

$$\begin{aligned} X_3 &= vA \\ Y_3 &= u(v^2 X_1 Z_2 - A) - v^3 Y_1 Z_2 \\ Z_3 &= v^3 Z_1 Z_2 \end{aligned}$$

where

$$\begin{aligned} u &= Y_2 Z_1 - Y_1 Z_2 \\ v &= X_2 Z_1 - X_1 Z_2 \\ A &= u^2 Z_1 Z_2 - v^3 - 2v^2 X_1 Z_2 \end{aligned}$$

- *EC projective point doubling formulas when $(P = Q)$:*

Let $P = (X_1, Y_1, Z_1)$ be a point on the curve. Then, $2P = (X_2, Y_2, Z_2)$.

$$\begin{aligned} X_3 &= 2hs \\ Y_3 &= w(4B - h) - 8Y_1^2 s^2 \\ Z_3 &= 8s^3 \end{aligned}$$

where

$$w = aZ_1^2 + 3X_1^2$$

$$s = Y_1Z_1$$

$$B = X_1Y_1$$

$$h = w^2 - 8B$$

Point addition for projective coordinates requires 12 multiplications, 2 squarings and doubling requires 7 multiplications, 5 squarings.

In the case of $a = p - 3$ the number of multiplications can be reduced [11].

3.7.2 ECC Arithmetic Using Modified Jacobian Coordinates

If we want to continue to make trade-offs between memory and speed we can use the *modified Jacobian coordinates* which require one more coordinate but have a faster point doubling equations than the projective coordinates [5]. In section 3.5.3 we explain elliptic scalar multiplication algorithms. These algorithms show that point doubling is used far more than point addition. Again, we will give the addition and doubling formulas for $GF(p)$. Converting affine coordinates to modified Jacobian coordinates is trivial as in projective coordinates. The affine coordinates (x, y) are converted as $(X = x, Y = y, Z = 1, W = a)$. Converting Jacobian coordinates to affine coordinates requires the following computations.

$$x = \frac{X}{Z^2}, \tag{3.6}$$

$$y = \frac{Y}{Z^3} \tag{3.7}$$

The new elliptic curve equation becomes the following

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{3.8}$$

The *modified Jacobian coordinates* provide the fastest possible doublings. The addition formulas for both Jacobian and *modified Jacobian coordinates* are given in [5].

Let $P = (X_1, Y_1, Z_1, W_1)$, $Q = (X_2, Y_2, Z_2, W_2)$ and $K = P+Q = (X_3, Y_3, Z_3, W_3)$ be points on elliptic curve $E(F_p)$.

- *EC Jacobian point addition formulas when $(P \neq \pm Q)$ [5]:*

$$\begin{aligned} X_3 &= -H^3 - 2U_1H^2 + r^2 \\ Y_3 &= -S_1H^3 + r(U_1H^2 - X_3) \\ Z_3 &= Z_1Z_2H \\ W_3 &= aZ_3^4 \end{aligned}$$

where

$$\begin{aligned} U_1 &= X_1Z_2^2 \\ S_1 &= Y_1Z_2^3 \\ U_2 &= X_2Z_1^2 \\ S_2 &= Y_2Z_1^3 \\ H &= U_1 - U_2 \end{aligned}$$

- *EC Jacobian point doubling formulas when $(P = Q)$*

$$\begin{aligned} X_3 &= T \\ Y_3 &= M(S - T) - U \\ Z_3 &= 2Y_1Z_1 \\ W_3 &= 2U(aZ_1^4) \end{aligned}$$

where

$$S = 4X_1Y_1^2$$

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

Point addition for modified Jacobian coordinates requires 13 multiplications, 6 squarings and point doubling requires 4 multiplications, 4 squarings.

3.7.3 Elliptic Scalar Multiplication

The elliptic scalar multiplication is expressed as nP , where n is a integer and P is an elliptic curve point. By nP , we mean adding point P to itself n times. Scalar multiplication can be performed efficiently by using the signed digit representation as outlined below [11].

Elliptic Scalar Multiplication Algorithm

Input: Integer n and an elliptic curve point P .

Output: Elliptic curve point $S = nP$.

Step 1. If $n = 0$ then output \mathcal{O} and stop.

Step 2. If $n \leq 0$ the set $Q = (-P)$ and $k = (-n)$, else set $Q = P$ and $k = n$.

Step 3. Let $h_l h_{l-1} \dots h_1 h_0$ be the binary representation of $3k$,
where the most significant bit h_l is 1.

Step 4. Let $k_l k_{l-1} \dots k_1 k_0$ be the binary representation of k .

Step 5. Set $S = Q$.

Step 6. For i from $l - 1$ down to 1 do

Step 7. Set $S = 2S$.

Step 8. If $h_i = 1$ and $k_i = 0$ then compute $S = S + Q$

Step 9. If $h_i = 0$ and $k_i = 1$ then compute $S = S - Q$

Step 10. Output S .

Another implementation that doesn't require much memory and brings some speed up by reducing the number of point additions is the m -ary method [17]. The

m-ary method can be generalized by scanning the bits of the scalar n , $r = \log_2 m$ bits at a time. The number m is chosen as a power of 2 to make the scanning simple.

M-ary Elliptic Scalar Multiplication Algorithm

Input: Integer n and an elliptic curve point P .

Output: Elliptic curve point $Q = nP$.

Step 1. Compute and store wP for all $w = 2, 3, 4, \dots, m - 1$.

Step 2. Decompose n into r -bit words f_i for $i = 0, 1, 2, \dots, s - 1$.

Step 3. $Q = f_{s-1}P$

Step 4. For i from $s - 2$ down to 0 do

Step 5. Set $Q = 2^r Q$.

Step 6. If $f_i \neq 0$ then compute $Q = Q + f_i P$

Step 7. Output Q .

There are several modifications that improve the performance of these algorithms. Various sliding windows techniques can be found in [17].

To speed-up the elliptic scalar multiplication precomputation methods are used [3, 23]. These methods try to eliminate doublings by using table lookup methods where elliptic curve points are stored in tables. In [23] the section 14.6.3 explains algorithms that use precomputation methods.

CHAPTER 4

THE ELLIPTIC CURVE DIGITAL SIGNATURE ALGORITHM (ECDSA)

”There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

C. A. R. Hoare

4.1 Introduction

The Digital Signature Algorithm (DSA) was specified in a U.S. Government Federal Information Processing Standard (FIPS) called as the Digital Signature Standard (DSS). DSA’s security relies on the discrete logarithm problem in the prime-order subgroups of Z_p^* . The ECDSA is the elliptic curve analogue of DSA. ECDSA was proposed by Scott Vanstone [34] in 1992. It was accepted in 1998 as an ISO (International Standards Organization) standard, accepted in 1999 as an ANSI (American National Standards Institute) standard (ANSI X9.62), and accepted in 2000 as an IEEE standard (IEEE 1363-2000) and FIPS standard (FIPS 186-2).

Digital signature schemes are designed to provide the same functions of handwritten signatures for the digital environment. A digital signature is a number that is calculated from a secret known only by the signer and from the contents of the message that is signed. The signature should be verifiable by other parties without having any knowledge of the secret key. The signature generated should be unforgeable to prevent the signer from repudiating a signature he/she created and others from claiming that the signature is his/her signature.

The digital signature schemes can provide data integrity, data origin authentication and non-repudiation, but not used to provide confidentiality.

4.2 Domain Parameters

The domain parameters for ECDSA consists of a finite field F_q of characteristic p , a suitable elliptic curve E defined over F_q , and a base point $P \in E(F_q)$. The following is a detailed list of the domain parameters.

1. field size q , where $q = p$, an odd prime, or $q = 2^m$.
2. the field representation type for the elements of F_q
3. $a, b \in F_q$ which define an elliptic curve E over F_q ($y^2 = x^3 + ax + b$ in case $p > 3$, $y^2 + xy = x^3 + ax + b$ in case $p = 2$).
4. x_P and $y_P \in F_q$ which define the $P = (x_P, y_P)$ of prime order in $E(F_q)$ called the base point.
5. the order n of the point P , $n > 2^{160}$ and $n > 4\sqrt{q}$.
6. the cofactor $h = \#E(F_q)/n$

4.3 Elliptic Curve Digital Signature Algorithm

The key pairs are associated with a particular set of EC domain parameters. One must have the assurance that the domain parameters are valid before generating the keys.

The following key generation primitive is used by each party to generate the individual public and private key pairs [11, 27].

ECDSA Key Generation The user A follows these steps:

1. Select a random integer $d \in [1, n - 1]$.
2. Compute $Q = d \times P$.
3. The public and private keys of the user A are (E, P, n, Q) and d , respectively.

The other parties can check if the the public key is valid by;

1. Checking that $Q \neq \mathcal{O}$.
2. Checking that x_Q and y_Q are properly represented elements of F_q .
3. Checking that Q is on the elliptic curve defined by a and b .
4. Checking that $nQ = \mathcal{O}$.

If any of these checks fail the public key Q is invalid, otherwise Q is valid. The following procedure describes how to generate the signature.

ECDSA Signature Generation The user A signs the message m using the following steps.

1. Select a pseudorandom integer $k \in [1, n - 1]$.
2. Compute $k \times P = (x_1, y_1)$ and $r = x_1 \bmod n$.
If $x_1 \in GF(2^k)$, it is assumed that x_1 is represented as a binary number.
If $r = 0$ then go to Step 1.
3. Compute $k^{-1} \bmod n$.
4. Compute $s = k^{-1}(H(m) + d \cdot r) \bmod n$.
Here H is the secure hash algorithm SHA-1.
If $s = 0$ go to Step 1.
5. The signature for the message m is the pair of integers (r, s) .

ECDSA Signature Verification The user B verifies A 's signature (r, s) on the message m by applying the following steps:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $c = s^{-1} \bmod n$ and $H(m)$.
3. Compute $u_1 = H(m) \cdot c \bmod n$ and $u_2 = r \cdot c \bmod n$.
4. Compute $u_1 \times P + u_2 \times Q = (x_0, y_0)$ and $v = x_0 \bmod n$.
5. Accept the signature if $v = r$.

CHAPTER 5

INCOMPLETE MODULAR ARITHMETIC IN $GF(p)$

5.1 Introduction

The basic arithmetic operations (i.e., addition, subtraction, and multiplication) in the finite field $GF(p)$ have several applications in cryptography, such as decipherment operation of the RSA algorithm [28], the Diffie-Hellman key exchange algorithm [7], elliptic curve cryptography [15, 24], and the Digital Signature Standard including the Elliptic Curve Digital Signature Algorithm (ECDSA) [27]. These applications demand high-speed software implementations of the arithmetic operations in $GF(p)$ for $160 \leq \lceil \log_2(p) \rceil \leq 2048$. In this chapter, we describe a new method for obtaining high-speed software implementations of the arithmetic operations on microprocessors and general-purpose computers. The most important feature of this method is that it avoids bit-level operations which are slow on modern microprocessors. The algorithms proposed in this chapter perform word-level operations, trading them off for bit-level operations, and thus, resulting in much higher speeds. We provide the timing results of our implementations on a Pentium II computer, supporting our speedup claims.

5.2 Representation of the Numbers

The arithmetic of $GF(p)$ is also called modular arithmetic where the modulus is p . The elements of the field are the set of integers $\{0, 1, \dots, (p-1)\}$, and the arithmetic functions (addition, subtraction, and multiplication) takes two input operands from this set and produces the output which is also in this set. We are assuming that the modulus p is a k -bit integer where $k \in [160, 2048]$. A number in this range is

represented as an array of words, where each word is of length w . Most software implementations require that $w = 32$, however, w can be selected as 8 or 16 on 8-bit or 16-bit microprocessors.

In order to create a scalable implementation, we are not placing any restrictions on the prime p or its length k . The prime number does not need to be in any special form, as some methods require, for example, the method in [6] requires that $p = 2^k - c$. Furthermore, the length of the prime p does not need to be an integer multiple of the wordsize of the computer. We have the following definitions:

- k : The exact number of bits required to represent the prime modulus p , i.e., $k = \lceil \log_2 p \rceil$.
- w : The word-size of the representation (i.e., the computer). Usually, $w = 8, 16, 32$.
- s : The exact number of words required to represent the prime modulus p , i.e., $s = \lceil \frac{k}{w} \rceil$.
- m : The total number of bits in s words, i.e., $m = sw$.

Since our computers are capable of performing only two's complement binary arithmetic, we represent the numbers as unsigned binary numbers. A number from the field $GF(p)$ is represented as an s -word array of unsigned binary integers. We will use the notation $A = (A_{s-1}A_{s-2} \dots A_1A_0)$, where the words A_i for $i = 0, 1, \dots, (s-1)$ are unsigned binary numbers of length w . The most significant word (MSW) of A is A_{s-1} , while the least significant word (LSW) of A is A_0 . The bit-level representation of A is given as $A = (a_{k-1}a_{k-2} \dots a_1a_0)$. Similarly, the most significant bit (MSB) of A is a_{k-1} , while the least significant bit (LSB) of A is a_0 . We will use exactly s words to represent a number. If k is not an integer multiple of w , then we have $k = (s-1)w + u$ where $u < w$ is a positive integer, and thus, only the least significant u bits of the MSW of A_{s-1} are occupied, and the most significant $(w - u)$ bits are all zero.

