Software Elliptic Curve Cryptography

by

Majid Khabbazian
B.Sc., Sharif University of Technology, 2002

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of

Masters of Applied Science

in the Department of Electrical and Computer Engineering

© Majid Khabbazian, 2004

University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part by photocopy or other means, without the permission of the author.

Supervisor: Dr. T.A. Gulliver

ABSTRACT

In this thesis, we study the software implementation of the NIST-recommended elliptic curves over prime fields. Our implementation goals are to achieve a fast, small, and portable cryptographic library, which supports elliptic curve digital signature generation and verification. The implementation results are presented on a Pentium II 448.81 MHZ.

We also consider the sliding window algorithm (SWA) and combine it with integer representations to generate point multiplication methods. We present a modified Katti representation to improve the SWA speed. We also present a simple signed binary representation (SSBR). A generalized sliding window algorithm is proposed which can be combined with the SSBR to obtain fast and efficient methods for both single and multiple point multiplication. These methods can use available memory efficiently, and so are ideal for memory-constrained devices.

Table of Contents

Abstract ii

Table of contests iv

List of Tables vii

List of Figures x

List of Abbreviations xi

Acknowledgement xii

Dedication xiii

1 Introduction 1

1.1 Significance of Research 2

1.2 Thesis Outline 2

2 Elliptic Curve Cryptography (ECC) 3

2.1 Introduction 3

2.2 Mathematical Foundations 4

2.2.1 Group 4

2.2.2 Field 4

2.2.3 Finite Field 5

2.2.4 Discrete Logarithm Problem (DLP) 5

2.3 Elliptic Curve over Finite Fields 6

2.3.1 Definition 6

2.3.2 Elliptic Curve over Prime Fields 7

2.3.2.1 Affine Coordinates 7

2.3.2.2 Standard Projective Coordinates 8

2.3.2.3 Jacobian Projective Coordinates 9

2.3.2.4 Chudnovsky Projective Coordinates 11

2.3.2.5 Mixed Coordinates 11

2.4 Elliptic Curve Discrete Logarithm Problem (ECDLP) 14

2.5 ECC Security 14

2.5.1 ECC Attacks 15

2.5.1.1 General-Purpose Attacks 15

2.5.1.2 Special-Purpose Attacks 16

2.5.2 Attack Countermeasures 17

3 Software Implementation of ECC over $GF(p)$ **18**

3.1 Software Implementation Options 18

3.2 NIST-Recommended Elliptic Curves over Prime Fields 22

3.3 Prime Field Arithmetic 24

3.3.1 Field Representation 24

3.3.2 Addition and Subtraction 25

3.3.3 Multiplication and Squaring 30

3.3.4 Modular Reduction 34

3.3.5 Modular Inversion 38

3.4 Arithmetic on Elliptic Curves over $GF(p)$ 40

3.4.1 Addition and Subtraction 40

3.4.2 Point Doubling 41

3.4.3 Point Multiplication 42

3.4.3.1 Random Point Multiplication 42

3.4.3.1.1 Integer Representations 43

3.4.3.1.1.1 Binary Representation 44

3.4.3.1.1.2 The Non-Adjacent Form Representation 44

3.4.3.1.1.3 The m-ary and Signed m-ary Representations 45

3.4.3.1.1.4 The KT and Katti Representations 46

3.4.3.1.1.5 A Simple Signed Binary Representation 46

3.4.3.1.2 Windowing Algorithms 47
3.4.3.1.2.1 The Sliding Window Algorithm 47
3.4.3.1.2.2 The Generalized Sliding Window Algorithm 51
3.4.3.2 Double Point Multiplication 53
3.4.3.3 Fixed Point Multiplication 56
3.5 Hash Function 60
3.6 Elliptic Curve Digital Signature Algorithm 61

4 Implementation Results **63**
4.1 Finite Field Arithmetic Timing 64
4.1.1 Modular Addition and Subtraction Timing 65
4.1.2 Modular Reduction Timing 66
4.1.3 Modular Multiplication and Squaring Timing 67
4.1.4 Modular Inversion Timing 69
4.2 Random Point Multiplication Timing 70
4.3 Fixed Point Multiplication 73
4.4 Double Point Multiplication Timing 74
4.5 ECDSA Timing 74
4.6 Summary 75