A_{s-1}	A_{s-2}	\cdots	A_1	A_0
$\underbrace{0 \cdots 0}_{w-u} a_{(s-1)w+u-1} \cdots a_{(s-1)w}$	$a_{(s-1)w-1} \cdots a_{(s-2)w}$	\cdots	$a_{2w-1} \cdots a_w$	$a_{w-1} \cdots a_0$

5.2.1 Incompletely Reduced Numbers

This representation methodology has a shortcoming: it requires bit-level operations on the MSW in order to perform arithmetic, which affects the speed of the operations in software implementations. In order to overcome this shortcoming, we introduce the concept of *incomplete modular arithmetic*. In order to explain the mechanics of the method, we make the following definitions:

- Completely Reduced Numbers: the numbers from 0 to $(p - 1)$.

$$\mathbf{C} = \{\mathbf{0}, \mathbf{1}, \dots, (\mathbf{p} - \mathbf{1})\} .$$

- Incompletely Reduced Numbers: the numbers from 0 to $(2^m - 1)$.

$$\mathbf{I} = \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{p} - \mathbf{1}, \mathbf{p}, \mathbf{p} + \mathbf{1}, \dots, (2^{\mathbf{m}} - \mathbf{1})\} .$$

- Unreduced Numbers: the numbers from \mathbf{p} to $(2^m - 1)$.

$$\mathbf{U} = \{\mathbf{p}, \mathbf{p} + \mathbf{1}, \dots, (2^{\mathbf{m}} - \mathbf{1})\} .$$

Note that we have the following relationship between these sets:

$$\mathbf{C} \subset \mathbf{I}$$

$$\mathbf{U} \subset \mathbf{I}$$

$$\mathbf{U} = \mathbf{I} - \mathbf{C}$$

If $A \in \mathbf{C}$ and $2p < 2^k$, then A has an incompletely reduced equivalent $B \in \mathbf{I}$ such that $A = B \pmod{p}$. Thus, instead of working with A , we can also work with B in our arithmetic operations. The incompletely reduced numbers completely occupy s words as follows:

B_{s-1}	B_{s-2}	\cdots	B_1	B_0
$b_{sw-1} \cdots b_{(s-1)w}$	$b_{(s-1)w-1} \cdots b_{(s-2)w}$	\cdots	$b_{2w-1} \cdots b_w$	$b_{w-1} \cdots b_0$

When we perform arithmetic with incompletely reduced numbers, we do not need to perform bit-level operations on the MSW. All operations are word-level operations, and checks for carry bits are performed on the word boundaries, not within the words. Furthermore, we skip unnecessary reductions until the actual output is produced, in which case, we make sure that it belongs to \mathbf{C} . This representation yields high-speed implementations of the arithmetic operations.

An incompletely reduced number B can be converted to its completely reduced equivalent A by subtracting integer multiples of p from B as many times as necessary (until it is less than p).

5.2.2 Positive and Negative Numbers

The numbers in the range $[0, p - 1]$ as defined are positive numbers. The implementation of the subtraction operation requires a method of representation for negative numbers as well. We use the *least positive residues* which allow simultaneous representation of the positive and negative numbers modulo p . In this representation, the numbers are always left as positive, i.e., if the result of a subtraction operation is a negative number, then it is converted back to positive by adding p to it. For example, for $p = 7$, the operation $s = 3 - 4$ is performed as $s = 3 - 4 + 7 = 6$. The numbers from 0 to $(p - 1)/2$ can be interpreted as positive numbers modulo p , while the numbers from $(p - 1)/2 + 1$ to $p - 1$ can be interpreted as negative numbers modulo p . However, numbers are always kept as unsigned positive integers.

5.2.3 A Representation Example

We take the prime modulus as $p = 11 = (1011)$ and the word size as $w = 3$. Thus, we have $k = 4$ and $s = \lceil k/w \rceil = \lceil 4/3 \rceil = 2$, which gives $m = 2 \cdot 3 = 6$. The

completely reduced set of numbers is $\mathbf{C} = \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{9}, \mathbf{10}\}$, while the incompletely reduced set contain the numbers ranging from 0 to $(2^m - 1) = (2^6 - 1) = 63$ as $\mathbf{I} = \{\mathbf{0}, \mathbf{1}, \dots, \mathbf{62}, \mathbf{63}\}$. The incompletely reduced numbers occupy 2 words as $A = (A_1 A_0) = (a_5 a_4 a_3 \ a_2 a_1 a_0)$. For example, the decimal number 44 is represented as (101 100) in binary or (5 4) in octal.

An incompletely reduced equivalent of A is given as $B = A + i \cdot p$, where $B \in [0, 63]$ and i is a positive integer. For example, if $A = 5$, then the incompletely reduced equivalents of A are given as $\{5, 16, 27, 38, 49, 60\}$. The incompletely reduced representation is a redundant representation: we use the notation $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ to represent the residue class $\bar{5}$. We will show in the following sections that this redundant representation provides more efficient arithmetic in $GF(p)$.

5.3 Modular Addition

Since we use the incompletely reduced numbers, our numbers are allowed to grow as large as $2^m - 1$. The incompletely reduced representation avoids unnecessary reduction operations. We add the input operands as $X := A + B \pmod{p}$. If the result does not exceed 2^m , we do not perform reduction. This check is simple to perform: since $m = sw$, we are only checking to see if there is a carry-out from the MSW. We will use the notation

$$(c, S_i) := A_i + B_i + c \tag{5.1}$$

to denote the word-level addition operation which adds the 1-word numbers A_i and B_i and the 1-bit carry-in c , producing the outputs c and S_i such that c is the 1-bit carry-out and S_i is the 1-word sum. The addition algorithm is given below:

Modular Addition Algorithm

Inputs: $A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
 Auxiliary: $F = (F_{s-1} \cdots F_1 F_0)$
 Output: $X = (X_{s-1} \cdots X_1 X_0)$
 Step 1: $c := 0$
 Step 2: for $i = 0$ to $s - 1$
 Step 3: $(c, S_i) := A_i + B_i + c$
 Step 4: if $c = 0$ then return $X = (S_{s-1} \cdots S_1 S_0)$
 Step 5: $c := 0$
 Step 6: for $i = 0$ to $s - 1$
 Step 7: $(c, T_i) := S_i + F_i + c$
 Step 8: if $c = 0$ then return $X = (T_{s-1} \cdots T_1 T_0)$
 Step 9: $c := 0$
 Step 10: for $i = 0$ to $s - 1$
 Step 11: $(c, U_i) := T_i + F_i + c$
 Step 12: return $X = (U_{s-1} \cdots U_1 U_0)$

If the carry-out from the MSW is zero, the algorithm produces the correct result in Step 4 as $X = S = (S_{s-1} \cdots S_1 S_0)$. If the carry-out is one, we first ignore the carry-out and then correct the result. By ignoring the carry-out, we are essentially performing the operation $S := S - 2^m$. Since we need to perform modulo p arithmetic, we are allowed only add or subtract integer multiples of p , therefore, we need to correct the result as $T := (S - 2^m) + F$, where $F = (F_{s-1} \cdots F_1 F_0)$ is called *the correction factor for addition* and is defined as

$$F = 2^m - Ip, \quad (5.2)$$

where I is largest possible integer which brings F to the range $[1, p - 1]$, in other words, $I = \lfloor 2^m/p \rfloor$. The number F is precomputed and saved. By performing the operation $T := (S - 2^m) + F$, we essentially perform a modulo p reduction as

$$T := (S - 2^m) + F = S - 2^m + 2^m - Ip = S - Ip. \quad (5.3)$$

Thus, the result $X = T$ will be correct as a modular number after Step 8. However, there is a danger in performing the operation $T := S + F$ since this might also cause a carry-out from the MSW. The input operands A and B are arbitrary numbers, they can be as large as $2^m - 1$, which gives $S = 2^{m+1} - 2$. By ignoring the carry in Step 3, we obtain $S = 2^m - 2$. Therefore, $T := S + F$ in Step 4 may exceed 2^m , and we need to correct the result one more time, which is accomplished in Steps 9–11. This is the last correction we need to perform. There is no need for another correction after Step 11, since the maximum value of U is strictly less than 2^m .

$$U = (T - 2^m) + F = 2^m - 2 - 2^m + F = -2 + F \leq -2 + p - 1 < 2^m . \quad (5.4)$$

5.4 Addition Examples

Let $p = 11$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. We also compute F as

$$F = 2^m - \lfloor 2^m/p \rfloor \cdot p = 64 - \lfloor 2^6/11 \rfloor \cdot 11 = 64 - 5 \cdot 11 = 9 . \quad (5.5)$$

- We illustrate the addition of $\bar{4} = \{4, 15, 26, 37, 48, 59\}$ and $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ and using the incompletely reduced numbers 26 and 27.

$$\begin{aligned} S &= 26 + 27 \\ &= 53 \quad (c = 0 \text{ return Step 4}) \end{aligned}$$

The result is indeed correct since 53 is equivalent to $\bar{9} = \{9, 20, 31, 42, 53\}$.

- We give an addition example where the first correction (Steps 5–8) would be required. The addition of $\bar{4} = \{4, 15, 26, 37, 48, 59\}$ and $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ and using the incompletely reduced numbers 37 and 49 is such an example.

$$\begin{aligned} S &= 37 + 49 \\ &= 86 \quad (c = 1 \text{ Step 4}) \\ &= 86 - 64 \quad (\text{ignore carry Step 4}) \\ &= 22 \\ T &= 22 + 9 \quad (\text{correction Steps 5–7}) \\ &= 31 \quad (c = 0 \text{ return Step 8}) \end{aligned}$$

The result is indeed correct since 31 is equivalent to $\bar{9} = \{9, 20, 31, 42, 53\}$.

- We give an addition example where the second correction (Steps 10 and 11) would also be required. The addition of $\bar{6} = \{6, 17, 28, 39, 50, 61\}$ and $\bar{7} = \{7, 18, 29, 40, 51, 62\}$ using the incompletely reduced numbers 61 and 62 is such an example.

$$\begin{aligned}
 S &= 61 + 62 \\
 &= 123 && (c = 1 \text{ Step 4}) \\
 &= 123 - 64 && (\text{ignore carry Step 4}) \\
 &= 59 \\
 T &= 59 + 9 && (\text{correction Steps 5-7}) \\
 &= 68 && (c = 1 \text{ Step 8}) \\
 &= 68 - 64 && (\text{ignore carry Step 8}) \\
 &= 4 \\
 U &= 4 + 9 && (\text{correction Steps 9-11}) \\
 &= 13 && (\text{return Step 12})
 \end{aligned}$$

The result is indeed correct since 13 is equivalent to $\bar{2} = \{2, 13, 24, 35, 46, 57\}$.

5.5 Modular Subtraction

The subtraction is performed using two's complement arithmetic. The input operands are least positive residues represented using incompletely reduced representation. We will use the notation

$$(b, S_i) := A_i - B_i - b \tag{5.6}$$

to denote the word-level subtraction operation which subtracts the 1-word number B_i and the 1-bit borrow-in b from the 1-word number A_i , producing the outputs which are the 1-word number S_i and the 1-bit borrow-out b . The field subtraction algorithm for computing $X = A - B \pmod{p}$ is given below:

Modular Subtraction Algorithm

Inputs: $A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
 Auxiliary: $G = (G_{s-1} \cdots G_1 G_0)$ and $F = (F_{s-1} \cdots F_1 F_0)$
 Output: $X = (X_{s-1} \cdots X_1 X_0)$
 Step 1: $b := 0$
 Step 2: for $i = 0$ to $s - 1$
 Step 3: $(b, S_i) := A_i - B_i - b$
 Step 4: if $b = 0$ then return $X = (S_{s-1} \cdots S_1 S_0)$
 Step 5: $c := 0$
 Step 6: for $i = 0$ to $s - 1$
 Step 7: $(c, T_i) := S_i + G_i + c$
 Step 8: if $c = 0$ then return $X = (T_{s-1} \cdots T_1 T_0)$
 Step 9: $c := 0$
 Step 10: for $i = 0$ to $s - 1$
 Step 11: $(c, U_i) := T_i + F_i + c$
 Step 12: return $X = (U_{s-1} \cdots U_1 U_0)$

If $b = 0$ after Step 4, then the result is positive, and it is a properly reduced modular number. If $b = 1$, then the result is negative, we obtain the two's complement result, i.e., we essentially compute

$$S := A - B = A + 2^m - B . \quad (5.7)$$

The result S is in the range $[0, 2^m - 1]$, as required. However, it is incorrectly reduced, i.e., 2^m is added, and thus, we need to correct the result by adding $G = (G_{s-1} \cdots G_1 G_0)$ which is the *correction factor for subtraction* defined as

$$G = Jp - 2^m , \quad (5.8)$$

where J is the smallest integer which brings G to the range $[1, p-1]$, i.e., $J = \lceil 2^m/p \rceil$. It is easily proven that the sum of the correction factors for addition and subtraction, i.e., the sum of F and G , is equal to p since

$$F + G = 2^m - Ip + Jp - 2^m = (J - I)p = (\lceil 2^m/p \rceil - \lfloor 2^m/p \rfloor)p = p , \quad (5.9)$$

in other words, we have $G = p - F$ or $F = p - G$. The result S is corrected to obtain T in Steps of 5–8. After the correction of S in Step 8, we obtain

$$T = S + G = A + 2^m - B + Jp - 2^m = A - B + Jp . \quad (5.10)$$

Similar to the addition algorithm in Step 8, this correction might cause a carry from the MSW, requiring another correction which we need to take care of using F . This is accomplished in Steps 9–11. There is no need for another correction after Step 12, since the maximum value $S = (2^m - 1)$ gives

$$U \leq (2^m - 1) + G - 2^m + F = -1 + p < 2^m . \quad (5.11)$$

5.6 Subtraction Examples

Let $p = 11$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. We also compute G as

$$G = \lceil 2^m/p \rceil \cdot p - 2^m = \lceil 2^6/11 \rceil \cdot 11 - 64 = 6 \cdot 11 - 64 = 2 . \quad (5.12)$$

Since $F + G = p$, we could have also obtained G using $G = p - F = 11 - 9 = 2$.

- We illustrate the subtraction operation $S := 5 - 7$, where $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ and $\bar{7} = \{7, 18, 29, 40, 51, 62\}$, using the incompletely reduced equivalents 49 and 40.

$$\begin{aligned} S &= 49 - 29 \\ &= 20 \quad (b = 0 \text{ return Step 4}) \end{aligned}$$

The result is indeed correct since 20 is a incompletely reduced number represents the reduced number $\bar{9} = \{9, 20, 31, 42, 53\}$ which is equal to $5 - 7 = -2 = 9 \pmod{11}$.

- On the other hand, the same subtraction $S := 5 - 7$ operation using the

incompletely reduced equivalents 16 and 40 is performed as

$$\begin{aligned}
 S &= 16 - 40 \\
 &= -24 \quad (b = 1 \text{ Step 4}) \\
 &= 64 - 24 \quad (\text{two's complement Step 4}) \\
 &= 40 \\
 T &= 40 + 2 \quad (\text{correction Steps 5-8}) \\
 &= 42 \quad (c = 0 \text{ return Step 8})
 \end{aligned}$$

The incompletely reduced number 42 is also the correct result since it represents $\bar{9} = \{9, 20, 31, 42, 53\}$.

- We now give an addition example where the second correction (Step 9–12) would be required. The subtraction operation $5-6$, where $\bar{5} = \{5, 16, 27, 38, 49, 60\}$ and $\bar{6} = \{6, 17, 28, 39, 50, 61\}$, using the incompletely reduced numbers 49 and 50 is such an example:

$$\begin{aligned}
 S &= 49 - 50 \\
 &= -1 \quad (b = 1 \text{ Step 4}) \\
 &= 64 - 1 \quad (\text{two's complement Step 4}) \\
 &= 63 \\
 T &= 63 + 2 \quad (\text{correction Steps 5-8}) \\
 &= 65 \quad (c = 1 \text{ Step 8}) \\
 &= 65 - 64 \quad (\text{ignore carry Step 8}) \\
 &= 1 \\
 U &= 1 + 9 \quad (\text{correction Steps 9-11}) \\
 &= 10 \quad (\text{return Step 12})
 \end{aligned}$$

The result is indeed correct since 10 is equal to (-1) modulo 11.

5.7 Montgomery Modular Multiplication

The modular multiplication operation multiplies the input operands A and B and reduces the product modulo p , i.e., it computes $C := AB \pmod{p}$. The reduction

operation often requires bit-level shift-subtract operations [17]. We are interested in algorithms which requires word-level operations. Instead of the classical modular multiplication operation, we prefer to use the Montgomery modular multiplication [26] which computes

$$T := ABR^{-1} \pmod{p}, \quad (5.13)$$

where R is an integer with property $\gcd(R, p) = 1$. The selection of R is very important since it determines the algorithmic details and the speed of the Montgomery multiplication. Generally R is selected as the smallest power of 2 which is larger than p , i.e., $R = 2^k$, where $k = \lceil \log_2 p \rceil$. Thus, we have $1 < p < R$, however, $2p > R$. If k is not an integer multiple of the word-length w , this selection requires that we perform bit-level operations. Following the general premise of this chapter, we propose the use of $R = 2^m$, where $m = sw$, in order to avoid performing bit-level operations. In this case, 2^m may be several times larger than p , as it was also the case for the addition and subtraction algorithms presented in the previous sections.

We propose to use the Montgomery multiplication algorithm for incompletely reduced numbers, which receives two numbers A and B in the range $[0, 2^m - 1]$, and computes the result T which is also an incompletely reduced number in the range $[0, 2^m - 1]$, given by (5.13). In the high-level view, the Montgomery multiplication algorithm computes the result T using

$$T = \frac{AB + p(ABp' \bmod R)}{R}, \quad (5.14)$$

where p' is defined using the multiplicative inverse $R^{-1} \pmod{p}$ as

$$RR^{-1} - pp' = 1, \quad (5.15)$$

and computed using the extended Euclidean algorithm. The algorithm receives the inputs $A, B \in [0, R - 1]$ and computes the result T in (5.14). Since $A, B < R$, the result of the operation in (5.14) will have the maximum value

$$\frac{(R - 1)(R - 1) + p(R - 1)}{R} = \frac{(R - 1)(R - 1 + p)}{R} < R - 1 + p. \quad (5.16)$$

In other words, T as computed by (5.14) exceeds R only by an additive factor of p , therefore, we need to perform only a single subtraction to bring it back to the range $[0, R - 1]$.

The word-level description of the Montgomery multiplication involves a word-level multiplication operation, which we denote as

$$(c, T_j) := T_j + A_i \cdot B_j + c , \quad (5.17)$$

in which the new value of T_j and the new carry word c are computed using the old value of T_j and also the 1-word operands A_i and B_j , and the old carry word c . Here, all operands $A_i, B_j, T_j, c \in [0, 2^w - 1]$, i.e., they are all 1-word numbers. Since we have

$$(2^w - 1) + (2^w - 1) \cdot (2^w - 1) + (2^w - 1) = (2^w - 1)(2^w + 1) = 2^{2w} - 1 , \quad (5.18)$$

the result of the operation in (5.17) is a 2-word number represented using the 1-word numbers T_j and c .

The details of different Montgomery multiplication algorithms can be found in [19]. Here we describe an algorithm which computes T using the least significant word of the number p' defined in (5.15). Since $R = 2^{sw}$, we can reduce the equation (5.15) modulo 2^w , and obtain

$$-pp' = 1 \pmod{2^w} . \quad (5.19)$$

Let P_0 and Q_0 be the LSW of p and p' , respectively. Then, Q_0 is the negative of the multiplicative inverse of the LSW of p modulo 2^w , i.e.,

$$Q_0 = -P_0^{-1} \pmod{2^w} . \quad (5.20)$$

This 1-word number can be computed very quickly using a variation of the extended Euclidean algorithm given in [8]. The Montgomery multiplication algorithm for computing $T = AB2^{-m} \pmod{p}$ using Q_0 is given below.

Montgomery Modular Multiplication Algorithm

Inputs: $A = (A_{s-1} \cdots A_1 A_0)$ and $B = (B_{s-1} \cdots B_1 B_0)$
 Auxiliary: Q_0 and $p = (P_{s-1} \cdots P_1 P_0)$
 Output: $T = (T_{s-1} \cdots T_1 T_0)$
 Step 1: for $j = 0$ to $s - 1$
 Step 2: $T_j := 0$
 Step 3: for $i = 0$ to $s - 1$
 Step 4: $c := 0$
 Step 5: for $j = 0$ to $s - 1$
 Step 6: $(c, T_j) := T_j + A_i \cdot B_j + c$
 Step 7: $T_s := c$
 Step 8: $M := T_0 \cdot Q_0 \pmod{2^w}$
 Step 9: $c := (T_0 + M \cdot P_0) / 2^w$
 Step 10: for $j = 1$ to $s - 1$
 Step 11: $(c, T_{j-1}) := T_j + M \cdot P_j + c$
 Step 12: $(c, T_{s-1}) := T_s + c$
 Step 13: if $c = 0$ return $T = (T_{s-1} \cdots T_1 T_0)$
 Step 14: $b := 0$
 Step 15: for $j = 0$ to $s - 1$
 Step 16: $(b, T_j) := T_j - P_j - b$
 Step 17: return $T = (T_{s-1} \cdots T_1 T_0)$

The explanations and proofs of the steps are given below:

- In Steps 1 and 2, we clear the words of the result T to zero. The final result $T = AB2^{-m} \pmod{p}$ will be located in the s -word T at the end of the computation.
- The first part of the multiplication loop (Steps 3-7) computes a partial product T which is of length $s + 1$. For $i = 0$, this value is given as

$$T := A_0 \cdot B .$$

Since $A_0 \in [0, 2^{w-1}]$ and $B \in [0, 2^{m-1}]$, this value of T is less than equal to

$$2^{w-1} \cdot 2^{m-1} = 2^{w-1} \cdot 2^{sw-1} = 2^{(s+1)w-2} .$$

- In Steps 8-12, we are reducing T modulo p in such a way that it is now of length s words at the end of Step 12. This is accomplished using the following substeps:

- First, in Step 8, we multiply the LSW of T by Q_0 modulo 2^w . Recall that Q_0 is the LSW of p' or it is equal to $-P_0^{-1} \pmod{2^w}$. Thus, M (which is a 1-word number) is given as

$$M := T_0 \cdot Q_0 = T_0 \cdot (-P_0^{-1}) = -T_0 P_0^{-1} \pmod{2^w} .$$

- Then, in Step 9, we compute $T_0 + M \cdot P_0$ which is equal to

$$X := T_0 + M \cdot P_0 := T_0 + (-T_0 P_0^{-1}) P_0 .$$

Note that X is a 2-word number, however, the LSW of X is zero since

$$T_0 + (-T_0 P_0^{-1}) P_0 = 0 \pmod{2^w} .$$

Therefore, after the division by 2^w in Step 9, we obtain the 1-word carry c from the computation $T_0 + M \cdot P_0$.

- Then, in the remaining steps, i.e., in Steps 10-12, we are finishing the computation of $T + M \cdot P$. Since the LSW of the result is zero, we are also shifting the result by 1 word to the right (towards the least significant) in order to obtain the s -word number given by Equation (5.14).
- According to Equation (5.16), the result computed at the end of Step 12 can exceed $R - 1$ by at most p , and thus, a single subtraction will bring it back to the range $[0, R - 1]$. In Step 13, we check if the carry computed at the end of Step 12 is 1, i.e., if T exceeds $R - 1$. If there is no carry, then we return the result T in Step 13 as the final product.

- Otherwise, we perform a simple subtraction $T := T - p$ in order to bring back T to the range $[0, R - 1]$. The subtraction operation is accomplished in Steps 14-16, and the final product value is returned in Step 17.

Therefore, the Montgomery modular multiplication works even if the modulus $R = 2^{sw}$ is much larger than p , i.e., it need not be the smallest number of the form 2^i which is larger than p . While there may be several correction steps needed in the addition and subtraction operations, a single subtraction operation is sufficient for computing the Montgomery product $T = AB2^{-sw} \pmod{p}$.

The complete Montgomery multiplication and the incomplete Montgomery multiplication algorithms differ only slightly from one another. Algorithmically, these two algorithms are similar. Their main differences are in the way the input and output operands are specified:

- The radix R in the complete Montgomery multiplication algorithm is taken as 2^k , while the incomplete Montgomery multiplication algorithm uses the value 2^{sw} , therefore avoiding bit-level operations if k is not an integer multiple of w .
- The complete Montgomery multiplication algorithm requires that input operands be complete, i.e., numbers in the range $[0, p - 1]$, while the incomplete Montgomery multiplication algorithm requires that input operands be in the range $[0, 2^m - 1]$.
- The complete Montgomery multiplication algorithm computes the final result as a complete number, i.e., a number in the range $[0, p - 1]$, while the incomplete Montgomery multiplication algorithm computes the result in the range $[0, 2^m - 1]$.