5 Conclusion and Suggestions for Future Work **76**
5.1 Conclusion 76
5.2 Suggestions for Future Work 77

Bibliography **78**

List of Tables

Table 2.1 Field properties 5

Table 2.2 Point addition in affine coordinates: $(x_3, y_3) = (x_2, y_2) + (x_1, y_1)$ 8

Table 2.3 Point addition in standard projective coordinates:
$(x_3, y_3, z_3) = (x_2, y_2, z_2) + (x_1, y_1, z_1)$ 9

Table 2.4 Point addition in Jacobian coordinates:
$(x_3, y_3, z_3) = (x_2, y_2, z_2) + (x_1, y_1, z_1)$ 10

Table 2.5 Point addition in mixed Jacobian-affine and mixed Jacobian-Cudnovsky coordinates 12

Table 2.6 Point conversion complexity 12

Table 2.7 Number of additions and doublings, A=affine, P=standard projective, J=Jacobian, C=Chudnovsky 13

Table 2.8 Required computations for doubling when $a = -3$ 13

Table 2.9 Comparative bit lengths 14

Table 2.10 Summary of ECDLP attacks and their countermeasures 17

Table 3.1 NIST-recommended elliptic curves over prime fields 23

Table 3.2 An elliptic curve similar to the NIST-recommended elliptic curves ... 24

Table 3.3 Definitions used in the C programs 27

Table 3.4 First implementation of Steps 1 and 2 of Algorithm 1 27

Table 3.5 Second implementation of Steps 1 and 2 of Algorithm 1 28

Table 3.6 Third implementation of Steps 1 and 2 of Algorithm 1 29

Table 3.7 Speed and memory trade-off for implementing modular addition 30

Table 3.8 Point subtraction formula in affine coordinates 41

Table 3.9 A classification of PM methods based on window size and integer representation 49
Table 3.10 SHA properties 60
Table 4.1 Measuring the execution time of $f(a=100)$ in μs on a Pentium II 400 MHZ using the C language 64
Table 4.2 Timing (in μs) for modular addition and subtraction including reduction (without compiler optimization) 65
Table 4.3 Timing (in μs) for modular addition and subtraction including reduction (with compiler optimization) 66
Table 4.4 Timing (in μs) for modular reduction 66
Table 4.5 Timing (in μs) for word multiplication functions 67
Table 4.6 Timing (in μs) for Barrett reduction function using the fast word multiplication function 67
Table 4.7 Timing (in μs) for classical and Karatsuba multiplication (including fast reduction) using the slow word multiplication function (Function 2) ... 68
Table 4.8 Timing (in μs) for classical and Karatsuba multiplication (including fast reduction) using the fast word multiplication function (Function 1) ... 69
Table 4.9 Timing (in μs) for classical squaring (including fast reduction) 69
Table 4.10 Timing (in μs) for two implementations of right shift function 70
Table 4.11 Timing (in μs) for inversion functions 70
Table 4.12 Timing (in ms) for the binary and NAF point multiplication methods using the slow word multiplication function 70
Table 4.13 Timing (in ms) for binary and NAF point multiplication methods using the fast word multiplication function 71
Table 4.14 Timing (in ms) for the proposed generalized sliding window method (Algorithm 18) 72

Table 4.15 Timing (in ms) for fixed point multiplication ($v=1$, $b=32$) 73

Table 4.16 Timing (in ms) for fixed point multiplication ($v=1$, $b=64$) 73

Table 4.17 Timing (in ms) for double point multiplication (Algorithm 19) 74

Table 4.18 Timing (in ms) for digital signature generation and verification 75

List of Figures

Figure 2.1 Adding points in an elliptic curve 6

Figure 3.1 Underling finite field options 18
Figure 3.2 Elliptic curve options 22
Figure 3.3 Partition of the multiplier k 58
Figure 3.4 Using a hash function in signature generation and verification algorithms 61

Figure 4.1 Implementation architecture 63
Figure 4.2 The percentage speed improvement of the proposed generalized sliding window method over the binary method 72