5.8 Multiplication Examples

Let $p = 53$, $k = 4$, $w = 3$, $m = 6$, and $s = 2$. Since $p = 53 = (110101)$ and $P_0 = (101) = 5$, we compute $Q_0 = -P_0^{-1} \pmod{2^w}$ as

$$Q_0 = -5^{-1} \pmod{8} = -5 = 3 .$$

Also, we have $R = 2^m = 2^6 = 64$. Here are two multiplication examples:

- We illustrate the multiplication of $\bar{5} = \{5, 58\}$ and $\bar{7} = \{7, 60\}$ using the incompletely reduced numbers 58 and 60. Taking $A = 58 = (111\ 010)$ and $B = 60 = (111\ 100)$, we compute $T = A \cdot B \cdot R^{-1} \pmod{p}$ as follows:

- Step 3: $i = 0$
- Step 4,5,6 and $j = 0$: $(c, T_0) := A_0 \cdot B_0 = 2 \cdot 4 = 8 = (001\ 000)$.
- Step 5,6 and $j = 1$: $(c, T_1) := A_0 \cdot B_1 + c = 2 \cdot 7 + 1 = 15 = (001\ 111)$.
- Step 7: $T_2 = c = 1$. Therefore, we have $T = (001\ 111\ 000)$
- Step 8: $M = T_0 \cdot Q_0 = 0 \cdot 3 \pmod{8} = 0$.
- Step 9: $c = (T_0 + M \cdot P_0)/8 = (0 + 0 \cdot 5)/8 = 0$.
- Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 7 + 0 \cdot 6 + 0 = 7 = (000\ 111)$.
- Step 12: $(c, T_1) = T_2 + c = 1 + 0 = 1 = (000\ 001)$. We now have $T = (001\ 111)$.
- Step 3: $i = 1$
- Step 4,5,6 and $j = 0$: $(c, T_0) := T_0 + A_1 \cdot B_0 = 7 + 7 \cdot 4 = 35 = (100\ 011)$.
- Step 5,6 and $j = 1$: $(c, T_1) := T_1 + A_1 \cdot B_1 + c = 1 + 7 \cdot 7 + 4 = 54 = (110\ 110)$.
- Step 7: $T_2 = c = 6$. Therefore, we have $T = (110\ 110\ 011)$.
- Step 8: $M = T_0 \cdot Q_0 = 3 \cdot 3 \pmod{8} = 1$.
- Step 9: $c = (T_0 + M \cdot P_0)/8 = (3 + 1 \cdot 5)/8 = 1$.
- Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 6 + 1 \cdot 6 + 1 = 13 = (001\ 101)$.
- Step 12: $(c, T_1) = T_2 + c = 6 + 1 = 7 = (000\ 111)$. We now have $T = (111\ 101)$.
- Step 13: since $c = 0$, the result $T = (111\ 101)$ is returned.

The final result is $T = (111\ 101) = 61$ which is an incomplete number. The complete equivalent of 61 is 8 which is equal to $5 \cdot 7 \cdot 64^{-1} \pmod{53}$.

- We now illustrate the multiplication of $\bar{8} = \{8, 61\}$ and $\bar{10} = \{10, 63\}$ using the incompletely reduced numbers 61 and 63. Taking $A = 61 = (111\ 101)$ and $B = 63 = (111\ 111)$, we compute $T = A \cdot B \cdot R^{-1} \pmod{p}$ below. In this multiplication, the subtraction steps (Steps 14-17) are performed.
 - Step 3: $i = 0$
 - Step 4,5,6 and $j = 0$: $(c, T_0) := A_0 \cdot B_0 = 5 \cdot 7 = 35 = (100\ 011)$.
 - Step 5,6 and $j = 1$: $(c, T_1) := A_0 \cdot B_1 + c = 5 \cdot 7 + 4 = 39 = (100\ 111)$.
 - Step 7: $T_2 = c = 4$. Therefore, we have $T = (100\ 111\ 011)$
 - Step 8: $M = T_0 \cdot Q_0 = 3 \cdot 3 \pmod{8} = 1$.
 - Step 9: $c = (T_0 + M \cdot P_0)/8 = (3 + 1 \cdot 5)/8 = 1$.
 - Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 7 + 1 \cdot 6 + 1 = 14 = (001\ 110)$.
 - Step 12: $(c, T_1) = T_2 + c = 4 + 1 = 5 = (000\ 101)$. We now have $T = (101\ 110)$.
 - Step 3: $i = 1$
 - Step 4,5,6 and $j = 0$: $(c, T_0) := T_0 + A_1 \cdot B_0 = 6 + 7 \cdot 7 = 55 = (110\ 111)$.
 - Step 5,6 and $j = 1$: $(c, T_1) := T_1 + A_1 \cdot B_1 + c = 5 + 7 \cdot 7 + 6 = 60 = (111\ 100)$.
 - Step 7: $T_2 = c = 6$. Therefore, we have $T = (110\ 110\ 011)$.
 - Step 8: $M = T_0 \cdot Q_0 = 7 \cdot 3 \pmod{8} = 5$.
 - Step 9: $c = (T_0 + M \cdot P_0)/8 = (7 + 5 \cdot 5)/8 = 4$.
 - Step 10,11 and $j = 1$: $(c, T_0) = T_1 + M \cdot P_1 + c = 4 + 5 \cdot 6 + 4 = 38 = (100\ 110)$.
 - Step 12: $(c, T_1) = T_2 + c = 7 + 4 = 11 = (001\ 011)$.
 - Step 13: since $c = 1$, we execute the subtraction steps below.
 - Step 14: $b = 0$.
 - Step 15,16 and $j = 0$: $(b, T_0) = T_0 - P_0 - b = 6 - 5 - 0 = 1 = (000\ 001)$.

- Step 15,16 and $j = 1$: $(b, T_1) = T_1 - P_1 - b = 3 - 6 - 0 = -3 \pmod{8} = 5(000\ 101)$.
- Step 17: $T = (101\ 001) = 41$ is returned.

The final result is $T = (101\ 001) = 41$ which is a complete number. The final result 41 is equal to $8 \cdot 10 \cdot 64^{-1} \pmod{53}$.

5.9 Implementation Results and Conclusions

The incomplete addition, subtraction, and Montgomery multiplication algorithms together with their complete counterparts have been implemented on a 450-MHz Pentium II computer running Windows NT 4.0 operating system, with 256 megabytes of memory. The code is written entirely in C. The timings of the complete and incomplete routines are tabulated in Table 5.1 in microseconds.

Table 5.1. Incomplete and complete arithmetic timings in microseconds.

k	Addition			Subtraction			Multiplication		
	Complete	Incomplete	%	Complete	Incomplete	%	Complete	Incomplete	%
161	1.85	1.11	40	1.43	1.10	23	4.80	4.58	5
176	1.90	1.11	42	1.38	1.10	20	4.74	4.57	4
192	2.00	1.26	37	1.38	1.04	25	4.79	4.62	4
193	1.98	1.23	38	1.47	1.20	18	6.36	6.17	3
208	2.14	1.22	43	1.46	1.19	18	6.40	6.13	4
224	2.03	1.28	37	1.45	1.16	20	6.35	6.17	3
225	2.20	1.30	41	1.58	1.29	18	8.06	7.73	4
240	2.23	1.32	41	1.53	1.27	17	8.03	7.74	4
256	2.31	1.52	34	1.53	1.27	17	8.02	7.76	3

The speedup in percentage is obtained by subtracting the incomplete timing result from the complete timing result and then dividing it by the complete timing result. As can be seen from Table 5.1, the incomplete addition is 34–43 % faster than the complete addition for the range of k from 161 to 256. Similarly, the incomplete subtraction is 17–23 % faster than the complete subtraction. The reduction in the speedup for the subtraction operation as compared to the addition operation is mainly due to the fact that we have to perform more corrections than the addition operation. On the other hand, we obtain only a small speedup (3–5 %) for the incomplete Montgomery multiplication operation since the incomplete and complete Montgomery multiplication algorithms differ only slightly.

Furthermore, we have also implemented the ECDSA [27, 12] over the finite field $GF(p)$ in order to find out the performance impact of the incomplete arithmetic as compared to the complete arithmetic. The timing results of the ECDSA signature generation algorithm are given in Table 5.2 in milliseconds. These ECDSA timings are obtained without using any precomputation. We executed the ECDSA code several hundred times using two different random elliptic curve sets for bit lengths as specified in Table 5.2. Furthermore, the addition, subtraction, and multiplication timings shown in Table 5.1 are also obtained by running the ECDSA code and measuring the timings of the arithmetic operations in that context.

The implementation results have shown that the ECDSA algorithm can be made to execute 10–13 % faster using the incomplete modular arithmetic, which is very significant. Coupled with some machine-level programming, the ECDSA algorithm can be made significantly faster, as shown in the last column of Table 5.2.

Table 5.2. ECDSA signature generation timings in milliseconds over $GF(p)$.

k	C code only			C + Assembly
	Complete	Incomplete	%	Incomplete
161	13.6	12.0	12	5.3
176	14.8	12.9	13	5.8
192	16.5	14.7	11	6.6
193	20.8	18.4	12	8.5
208	22.6	19.7	13	9.1
224	23.7	21.1	11	9.7
225	29.8	26.5	11	12.2
240	31.1	27.9	10	12.8
256	34.2	30.8	10	14.0

In this chapter, we presented a new methodology for speeding up arithmetic operations, and shown that the new method provides up to 13 % speedup in the execution of the ECDSA algorithm over the field $GF(p)$ for the length of p in the range $161 \leq k \leq 256$. In this chapter, we applied the incomplete arithmetic only to the modular addition, the modular subtraction, and the Montgomery multiplication operations. A similar word-level methodology was described in [30] for the modular inverse and Montgomery modular inverse operations. These algorithms are also of type word-level, i.e., they avoid bit-level operation.

CHAPTER 6

DOUBLE MONTGOMERY MULTIPLICATION IN $GF(p^k)$

6.1 Introduction

Arithmetic in finite fields is the base of many public-key algorithms, including the Diffie-Hellman key exchange algorithm [7], elliptic curve cryptography [25, 15], and the Elliptic Curve Digital Signature Algorithm [27]. The overall performance of these cryptographic algorithms depend on the efficiency of the arithmetic performed in the underlying finite field.

Finite fields are represented with the notation $GF(p^k)$, where p is a prime and k is a positive integer. We know that there exists a field for each prime p and integer k [21]. Many cryptographic algorithms are implemented for the following two cases:

- p is a large prime number and k is equal to 1, represented with the notation $GF(p)$
- p is equal to 2 and k is positive number, represented with the notation $GF(2^k)$

The case for some large p (an 8-bit, 16-bit or even larger prime) and some positive integer k is also receiving attention, particularly, in the context of elliptic curve and hyperelliptic curve cryptographic algorithms [25, 15, 16]. The previous work of Bailey and Paar exploits the fact that software implementations of $GF(p^k)$ have advantages over $GF(p)$ for performing multiprecision arithmetic without carry propagation, and over $GF(2^k)$ for performing word size calculations supported by the microprocessor [1, 2]. They propose to use special forms for p and the generator polynomial to construct the extension field $GF(p^k)$. This choice results in a dramatic tradeoff between the performance of arithmetic in the extension field and the number of the

extension fields available to use. Most of the research conducted in this subject has advanced on that direction [22, 4].

We are proposing two new methods for modular multiplication in $GF(p^k)$. The first method is called the *Double Montgomery Multiplication* where we use the Montgomery multiplication algorithm both for the polynomial multiplication in the extension field and the coefficient multiplications in the subfield $GF(p)$. We obtained fast timing results using this method. The second method is an improvement over the Double Montgomery Multiplication method. We are using the incompletely reduced arithmetic concept introduced in the previous chapter to reduce the complexity for the subfield multiplications performed in $GF(p)$. The second method provides us a 46–69 % speedup over the first method on the 450-MHz Pentium II. Both methods have no restriction on the generator polynomial except it is monic and irreducible as expected. In addition, both algorithms restrict the prime p to have less bit-length than the word-length of the microprocessor. We provide the timing results of our implementations on a Pentium II and an ARM microprocessor, supporting our speedup claims.

6.2 Previous Work

To optimize the arithmetic in $GF(p^k)$, Bailey and Paar [1, 2] introduced the concept of the Optimal Extension Fields. In the definition of the OEF, p is a *pseudo-Mersenne* prime of the form $2^n \pm c$ for some $\log_2 c \leq \frac{1}{2}n$, and k is chosen such that an irreducible binomial $x^k - w$ exist so as to construct the field. These special forms of both p and the generator polynomial allow us to perform simple reduction steps in the extension field and the subfield. Furthermore, two special types of OEFs were introduced: In *OEF Type I* the pseudo-Mersenne prime was fixed as $p = 2^n \pm 1$, and in *OEF Type II* the generator polynomial was fixed as $x^k - 2$ to perform even simpler reduction.

Lee, Kim, and Lee [20] suggested the use of binomials for polynomial reduction, however, they do not seem to restrict the prime p . Their timing results are reported for a special prime $p = 2^{16} - 129$ for the field $GF(p^{11})$.

Lim and Hwang [22] placed a restriction on p to perform further optimizations on OEFs. They proposed to choose p so that multiplication results of two p -bit numbers can be accumulated many times without exceeding the computer's wordsize and eliminating reductions. They also used a Karatsuba-variant multiplication method, however, the timings of the Lim and Hwang implementation are better than those of Bailey and Paar on similar microprocessors.

Chung, Sim and Lee [4] implemented the OEF approach on a microcontroller with a DSP coprocessor support. A variant of the standard multiplication method was used instead of the Karatsuba multiplication.

In this chapter, we are proposing to implement the finite field arithmetic without limiting the prime characteristic p and the generator polynomial and still obtain comparable timing results to the arithmetic implemented in OEFs. The algorithms proposed in this chapter are more flexible, and provide scalable implementations of the elliptic curve cryptography in software, by not putting any limitations on p , k , the irreducible polynomial, and the sizes of these operands. The prime p is expected to fit into the word of the computer, which can be 8, 16, or 32 bits.

6.3 Polynomial Representation

The extension field $GF(p^k)$ can be constructed with a monic irreducible polynomial of degree k , which we call the generator polynomial and represent as

$$f(x) = x^k + \sum_{i=0}^{k-1} a_i x^i, \quad (6.1)$$

where $a_i \in GF(p)$. The field elements in $GF(p^k)$ can be represented as

$$a(x) = a_{k-1}x^{k-1} + \cdots + a_1x + a_0, \quad (6.2)$$

where $a_i \in GF(p)$. We choose p so that its bit-length is less than the word-length of the computer, which is denoted by w . We can represent each coefficient with one word (w bits), requiring k words (kw bits) for a field element.

The addition and subtraction of two field elements in $GF(p^k)$ is performed by adding or subtracting the coefficients of like terms modulo p , which requires one

subtraction of p if the sum is greater than p . However, there is no carry propagation between these k digits. Multiplication is performed by computing $c(x) = a(x) \cdot b(x) \bmod f(x)$, where $a(x), b(x) \in GF(p^k)$. The modular multiplication is the most important operation because it consumes more time and it is frequently used.

6.4 Double Montgomery Multiplication in $GF(p^k)$

The Montgomery multiplication concept is given in $GF(p)$ [26, 19] and $GF(2^k)$ [18]. The timing results obtained are demonstrating that Montgomery multiplication is efficient when arbitrary values of p and arbitrary generator polynomials are used. Based on this fact, the Double Montgomery multiplication algorithm proposed in this chapter uses the Montgomery multiplication method both in the extension field and the subfield.

Instead of computing $a(x) \cdot b(x)$ in $GF(p^k)$, we are proposing to compute $\overline{a(x)} \cdot \overline{b(x)} \cdot R(x)^{-1}$ in $GF(p^k)$, where $\overline{a(x)}, \overline{b(x)} \in GF(p^k)$ and $R(x)$ is a special fixed element in $GF(p^k)$. The terms $\overline{a(x)}$ and $\overline{b(x)}$ are called the residue forms of $a(x)$ and $b(x)$ where $\overline{a(x)} = a(x) \cdot R(x) \bmod f(x)$ and $\overline{b(x)} = b(x) \cdot R(x) \bmod f(x)$. We select $R(x) = x^k$ in order to obtain fast software implementations. This selection of $R(x)$ provides us with a nice form of residue arithmetic, which is demonstrated below. The result of the Double Montgomery multiplication is obtained in the residue form. We can demonstrate this as

$$\begin{aligned}
 \overline{c(x)} &= \overline{a(x)} \cdot \overline{b(x)} \cdot R(x)^{-1} \bmod f(x) \\
 &= a(x) \cdot R(x) \cdot b(x) \cdot R(x) \cdot R(x)^{-1} \bmod f(x) \\
 &= a(x) \cdot b(x) \cdot R(x) \bmod f(x) \\
 &= c(x) \cdot R(x) \bmod f(x)
 \end{aligned}$$

The extension field polynomial multiplication is very similar to the $GF(2^k)$ polynomial multiplication which was explained in [18]. For the sake of completeness, we are also explaining it here for $GF(p^k)$. The Montgomery multiplication method requires that $R(x)$ and $f(x)$ are relatively prime. This requirement always holds, due to the

fact that $f(x)$ is irreducible in $GF(p^k)$ and $R(x) < f(x)$. Since $R(x)$ and $f(x)$ are relatively prime, there exist two polynomials $R(x)^{-1}$ and $f'(x)$ having the property that

$$R(x)R(x)^{-1} - f(x)f'(x) = 1, \quad (6.3)$$

where $R(x)^{-1}$ is the inverse of $R(x)$ modulo $f(x)$. The values of $R(x)$ and $f'(x)$ can be calculated using the extended Euclidean algorithm [21]. The general Double Montgomery algorithm in $GF(p^k)$ without exposing the subfield arithmetic in $GF(p)$ is given below.

DOUBLE MONTGOMERY MULTIPLICATION ALGORITHM

Inputs: $a(x), b(x), R(x), f'(x)$

Output: $c(x)$ such that $c(x) = a(x) \cdot b(x) \cdot R(x)^{-1} \bmod f(x)$

Step 1: $t(x) := a(x) \cdot b(x)$

Step 2: $u(x) := t(x) \cdot f'(x) \bmod R(x)$

Step 3: $c(x) := [t(x) + u(x) \cdot f(x)]/R(x)$

Step 4: return $c(x)$

We can prove the correctness of the above algorithm with similar calculations as in [18]. The equation $u(x) = t(x) \cdot f'(x) \bmod R(x)$ implies that there exists a polynomial $K(x)$ over $GF(p^k)$ such that

$$u(x) = t(x) \cdot f'(x) + K(x) \cdot R(x). \quad (6.4)$$

We substitute Equation (6.4) into the equation for $c(x)$, which is defined as Step 3 in the algorithm.

$$\begin{aligned} c(x) &= \frac{1}{R(x)}[t(x) + u(x)f(x)] \\ &= \frac{1}{R(x)}[t(x) + t(x)f'(x)f(x) + K(x)R(x)f(x)]. \end{aligned}$$

Furthermore, we have $f(x)f'(x) = R(x)R(x)^{-1} - 1$ according to Equation (6.3). We substitute this into the equation for $c(x)$, and obtain

$$c(x) = \frac{1}{R(x)}[t(x) + t(x)[R(x)R(x)^{-1} - 1] + K(x)R(x)f(x)]$$

$$\begin{aligned}
&= \frac{1}{R(x)} [t(x)R(x)R(x)^{-1} + K(x)R(x)f(x)] \\
&= t(x)R(x)^{-1} + K(x)f(x) \\
&= a(x)b(x)R(x)^{-1} \bmod f(x) .
\end{aligned}$$

The result proves the correctness of the given algorithm. Similar to the $GF(2^k)$ case, the final subtraction step required in the $GF(p)$ algorithm is not required in the $GF(p^k)$ algorithm. The degree of the polynomial $c(x)$ computed by this algorithm is less than or equal to $k - 1$, which is less than the degree of $f(x)$. In Step 3, we realize that the degree of $c(x)$ can be calculated from $(t(x)/R(x)) = 2k - 2 - k = k - 2$ or $(u(x)n(x)/R(x)) = k - 1 + k - k = k - 1$, which proves the previous statement.

The algorithm involves a regular multiplication at Step 1, a modulo $R(x)$ multiplication in Step 2, and a regular multiplication followed by a division with $R(x)$ in Step 3. Since $R(x) = x^k$, the reduction part of the modular multiplication and the division operation can be performed very fast. The reduction step of the modular multiplication is performed by ignoring the terms which have larger degrees than or equal degree to x^k . The division by $R(x)$ is performed by shifting the polynomial to the right by k terms. The computation of $f'(x)$ seems to be an overhead, but the word-level algorithm requires only the calculation of the least significant word $f'_0(x)$ instead the whole $f'(x)$ [8].

In order to implement the Double Montgomery Multiplication method on the Intel Pentium II and ARM processors, we designed a word-level algorithm based on the Coarsely Integrated Operand Scanning (CIOS) method given in [19] which seems to be the fastest Montgomery multiplication method. The CIOS method is integrating the multiplication and reduction steps by alternating between iterations of the outer loops for multiplication and reduction. The alternative methods for the Montgomery multiplication described in [19] could be preferable for different classes of processors. The compilers for Pentium and ARM processors support 32 bit and 64 bit numbers. We are using the term *word* to refer to 32 bit numbers and the term *double-word* to refer to 64 bit numbers. The most significant part of a double-word is called *high-word* while the least significant part is called *low-word*.

Before introducing the word-level algorithm in detail, we will explain the multiplication performed in the subfield. A Montgomery multiplication operation is performed for the multiplication of the coefficients in $GF(p)$. We know that the multiplication of two coefficients will produce a term that can reach the size of a double-word. We are reducing this term modulo p within one iteration using the Montgomery reduction method. For this multiplication, we define $r = 2^w$ and $p'_0 = -p^{-1} \bmod 2^w$, where w stands for the word-length of the computer, as defined previously. We can summarize this reduction as adding a multiple of p to the term that generates a zero in the low-word of this summation and dividing by r . The algorithm for the subfield multiplication is as follows.

SUBFIELD MONTGOMERY MULTIPLICATION ALGORITHM

Inputs: a, b, p, p'_0 , and r
Output: c such that $c = a \cdot b \cdot r^{-1} \bmod p$
Step 1: $T := a \cdot b$
Step 2: $u := \text{LW}(T) \cdot p'_0 \bmod r$
Step 3: $c := \text{HW}(T + u \cdot p)$
Step 4: if $c \geq p$ then $c := c - p$
Step 5: return c

Note that the computation of c is equivalent to $c := [T + u \cdot p]/r$ and the term T is a double-word. The high-word and low-word of a double-word (**dw**) are addressed with the keywords HW and LW followed by the term in parenthesis. Using the same notation we are giving the word-level algorithm for the whole multiplication below.

WORD-LEVEL DOUBLE MONTGOMERY MULTIPLICATION ALGORITHM

Inputs: $a(x) = (a_{k-1} \cdots a_1 a_0)$, $b(x) = (b_{k-1} \cdots b_1 b_0)$,
 $f(x) = (f_k \cdots f_1 f_0)$, p'_0 , and f'_0
Auxiliary $t = (t_k \cdots t_1 t_0)$
Output: $c(x) = (c_{k-1} \cdots c_1 c_0)$
Step 1: for $i = 0$ to $k - 1$ do
Step 2: for $j = 0$ to $k - 1$ do

Step 3: $\mathbf{dw1} = \mathbf{a}_i * \mathbf{b}_j$
 Step 4: $m = \text{LW}(\mathbf{dw1}) * \mathbf{p}'_0$
 Step 5: $\mathbf{dw2} = \mathbf{m} * \mathbf{p}$
 Step 6: $\mathbf{dw3} = \text{HW}(\mathbf{dw1}) + \text{HW}(\mathbf{dw2}) + \mathbf{t}_j + \mathbf{1}$
 Step 7: if $p < \mathbf{dw3}$ then
 Step 8: if $(2p) < \mathbf{dw3}$ then $t_j = \text{LW}(\mathbf{dw3} - \mathbf{2p})$
 Step 9: else $t_j = \text{LW}(\mathbf{dw3} - \mathbf{p})$
 Step 10: else $t_j = \text{LW}(\mathbf{dw3})$
 Step 11: $\mathbf{dw1} = \mathbf{t}_0 * \mathbf{f}'_0$
 Step 12: $m = \text{LW}(\mathbf{dw1}) * \mathbf{p}'_0$
 Step 13: $\mathbf{dw2} = \mathbf{m} * \mathbf{p}$
 Step 14: $\mathbf{dw3} = \text{HW}(\mathbf{dw1}) + \text{HW}(\mathbf{dw2}) + \mathbf{1}$
 Step 15: if $p < \mathbf{dw3}$ then $m1 = \text{LW}(\mathbf{dw3} - \mathbf{p})$
 Step 16: else $m1 = \text{LW}(\mathbf{dw3})$
 Step 17: for $j = 0$ to $k - 1$ do
 Step 18: $\mathbf{dw1} = \mathbf{m1} * \mathbf{f}_{j+1}$
 Step 19: $m = \text{LW}(\mathbf{dw1}) * \mathbf{p}'_0$
 Step 20: $\mathbf{dw2} = \mathbf{m} * \mathbf{p}$
 Step 21: $\mathbf{dw3} = \text{HW}(\mathbf{dw1}) + \text{HW}(\mathbf{dw2}) + \mathbf{t}_{j+1} + \mathbf{1}$
 Step 22: if $p < \mathbf{dw3}$ then
 Step 23: if $(2p) < \mathbf{dw3}$ then $t_j = \text{LW}(\mathbf{dw3} - \mathbf{2p})$
 Step 24: else $t_j = \text{LW}(\mathbf{dw3} - \mathbf{p})$
 Step 25: else $t_j = \text{LW}(\mathbf{dw3})$
 Step 26: for $i = 0$ to $k - 1$ do
 Step 27: $c_i = t_i$
 Step 28: return $c(x)$

The word-level algorithm consists of an outer loop, which scans through the coefficients of one of the polynomials, and two inner loops for the partial multiplication and the reduction operations. The first inner loop starts at Step 2 and finishes

at Step 10. In Step 3 the coefficient a_i is multiplied with all other coefficients of polynomial $b(x)$ through the loop. The multiplication results, which are stored in **dw1**, can be double-word size at most. We need to reduce this values to one-word numbers. In Step 4 and Step 5 we are calculating the multiple of p which generates a zero low-word when added to **dw1**.

In Step 6 we perform the addition of the terms **dw1** and the multiple of p . We add only the high-words of these terms and the number one because we know that the low-words will add up to zero producing a carry. Discarding the zero low-word also means dividing by r , which is required in the algorithm. The term t_j accumulates the partial results, which is set to 0 initially. Within the first execution of the inner loop the t_j terms are equal to zero due to the 0 initialization. Based on the Montgomery multiplication algorithm we know that the sum of the two high-words and the number one is smaller than $2p$. Normally we subtract p if the sum is grater than p to reduce the result. But we have to consider the t_j terms for the future iterations of the inner loop. They will be smaller than p due to the reduction steps in the inner loops. This means the **dw3** term in Step 6 can take a value between 0 and $3p - 2$. In the rest of this inner loop we are reducing the term **dw3** modulo p by subtracting p or $2p$ depending to its value.

We realize that the coefficients need to be in residue form to calculate a result that is also in residue form. At this point, we have a partial result that needs to be divided by x so that we can continue to accumulate the other partial results without increasing the space requirements. This is done by adding a multiple of the generator polynomial to the partial result that produces a zero at the last term. By discarding this last term we divide by x . This is the polynomial level Montgomery multiplication which we explained previously.