List of Abbreviations

DLP	Discrete Logarithm Problem
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
ECDLP	Elliptic Curve Discrete Logarithm Problem
EEA	Extended Euclidean Algorithm
IEEE	Institute of Electrical & Electronics Engineers
IFP	Integer Factorization Problem
ISO	International Organization for Standardization
LL	Lim-Lee
NAF	Non-Adjacent Form
NB	Normal Basis
NIST	National Institute of Standards and Technology
OEF	Optimal Extension Fields
ONB	Optimal Normal Basis
RDTSC	Read-Time Stamp Counter
SHA	Secure Hash Algorithm
SSBR	Simple Signed Binary Representation
SWA	Sliding Window Algorithm

Acknowledgement

I would like to express my gratitude to all who helped me complete this thesis. I am deeply indebted to my supervisor Professor Aaron Gulliver whose help, guidance and encouragement helped me during the research and writing of this thesis.

Dedication

To My Wonderful Wife, Hosna

Chapter 1

Introduction

Cryptography plays a central role in systems in which an insecure channel is used to transfer data. It not only can be employed to protect the privacy of data, but also for authentication (the process of proving one's identity), integrity (assuring the receiver that the received message has not been altered) and non-repudiation (a mechanism that prevents the sender from denying that they sent the message). In general, cryptographic techniques can be divided into two classes, symmetric key and asymmetric key (also called public-key) cryptography. In symmetric key cryptography, the same key is used for both encryption and decryption. The main problem using this approach is the exchange of the key. In contrast to symmetric key, public-key cryptography uses two keys, public and private key, for encryption and decryption, respectively. The primary advantage of public-key cryptography is that it removes the need to exchange the key between the sender and the receiver.

Since the invention of public key cryptography by Whitfield Diffie and Martin Hellman in 1976 [1], numerous public key cryptographic systems have been proposed. All of these systems rely on the difficulty of a mathematical problem for their security. Among these hard problems, only two have passed the test of time, namely the Integer Factorization Problem (IFP) and the Discrete Logarithm Problem (DLP). The two widely used cryptosystems, RSA and El-Gamal, are based on the IFP and DLP over the multiplicative group of a finite field, respectively.

In 1985, Neal Koblitz [2] and V.S. Miller [3] independently proposed using elliptic curves for public key cryptosystems. They did not invent a new cryptographic algorithm, but they used the group of points on an elliptic curve for the DLP. The DLP over this group is called the Elliptic Curve Discrete Logarithm Problem (ECDLP). Since there is no known subexponential running time algorithm to solve the ECDLP (IFP and DLP over the multiplicative group of a finite field, can be solved in subexponential running time), it is believed to be much harder than the IFP and DLP over the multiplicative group of a finite field. Therefore, smaller key sizes can be used in an

elliptic curve cryptosystem to get the same level of security as counterparts such as RSA. Smaller key sizes result in smaller system parameters, bandwidth saving, faster implementation, and lower power consumption. These characteristics make an Elliptic Curve Cryptography (ECC) very practical. This is due to the fact that information technology is developing very fast and many of the information technology applications such as handhelds, mobile phones and pay-TV are realized as embedded systems in which memory, power and bandwidth for communication are constrained.

Elliptic curve cryptography is being considered by standard organizations such as the National Institute of Standards and Technology (NIST), International Organization for Standardization (ISO) and Institute of Electrical & Electronics Engineers (IEEE). All these organizations have released standards, which are continuously up-dated to conform to the state-of-the-art in ECC.

1.1 Significance of Research

The performance of an elliptic curve cryptosystem is directly related to the performance of the point multiplication operation (an elliptic curve operation, which is explained in Chapter 3). There are many existing algorithms for implementation of this operation. This thesis investigates and improves the algorithms for these operations with the goal of increasing the speed and decreasing the required memory. It also presents a software implementation of the NIST-recommended elliptic curves over prime fields.