In Step 11 we are calculating the value needed to obtain the multiple of the generator polynomial. f'_0 is calculated as $f'_0 = -f_0^{-1}r^2 \bmod 2^W$. The result stored in **dw1** has r^3 as a factor and reaches the size of a double-word. We need to reduce it to a word size number to be able to use it within the second inner loop for further multiplications. From Step 12 to Step 16 the number in **dw1** is reduced and the

result is written into variable $m1$. This is similar what is done in the first inner loop, except we don't have any terms for accumulation. We note that the factor r^3 is reducing to r^2 because of the Montgomery reduction applied within these steps.

The second inner loop starts at Step 17 and ends at Step 25. It is very similar to the first inner loop except the accumulation in Step 21 is done in such a way that we divide the partial result by x . We are shifting the polynomial by one term to the right by using the t_{j+1} in Step 21 and writing the result to t_j in Step 24 and 25. The f_{j+1} term in Step 18 is representing the coefficients of the generator polynomial. These terms are not in residue form because other arithmetic functions like modular addition have to use generator polynomial. This is the reason we left the r^2 factor within the $m1$ term which was calculated before entering the second inner loop. The outer loop iterates k times producing a polynomial which has a degree of $k - 1$.

In Steps 26 through 28 we copy the result as output and finish the Double Montgomery Multiplication algorithm. We can use the Double Montgomery Multiplication algorithm to calculate the residue form of a given polynomial in $GF(p^k)$. First we should calculate $x^{2k} \bmod f(x)$ and multiply each coefficient with r^2 if $GF(p)$. Multiplying with this polynomial will give us the other polynomial in residue form.

In order to calculate the complexity of the Double Montgomery Algorithm, we are counting the number of multiplications, additions memory read-writes. Table 6.1 demonstrates the steps and operation counts in detail.

Using double-word variables to store and retrieve numbers does not require extensive memory access cycles. The values they hold are temporary values which are discarded within a couple of processor cycles. We considered storing and retrieving to arrays as memory access and assumed the worst case situations for the if clauses. The Double Montgomery multiplication method requires $6k^2 + 3k$ multiplication, $8k^2 + 3k$ addition, $4k^2 + 3k$ read, and $2k^2 + k$ write operations.

Table 6.1. The operation counts for DMM.

	Mult	Add	Read	Write
for i=0 to k-1 do	-	-	-	-
for j=0 to k-1 do	-	-	-	-
dw1 = a[i]*b[j]	k^2	-	$k^2 + k$	-
m = LW(dw1)*p0'	k^2	-	-	-
dw2 = m*p	k^2	-	-	-
dw3 = HW(dw1)+HW(dw2)+t[j]+1	-	$3k^2$	k^2	-
if p<dw3 then	-	-	-	-
if 2p<dw3 then t[j] = LW(dw3-2p)	-	k^2	-	k^2
else t[j] = LW(dw3-p)	-	-	-	-
else t[j] = LW(dw3)	-	-	-	-
dw1 = t[0]*f0'	k	-	k	-
m = LW(dw1)*p0'	k	-	-	-
dw2 = m*p	k	-	-	-
dw3 = HW(dw1)+HW(dw2)+1	-	$2k$	-	-
if p<dw3 then m1 = LW(dw3-p)	-	k	-	-
else m1 = LW(dw3)	-	-	-	-
for j=0 to k-1 do	-	-	-	-
dw1 = m1*f[j+1]	k^2	-	k^2	-
m = LW(dw1) * p0'	k^2	-	-	-
dw2 = m*p	k^2	-	-	-
dw3 = HW(dw1)+HW(dw2)+t[j+1]+1	-	$3k^2$	k^2	-
if p<dw3 then	-	-	-	-
if 2p<dw3 then t[j] = LW(dw3-2p)	-	k^2	-	k^2
else t[j] = LW(dw3-p)	-	-	-	-
else t[j] = LW(dw3)	-	-	-	-
for i=0 to k-1 do	-	-	-	-
c[i] = t[i]	-	-	k	k
return c	-	-	-	-
	$6k^2 + 3k$	$8k^2 + 3k$	$4k^2 + 3k$	$2k^2 + k$

6.5 Incomplete Reduction in $GF(p^k)$

The method we are introducing in this section is an improvement over the Double Montgomery multiplication method. We are reducing the complexity by eliminating the reductions performed in the subfield, using the incomplete modular arithmetic concept explained in the previous chapter. The coefficients of the polynomials are allowed to be in the range of $[1, 2^W - 1]$. The double-word numbers obtained from the coefficient multiplications are allowed to grow as large as $2^{2W} - 1$. The incompletely reduced representation avoids unnecessary reduction operations. If the number does not exceed $2^{2W} - 1$, we do not perform a reduction. This check is simple to perform, we are only checking to see if there is a carry-out from the double-word. We are using the notation

$$(c, \mathbf{dw}) = a_i * b_j + \mathbf{t}_j \quad (6.5)$$

to denote the word-level multiplication and accumulation operation which multiplies the two one-word numbers a_i and b_j and accumulates the double-word \mathbf{t}_j , producing the outputs c and \mathbf{dw} , such that c is the 1-bit carry-out and \mathbf{dw} is the double-word result. If the carry-out is zero, the result is correct and we can further accumulate on this number. If the carry out is 1, we first ignore the carry-out and then correct the result. By ignoring the carry-out, we are essentially performing the operation $\mathbf{dw} = \mathbf{dw} - 2^{2W}$. Since we need to perform modulo p arithmetic, we are allowed only add or subtract integer multiples of p , therefore, we need to correct the result as $\mathbf{dw} = \mathbf{dw} - 2^{2W} + \mathbf{CF}$, where \mathbf{CF} is called the correction factor and is defined as

$$\mathbf{CF} = 2^{2W} - Ip, \quad (6.6)$$

where I is the largest possible integer which brings \mathbf{CF} to the range $[1, p - 1]$. The number \mathbf{CF} is precomputed and saved. By performing the operation $\mathbf{dw} = \mathbf{dw} - 2^{2W} + \mathbf{CF}$, we essentially perform a modulo p reduction as

$$\mathbf{dw} = \mathbf{dw} - 2^{2W} + \mathbf{CF} = \mathbf{dw} - 2^{2W} + 2^{2W} - Ip = \mathbf{dw} - Ip. \quad (6.7)$$

The result will be correct as a modular number. If \mathbf{CF} and the other numbers were as large as $2^{2W} - 1$, there could be a possibility that adding the \mathbf{CF} value would cause

another carry-out requiring similar steps to correct the result for a second time. But as we will explain later, the algorithm does not require a second correction. At the end of the algorithm, all of the double-word terms will be reduced to one-word term using the Montgomery reduction method. These one-word terms will be left in the range of $[1, 2^W - 1]$.

The incomplete modular arithmetic method allows the coefficients to be as large as $2^W - 1$. We do not need to reduce the results modulo p . We need to take care of the carry-overs generated during the addition of coefficients in a similar way. The polynomial multiplication in the extension field $GF(p^k)$ is exactly the same as the Double Montgomery multiplication method. The polynomials and their coefficients need to be in the residue form requiring the terms r and $R(x)$ having the same values as in the previous algorithm. Using the same notations as before we are giving the word-level algorithm for the modified multiplication below.

WORD-LEVEL INCOMPLETE MONTGOMERY MULTIPLICATION ALGORITHM

Inputs: $a(x) = (a_{k-1} \cdots a_1 a_0)$, $b(x) = (b_{k-1} \cdots b_1 b_0)$,

$f(x) = (f_k \cdots f_1 f_0)$, p'_0 , f'_0 , and CF

Auxiliary $\mathbf{t} = (\mathbf{t}_k \cdots \mathbf{t}_1 \mathbf{t}_0)$

Output: $c(x) = (c_{k-1} \cdots c_1 c_0)$

Step 1: for $i = 0$ to $k - 1$ do

Step 2: for $j = 0$ to $k - 1$ do

Step 3: $(c, \mathbf{dw}) = \mathbf{a}_i * \mathbf{b}_j + \mathbf{t}_j$

Step 4: if $c = 1$ then $\mathbf{t}_j = \mathbf{dw} + \mathbf{CF}$

Step 5: else $\mathbf{t}_j = \mathbf{dw}$

Step 6: $m = LW(\mathbf{t}_0) * \mathbf{p}'_0$

Step 7: $\mathbf{dw} = \mathbf{m} * \mathbf{p}$

Step 8: $(c, \mathit{word}) = HW(\mathbf{t}_0) + \mathbf{HW}(\mathbf{dw}) + 1$

Step 9: if $c = 1$ then $\mathit{word} = \mathit{word} - p$

Step 10: $\mathbf{dw} = \mathit{word} * \mathbf{f}'_0$

Step 11: $m = LW(\mathbf{dw}) * \mathbf{p}'_0$

Step 12: $\mathbf{dw1} = \mathbf{m} * \mathbf{p}$
 Step 13: $(c, word) = HW(\mathbf{dw}) + \mathbf{HW}(\mathbf{dw1}) + \mathbf{1}$
 Step 14: if $c = 1$ then $word := word - p$
 Step 15: for $j = 0$ to $k - 1$ do
 Step 16: $(c, \mathbf{dw}) = \mathbf{word} * \mathbf{f}_{j+1} + \mathbf{t}_{j+1}$
 Step 17: if $c = 1$ then $\mathbf{t}_j = \mathbf{dw} + \mathbf{CF}$
 Step 18: else $\mathbf{t}_j = \mathbf{dw}$
 Step 19: for $i = 0$ to $k - 1$ do
 Step 20: $m = LW(\mathbf{t}_j) * \mathbf{p}'_0$
 Step 21: $\mathbf{dw} = \mathbf{m} * \mathbf{p}$
 Step 22: $(c, word) = HW(\mathbf{t}_j) + \mathbf{HW}(\mathbf{dw}) + \mathbf{1}$
 Step 23: if $c = 1$ then $word = word - p$
 Step 24: $c_j = word$
 Step 25: return $c(x)$

Similar to the previous one, this word-level algorithm has a large loop which scans through the coefficients of one of the polynomials, two inner loops for the partial multiplication, and the reduction operations. The final loop scans through the coefficients of the result and reduces them to one-word numbers.

The first inner loop starts at Step 2 and finishes at Step 5. In Step 3 the coefficient a_i is multiplied with all other coefficients of polynomial $b(x)$ through the loop. The accumulated partial values are added to the result within this step. The final result in \mathbf{dw} can be larger than a double-word. We can demonstrate this as follows:

$$a_i * b_j + \mathbf{t}_j = (2^{\mathbf{W}} - \mathbf{1}) * (2^{\mathbf{W}} - \mathbf{1}) + 2^{2\mathbf{W}} - \mathbf{1} = (2^{2\mathbf{W}+1} - 2^{\mathbf{W}+1}) > (2^{2\mathbf{W}} - \mathbf{1}) .$$

In the following two steps we are ignoring the carry if present which means subtracting $2^{2\mathbf{W}}$ from the result and adding CF . Thus, the result \mathbf{dw} will now be less than $2^{2\mathbf{W}}$ since

$$(2^{2\mathbf{W}+1} - 2^{\mathbf{W}+1}) - 2^{2\mathbf{W}} + 2^{\mathbf{W}} - \mathbf{1} = 2^{2\mathbf{W}} - 2^{\mathbf{W}} - \mathbf{1} < 2^{2\mathbf{W}} .$$

This proves that no second carry will be produced by adding the correction factor CF . At this point, we have a partial result that lies within a double-word array and needs to be divided by x so that we can continue to accumulate the other partial results without increasing the space requirements. Between Step 6 and Step 14 we are reducing the last coefficient to a one-word number and using this number to calculate the multiple of the generator polynomial that produces a zero in the last term when added to the partial result.

The second inner loop starts at Step 15 and finishes at Step 18. In Step 16 the *word* variable is multiplied with the coefficients of generator polynomial $f(x)$. The accumulated partial values are added to the result within this step. Similar to Step 3, the final result in **dw** can be larger than a double-word. The proof for Step 3 applies for Step 16 as well. Finally, the double-word representations are reduced to one-word number within the loop that starts at step 19 and ends at Step 24.

The Incomplete Double Montgomery multiplication method requires $2k^2 + 7k$ multiplications, $4k^2 + 9k$ additions, $4k^2 + 5k$ reads and $2k^2 + k$ writes. Its steps and operation counts are shown in Table 6.2.

Table 6.2. The operation counts for IDMM.

	Mult	Add	Read	Write
for i=0 to k-1 do	-	-	-	-
for j=0 to k-1 do	-	-	-	-
(c,dw) = a[i]*b[j]+t[j]	k^2	k^2	$2k^2 + k$	-
if c=1 then t[j] = dw+CF	-	k^2	-	k^2
else t[j] = dw	-	-	-	-
m = LW(t[0])*p0'	k	-	k	-
dw = m*p	k	-	-	-
(c,word) = HW(t[0])+HW(dw)+1	-	$2k$	k	-
if c=1 then word = word-p	-	k	-	-
dw = word*f0'	k	-	-	-
m = LW(dw)*p0'	k	-	-	-
dw1 = m*p	k	-	-	-
(c,word) = HW(dw)+HW(dw1)+1	-	$2k$	-	-
if c=1 then word = word-p	-	k	-	-
for j=0 to k-1 do	-	-	-	-
(c,dw) = word*f[j+1]+t[j+1]	k^2	k^2	$2k^2$	-
if c=1 then t[j] = dw+CF	-	k^2	-	k^2
else t[j] = dw	-	-	-	-
for i=0 to k-1 do	-	-	-	-
m = LW(t[j])*p0'	k	-	k	-
dw = m*p	k	-	-	-
(c,word) = HW(t[j])+HW(dw)+1	-	$2k$	k	-
if c=1 then word = word-p	-	k	-	-
c[j] = word	-	-	-	k
return c	-	-	-	-
	$2k^2 + 7k$	$4k^2 + 9k$	$4k^2 + 5k$	$2k^2 + k$

6.6 Implementation Results

The Double Montgomery multiplication algorithm and its incomplete version have been implemented on a 450-MHz Pentium II computer running the Windows 2000 operating system with 256 megabytes of memory. We also obtained timings on the ARM SDK v2.50 platform for the 80-MHz ARM7TDMI microprocessor. The codes were written in the assembly languages. The timings of the two methods for two different platforms are tabulated in Table 6.3 in microseconds. In Table 6.3, the acronyms DMM and IDMM stand for Double Montgomery Multiplication and Incomplete Double Montgomery Multiplication, respectively.

The speedup in percentage is obtained by subtracting the incomplete timing result from the complete timing result and then dividing it by the complete timing result. As can be seen from Table 6.3, choosing a prime p close to the word boundary and using smaller k values gives better results than using a prime p with less bit length and larger k values when the overall bit length is fixed. The number of operations to perform the multiplication depends on the k value.

For the ARM platform the incomplete multiplication version is 18–30 % faster than the complete multiplication for overall bit lengths from 160 to 512. Similarly, the Pentium platform has a speedup of 46–69 % for the same measurements. The Pentium cash memory is an important factor in obtaining better speedup values than the ARM platform.

The timing results for multiplication in OEFs obtained on a 233-Mhz Pentium/MMX processor are given in [2]. According to those timing results, the Karatsuba-variant multiplication code written in C for $p = 2^{31} - 1$ and $f(x) = x^6 - 7$ executes in 4.6 microseconds. This corresponds to the operands of bit length 192. Our method offers several choices for 192 bit operands, as emphasized in bold font in Table 6.3. On a 450-MHz Pentium II, the fastest speed we obtain is 3.4 microseconds using assembly language programming, where p is a 32-bit prime and $k = 6$. Furthermore, for the same platform and the same values of p and $f(x)$, we executed the IDMM code (also written in C) and obtained 11.4 microseconds.

Table 6.3. Timings of the DMM and IDMM in two different platforms in microseconds.

Field			ARM7TDMI (80 MHz)			Pentium II (450 MHz)		
p	k	bits	DMM	IDMM	%	DMM	IDMM	%
8 bits	20	160	339.8	241.5	29	59.0	18.5	68
8 bits	22	176	409.9	289.7	29	71.4	21.7	69
8 bits	24	192	486.0	342.4	29	84.1	25.4	69
16 bits	10	160	95.2	69.2	27	15.3	5.8	62
16 bits	11	176	114.5	82.3	28	18.7	6.7	64
16 bits	12	192	135.6	96.6	29	21.5	7.9	63
16 bits	14	224	183.0	128.6	30	29.2	10.1	65
16 bits	16	256	237.9	165.2	30	37.9	12.5	67
32 bits	5	160	26.7	21.9	18	4.4	2.3	47
32 bits	6	192	37.5	29.8	21	6.3	3.4	46
32 bits	7	224	50.3	38.9	22	8.4	4.3	48
32 bits	8	256	64.9	49.3	24	11.1	5.3	52
32 bits	9	288	81.5	60.9	25	13.5	7.2	46
32 bits	10	320	99.8	73.7	26	16.5	8.6	47
32 bits	13	416	166.6	119.9	28	28.7	14.2	50
32 bits	16	512	250.4	177.3	29	42.2	17.8	58

While our method is slower in this particular case, it is more general (any p and any $f(x)$), and offers several choices based for the platform (for example, an 8-bit microcontroller versus a 32-bit general purpose processor). A general approach is more valuable, and with the addition of some assembly language programming, it can be made as fast as a special case method.

6.7 Comparing $GF(p)$ and $GF(p^k)$ Arithmetic

Another issue is the comparison of $GF(p)$ versus $GF(p^k)$ arithmetic. For the same length operands, one could question the value of $GF(p^k)$ arithmetic. Why use $GF(p^k)$ arithmetic if $GF(p)$ arithmetic has similar or faster speed? We tried to find a partial answer to this question by comparing the speed of the IDMM algorithm for $GF(p^k)$ and the IMM (Incomplete Montgomery Multiplication) algorithm $GF(q)$ such that q is a prime of length approximately k times that of p , i.e., $q \approx p^k$. Note that since $GF(q)$ is a one-level field, the double Montgomery multiplication is not needed.

In Table 6.4, we compare the IDMM in $GF(p^k)$ with IMM in $GF(q)$. These codes were also written in assembly. We expected that $GF(p^k)$ arithmetic would be significantly slower than $GF(q)$ arithmetic, however, we were surprised that the difference is not that great. However, more comparisons for different types of algorithms and platforms are needed in order to settle this issue.

Table 6.4. Timings of the IMM code in $GF(q)$ and the IDMM code in $GF(p^k)$ in microseconds.

q	p	k	IMM in $GF(q)$	IDMM in $GF(p^k)$
160 bits	32 bits	5	1.6	2.3
192 bits	32 bits	6	2.3	3.4
224 bits	32 bits	7	3.2	4.3
256 bits	32 bits	8	3.9	5.3
288 bits	32 bits	9	4.6	7.2
320 bits	32 bits	10	5.3	8.6
416 bits	32 bits	13	8.3	14.2
512 bits	32 bits	16	12.3	17.8

6.8 Conclusions and Further Research

In this research, we are proposing two general purpose multiplication methods for the field of $GF(p^k)$. We use the Montgomery multiplication [18] and the incomplete reduction methods to build our new algorithms. Despite the higher complexity of the arithmetic in $GF(p^k)$ compared to $GF(p)$, the timing results show that the new multiplication method in $GF(p^k)$ is almost as fast as multiplication in $GF(p)$ for fixed bit lengths. We believe this method will be useful for creating fast implementations of cryptographic functions based on $GF(p^k)$ for arbitrary primes and arbitrary generating polynomials. The general approach is more robust from the security point of view since it does not depend on particular instances of the Galois field and does not restrict the field parameters in any way.

We are currently investigating other methods for multiplication in $GF(p^k)$. Further research is also needed to design efficient modular inversion methods which can work together with the proposed multiplication algorithms in $GF(p^k)$ for efficient software implementations of elliptic curve cryptographic functions.

CHAPTER 7

MODULAR INVERSION IN $GF(p^k)$

7.1 Introduction

The modular inversion operation plays an important role in public key cryptography, particularly, in computing point operations on an elliptic curve defined over the finite field $GF(p)$ or the finite extension field $GF(p^k)$ [15, 24].

Although it is not as performance critical as modular multiplication, inversion is the most costly arithmetic operation in EC systems. Therefore, most of the practical implementations try to avoid the use of inversion as much as possible. But it is not possible to avoid it completely. An example for this is the use of projective coordinates instead of the affine coordinates which are explained in Chapter 3.

In $GF(p)$ modular inversion can be calculated with the binary extended euclidean algorithm [23] or more efficiently with the montgomery inverse algorithm [14, 30]. In $GF(2^k)$ the Itoh-Tsujii algorithm [13] and its variants or the almost inverse algorithm [32] can be used.

In $GF(p^k)$, Bailey and Paar [2] proposed an inversion based on the Itoh-Tsujii inversion for the specific case in which $f(x)$ is a binomial. Lee et.al. [20] uses the extended euclidean and the almost inverse algorithms for inversion. Lim and Hwang [22] are introducing two different versions of the extended euclidean algorithm.

7.2 Inversion Algorithms

7.2.1 OEF Inversion

For any nonzero element $A(x) \in GF(p^k)$, there exists $B(x)$ and $U(x) \in GF(p^k)$ such that $A(x)B(x) + U(x)f(x) = 1$, where $B(x)$ is the inverse of $A(x)$ in $GF(p^k)$. Here, $f(x)$ is the generator polynomial which is monic and irreducible.

Bailey and Paar proposed an inversion method in the Optimal Extension Field (OEF) via a modification of the Itoh-Tsujii algorithm [2]. As it is shown in [2] the problem of extension field inversion is reduced to a subfield inversion using the fact that for any element $\alpha \in GF(p^k)$, $\alpha^{(p^k-1)/(p-1)} \in GF(p)$ [21]. The algorithm computes an inverse in $GF(p^k)$ as

$$A(x)^{-1} = (A(x)^r)^{-1}A(x)^{r-1} \text{ mod } f(x) \quad (7.1)$$

where $r = (p^k - 1)/(p - 1) = p^{k-1} + p^{k-2} + \dots + p + 1$. The algorithm below describes the procedure to compute the inverse. The subfield elements are written in lower case letters.

Algorithm 1: Optimal Extension Field Inversion

Input: $A(x) \in GF(p^k)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = 1 \text{ mod } f(x)$.

Step 1. $B(x) = A(x)$

Step 2. Use an addition chain to compute $B(x) = B(x)^{r-1}$

Step 3. $c_0 = B(x)A(x)$

Step 4. $c = c_0^{-1}$

Step 5. $B(x) = B(x)c$

Step 6. Output $B(x)$

To evaluate the equation (7.1), an efficient method to evaluate $A(x)^{r-1}$ is required. For a given field the exponent $r - 1$ will be fixed. Analogous to the Itoh-Tsujii

inversion, an addition chain is used for exponentiation. To construct this chain the frobenius map plays an important role in raising a field element to the p -th power. The calculations for this operation are easy when $f(x)$ is a binomial. For the rest of the calculations we need multiplications and a subfield inversion which can be calculated using the extended euclidean algorithm. The general expression for the complexity of this algorithm in terms of subfield multiplication is given as;

$$\begin{aligned} \#SM = & \llbracket \log_2(k-1) \rrbracket + H_w(k-1)(k-1) + \llbracket \log_2(k-1) \rrbracket \\ & + H_w(k-1) - 1(k^2 + k - 1) + 2k \end{aligned}$$

7.2.2 Extended Euclidian Algorithms

In [22] the inverse $B(x)$ is computed with the extended euclidean algorithms given below. These are general algorithms not depending on special values. We denote $\deg()$ as the degree of a polynomial. $F(x)_i$ and $G(x)_i$ are coefficients of the i th degree of the polynomial $F(x)$ or $G(x)$, respectively.

Algorithm 2: Exended Eulidean Algorithm

Input: $A(x) \in GF(p^k)$ and $f(x)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = 1 \pmod{f(x)}$.

Step 1. Set $B(x) = 0, C(x) = 1, F(x) = f(x)$ and $G(x) = A(x)$.

Step 2. While $\deg(F(x)) \neq 0$

Step 3. If $\deg(F(x)) < \deg(G(x))$, then exchange $F(x), B(x)$ with $G(x), C(x)$, respectively

Step 4. Update $F(x)$ and $B(x)$ as follows ($j = \deg(F(x)) - \deg(G(x))$)

$$\alpha = F(x)_{\deg(F(x))} G(x)_{\deg(G(x))}^{-1}, F(x) = F(x) - \alpha x^j G(x)$$

$$B(x) = B(x) - \alpha x^j C(x)$$

Step 5. Output $B(x) = F(x)_0^{-1} B(x)$

The algorithm above reduces the degree of the larger of $F(x)$ and $G(x)$ by at least one in each iteration of Step 2. Therefore, at most $2k - 2$ iterations are needed in Step 2. The most time-consuming operation is the subfield inversion in Step 4. In [22] an improved algorithm that uses less subfield inversions is given as follows.

Algorithm 3: Improved Extended Euclidean Algorithm

Input: $A(x) \in GF(p^k)$ and $f(x)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = 1 \pmod{f(x)}$.

Step 1. Set $B(x) = 0, C(x) = 1, F(x) = f(x)$ and $G(x) = A(x)$.

Step 2. While $\deg(F(x)) \neq 0$

Step 3. If $\deg(F(x)) < \deg(G(x))$, then exchange $F(x), B(x)$ with $G(x), C(x)$, respectively

Step 4. Update $F(x)$ and $B(x)$ as follows ($j = \deg(F(x)) - \deg(G(x))$)

$$\alpha = F(x)_{\deg(F(x))} G(x)_{\deg(G(x))}^{-1},$$

$$\beta = (F(x)_{\deg(F(x))-1} - \alpha G(x)_{\deg(G(x))-1}) G(x)_{\deg(G(x))}^{-1},$$

$$F(x) = F(x) - (\alpha x^j + \beta x^{j-1}) G(x)$$

$$B(x) = B(x) - (\alpha x^j + \beta x^{j-1}) C(x)$$

Step 5. If $\deg(F(x)) = \deg(G(x))$, then execute the following

$$\alpha = F(x)_{\deg(F(x))} G(x)_{\deg(G(x))}^{-1}, F(x) = F(x) - \alpha G(x)$$

$$B(x) = B(x) - \alpha C(x)$$

Step 6. Output $B(x) = F(x)_0^{-1} B(x)$

In the improved extended euclidean algorithm each iteration of Step 2 reduces the degree of $F(x)$ by at least two by subtracting a suitable multiple of $G(x)$. The numbers of subfield inversions and modular reductions are reduced by half compared to the extended euclidean algorithm.

The subfield inversion becomes more expensive when the size of p increases. We still need to minimize the number of subfield inversions. The extended euclidean algorithm (Algorithm 2) maintains the following equations throughout its internal processing.

$$A(x)B(x) + U(x)f(x) = F(x), A(x)C(x) + V(x)f(x) = G(x) \quad (7.2)$$

for some polynomials $U(x)$ and $V(x)$ (not interesting to us). Therefore we can see that these equations will hold if we multiply both $F(x)$ and $B(x)$ or $G(x)$ and $C(x)$ with the same constant. The following algorithm works because of this fact.

Algorithm 4: EE Algorithm using multiplication

Input: $A(x) \in GF(p^k)$ and $f(x)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = 1 \pmod{f(x)}$.

Step 1. Set $B(x) = 0, C(x) = 1, F(x) = f(x)$ and $G(x) = A(x)$.

Step 2. While $\deg(F(x)) \neq 0$

Step 3. If $\deg(F(x)) < \deg(G(x))$, then exchange $F(x), B(x)$ with $G(x), C(x)$, respectively

Step 4. Update $F(x)$ and $B(x)$ as follows ($j = \deg(F(x)) - \deg(G(x))$)

$$\alpha = G(x)_{\deg(G(x))}^2, \beta = (F(x)_{\deg(F(x))} G(x)_{\deg(G(x))}),$$

$$\gamma = G(x)_{\deg(G(x))} F(x)_{\deg(F(x))-1} - F(x)_{\deg(F(x))} G(x)_{\deg(G(x))-1}$$

$$F(x) = \alpha F(x) - (\beta x^j + \gamma x^{j-1}) G(x)$$

$$B(x) = \alpha B(x) - (\beta x^j + \gamma x^{j-1}) C(x)$$

Step 5. If $\deg(F(x)) = \deg(G(x))$, then execute the following

$$F(x) = G(x)_{\deg(G(x))} F(x) - F(x)_{\deg(F(x))} G(x)$$

$$B(x) = G(x)_{\deg(G(x))} B(x) - F(x)_{\deg(F(x))} C(x)$$

Step 6. Output $B(x) = F(x)_0^{-1} B(x)$

Algorithm 4. requires just one subfield inversion at the final step and has lower complexity than Bailey and Paar's OEF inversion. Table (7.1) summarizes the complexity of the three algorithms in this section. Further details can be found in [22].

Table 7.1. Complexity for inverse algorithms.

<i>Algorithm#</i>	<i>#multiplication</i>	<i>#reductions</i>	<i>#inversions</i>
Algorithm 2	$2k^2 + k - 4$	$2k^2 + k - 4$	$2k - 1$
Algorithm 3	$2k^2 + k - 4$	$k^2 - k - 2$	$k + 1$
Algorithm 4	$3k^2 - 5$	$k^2 + 4k - 6$	1

7.2.3 Almost Inverse Algorithms

In [20] the following algorithm is called the almost inverse algorithm. It finds $B(x)$ and an integer e satisfying $B(x)A(x) + U(x)f(x) = x^e$. We denote $\deg()$ as the degree of a polynomial. $F(x)_i$ and $G(x)_i$ are coefficients of the i th degree of the polynomial $F(x)$ or $G(x)$, respectively.

Algorithm 5: Almost Inverse Algorithm

Input: $A(x) \in GF(p^k)$ and $f(x)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = x^e \pmod{f(x)}$.

Step 1. Set $e = 0, B(x) = 0, C(x) = 1, F(x) = f(x)$ and $G(x) = A(x)$.

Step 2. While $F(x)$ contains factor x

Step 3. $F(x) = F(x)/x, C(x) = C(x)x, e = e + 1$

Step 4. If $\deg(F(x)) = 0$, then return $B(x) = F_0^{-1}B(x), e$

Step 5. If $\deg(F(x)) < \deg(G(x))$, then exchange $F(x), B(x)$
with $G(x), C(x)$, respectively

Step 6. $\alpha = -F(x)_0/G(x)_0$

Step 7. $F(x) = F(x) + \alpha G(x), B(x) = B(x) + \alpha C(x)$

Step 8. goto Step 2.

In [20] an improved version that reduces the number of iterations is given as follows.

Algorithm 6: Modified Almost Inverse Algorithm

Input: $A(x) \in GF(p^k)$ and $f(x)$.

Output: $B(x) = \sum b_i x^i$, so that $A(x)B(x) = x^e \pmod{f(x)}$.

Step 1. Set $e = 0, B(x) = 0, C(x) = 1, F(x) = f(x)$ and $G(x) = A(x)$.

Step 2. While $F(x)$ contains factor x

Step 3. $F(x) = F(x)/x, C(x) = C(x)x, e = e + 1$

Step 4. If $\deg(F(x)) = 0$, then return $B(x) = F_0^{-1}B(x), e$

Step 5. If $\deg(F(x)) < \deg(G(x))$, then exchange $F(x), B(x)$
with $G(x), C(x)$, respectively

Step 6. $j = \deg(F(x)) - \deg(G(x)), \beta = -F(x)_0/G(x)_0$

Step 7. If $(j \neq 0)$ then $\alpha = -F(x)_{\deg(F(x))}/G(x)_{\deg(G(x))}$, else $\alpha = 0$

Step 8. $F(x) = F(x) + (\alpha x^j + \beta)G(x), B(x) = B(x) + (\alpha x^j + \beta)C(x)$

Step 9. goto Step 2.

The modified almost inverse algorithm gives better performance than the extended euclidean algorithm (Algorithm 2) which was explained in the previous section. Further details can be found in [20]. In Algorithm 6, we know that $0 < e < 2k$ and the result is $B(x)x^{-e}$ where $B(x) = A(x)^{-1}x^e$.

The classical modular inverse can be defined as $A(x)^{-1} \pmod{f(x)}$ and the montgomery inverse as $A(x)^{-1}x^k \pmod{f(x)}$ [30] where $R = x^k$ is known as the residue (Chapter 6). We can further calculate the classical inverse or the montgomery inverse with similar methods in [30] for $GF(p^k)$ using the Double Montgomery Multiplication (DMM) or the Incomplete Double Montgomery Multiplication (IDMM). For the montgomery inverse we can calculate the following.

$$S(x) = DMM(B(x), x^{2k-e}) = A(x)^{-1}x^e x^{2k-e} x^{-k} = A(x)^{-1}x^k \pmod{f(x)}. \quad (7.3)$$

For the classical inverse we can calculate

$$DMM(S(x), 1) = A(x)^{-1}x^k 1x^{-k} = A(x)^{-1} \pmod{f(x)}. \quad (7.4)$$

7.3 Conclusion

We explained six different inversion algorithms in this Chapter. We can conclude that the best algorithm to calculate the inverse in $GF(p^k)$ is Algorithm 4., having even a lower complexity than the OEF inversion. The almost inverse algorithm can also be very useful if residue arithmetic is in use.

CHAPTER 8 CONCLUSIONS

This thesis describes new methods for finite field arithmetic and explains their use in Elliptic Curve Cryptosystems.

The *incomplete modular arithmetic* concept is applied to finite field arithmetic over $GF(p)$ and $GF(p^k)$. The results are demonstrating the effectiveness and practicality of the method. We introduced new modular addition and modular subtraction algorithms and implemented a new Montgomery variant modular multiplication algorithm in $GF(p)$. The *incomplete modular arithmetic* concept provides up to 13 % speedup in the execution of the ECDSA algorithm over the field $GF(p)$ for the bit length l of p in the range $161 \leq l \leq 256$.

We introduced two new modular multiplication algorithms based on the Montgomery multiplication method in $GF(p^k)$. The use of the *incomplete modular arithmetic* concept is crucial to perform an efficient modular multiplication algorithm in $GF(p^k)$. We are proposing to implement the finite field arithmetic without limiting the prime characteristic p and the generator polynomial and still obtain comparable timing results to the arithmetic implemented in $GF(p)$.

In Chapter 7, we explained various inversion algorithms for $G(p^k)$, most of them general purpose algorithms, that can be used with our multiplication algorithms. We realized that the modified extended euclidean algorithms were very efficient to perform the inversion operation.

Further research can be conducted in finding better general purpose multiplication and inversion algorithms for $GF(p^k)$.

BIBLIOGRAPHY

- [1] Daniel V. Bailey and Christof Paar. Optimal extension fields for fast arithmetic in public-key algorithms. In *CRYPTO*, pages 472–485, 1998.
- [2] D.V. Bailey and C. Paar. Efficient arithmetic in finite field extensions with application in elliptic curve cryptography. *Journal of Cryptology*, 2000.
- [3] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 200–207. Springer, Berlin, Germany, 1992.
- [4] J.W. Chung, S.G. Sim, and P.J. Lee. Fast implementation of elliptic curve defined over $GF(p^m)$ on calmrisc with mac2424 coprocessor. In C.K. Koc and C. Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2000*, Lecture Notes in Computer Science, No. 1965, pages 57–70. Springer, Berlin, Germany, 2000.
- [5] H. Cohen, A. Miyaji, and T. Ono. Efficient elliptic curve exponentiation using mixed coordinates. In K. Ohta and D. Pei, editors, *Advances in Cryptology — ASIACRYPT 98*, Lecture Notes in Computer Science, No. 1514, pages 51–65. Springer, Berlin, Germany, 1998.
- [6] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. U.S. Patent Number 5,463,690, October 1995.
- [7] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22:644–654, November 1976.
- [8] S. R. Dussé and B. S. Kaliski Jr. A cryptographic library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EUROCRYPT*

- 90, Lecture Notes in Computer Science, No. 473, pages 230–244. Springer, Berlin, Germany, 1990.
- [9] T. ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, July 1985.
- [10] T. Hasegawa, J. Nakajima, and M. Matsui. A practical implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer. In H. Imai and Y. Zheng, editors, *First International Workshop on Practice and Theory in Public Key Cryptography*, Lecture Notes in Computer Science, No. 1431, pages 182–194. Springer, Berlin, Germany, 1998.
- [11] IEEE. P1363: Standard specifications for public-key cryptography. Draft Version 7, September 1998.
- [12] IEEE. P1363: Standard specifications for public-key cryptography. Draft Version 13, November 12, 1999.
- [13] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, September 1988.
- [14] B. S. Kaliski Jr. The Montgomery inverse and its applications. *IEEE Transactions on Computers*, 44(8):1064–1065, August 1995.
- [15] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.
- [16] N. Koblitz. Hyperelliptic cryptosystems. *Journal of Cryptology*, 1(3):139–150, 1989.
- [17] Ç. K. Koç. High-Speed RSA Implementation. Technical Report TR 201, RSA Laboratories, 73 pages, November 1994.
- [18] Ç. K. Koç and T. Acar. Montgomery multiplication in $GF(2^k)$. *Designs, Codes and Cryptography*, 14(1):57–69, April 1998.

- [19] Ç. K. Koç, T. Acar, and B. S. Kaliski Jr. Analyzing and comparing Montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, June 1996.
- [20] E. J. Lee, D. S. Kim, and P. J. Lee. Speedup of F_p^m for elliptic curve cryptosystems. In *In Proceedings of ICISC'98*, ISBN 89-85305-14-X, pages 81–91, December 1998.
- [21] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, New York, NY, 1994.
- [22] C.H. Lim and H. S. Hwang. Fast implementation of elliptic curve arithmetic in $GF(p^n)$. In H. Imai and Y. Zheng, editors, *Third International Workshop on Practice and Theory in Public Key Cryptography*, Lecture Notes in Computer Science, No. 1751, pages 405–421. Springer, Berlin, Germany, 2000.
- [23] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.
- [24] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, MA, 1993.
- [25] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. Springer, Berlin, Germany, 1985.
- [26] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.
- [27] National Institute for Standards and Technology. Digital Signature Standard (DSS). FIPS PUB 186-2, January 2000.
- [28] J.-J. Quisquater and C. Couvreur. Fast decipherment algorithm for RSA public-key cryptosystem. *Electronics Letters*, 18(21):905–907, October 1982.
- [29] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

- [30] E. Savaş and Ç. K. Koç. The Montgomery modular inverse - revisited. *IEEE Transactions on Computers*, 49(7):763–766, July 2000.
- [31] R.J. Schoof. Elliptic curves over finite fields and the computation of square roots mod p . *Mathematics of Computation*, 44:483–494, 1985.
- [32] R. Schroepfel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. Springer, Berlin, Germany, 1995.
- [33] J. H. Silverman. *The Arithmetic of Elliptic Curves*. Springer, Berlin, Germany, 1986.
- [34] S. Vanstone. Responses to nist's proposal (communicated by john anderson). *Communications of the ACM*, 35:50–52, July 1992.