1.2 Thesis Outline

This thesis consists of five chapters. Chapter 2 provides background information on finite field and elliptic curve arithmetic. It also summarized the elliptic curve cryptography attacks and their countermeasures. In Chapter 3, we briefly describe the elliptic curve software implementation options. We then present required algorithms to implement Elliptic Curve Digital Signature Algorithm. We also introduce a Simple Signed Binary Representation (SSBR) and a modified Katti representation. We then use the SSBR with our proposed algorithms to introduce new simple methods for computing single and multiple point multiplication. Finally, we present our ECDSA implementation results in Chapter 4 and conclude in Chapter 5.

Chapter 2

Elliptic Curve Cryptography (ECC)

2.1 Introduction

Public key cryptosystems are based on special kind of one-way functions known as trapdoor one-way functions. Mathematically, a one-way function f is one for which $f(x)$ is easy to compute, but for a general value y within the selected range, it is computationally difficult to find a value x within the expected domain such that $f(x)=y$. A trapdoor one-way function is one for which $f(x)=y$ becomes easy to solve if additional information, called the trapdoor, is available.

Number theory is one of the most important sources of one-way functions. Examples of such functions are the Integer Factorization Problem (IFP) and the Discrete Logarithm Problem (DLP). The security of RSA and the Diffie-Hellman key exchange (the first public key-exchange algorithm), is based on these two problems, respectively. The discrete logarithm problem applies to groups. The difficulty of DLP depends on the choice of this group. For example if the additive group of a finite field is used, then computing DLP is equivalent to solving the $ax = b \pmod{n}$. This can be done easily using the extended Euclidean algorithm. If the multiplicative group of a finite field is used (as in the Diffie-Hellman algorithm), then the problem can be hard. The Jacobian, an abelian group associated with an algebraic curve over finite field, are attractive alternative groups. Examples of such algebraic curves are C_{ab} curves, superelliptic curves, hyperelliptic curves, and elliptic curves (hyperelliptic curves of genus 1 [4]).

Elliptic curves are algebraic/geometric entities that have been studied for a long time. They arise naturally in many branches of mathematics. In the recent past they have, for instance, been used in the proof of Fermat's last theorem. The application of elliptic curves in cryptography was first introduced in 1985 by Neal Koblitz [2] and Victor Miller [3]. They independently proposed

public key cryptosystems based on the elliptic curve discrete logarithm problem (the discrete logarithm problem over a group of points on an elliptic curve). Unlike the discrete logarithm problem over the multiplicative group of a finite field and the integer factorization problem, there is no known subexponential algorithm to solve the elliptic curve discrete logarithm problem (ECDLP). Consequently, ECDLP can be used to implement cryptosystems similar to Diffie-Hellman using much smaller key sizes with the same level of security. For example, 160-bit elliptic curve cryptosystems are believed to provide a level of security equivalent to 1024 RSA. The smaller keys result in smaller system parameters, bandwidth savings, faster implementations and lower power consumption. These advantages make elliptic curve cryptosystems ideal for restricted devices such as smart cards or mobile phones.

2.2 Mathematical Foundations

Before considering elliptic curves, we require some algebraic definitions, namely group, field, and finite field definitions.

2.2.1 Group

A group G is a finite or infinite set of elements together with a binary operation, which together satisfy the four fundamental properties of closure, associativity, the identity property, and the inverse property:

1. Closure: If $A,B \in G$, then $AB \in G$.
2. Associativity: For All $A,B,C \in G, (AB)C = A(BC)$.
3. Identity: There exists an element I, such that $AI = IA = A$ for all $A \in G$.
4. Inverse: For every $A \in G$ there exists an element $B = A^{-1}$ such that $AB = BA = I$.

2.2.2 Field

A field is a set together with two binary operation "+" and "." (Addition and Multiplication, respectively), satisfying the properties given in Table 2.1.

Property	**Addition**	**Multiplication**
Commutativity	$a+b=b+a$	$ab=ba$
Associativity	$(a+b)+c=a+(b+c)$	$(ab)c=a(bc)$
Distributivity	$a(b+c)=ab+ac$	$(a+b)c=ac+bc$
Identity	$a+0=a=0+a$	$a\cdot 1=a=1\cdot a$
Inverses	$a+(-a)=0=(-a)+a$	$a\cdot a^{-1}=1=a^{-1}\cdot a, a\neq 0$

Table 2.1. Field properties

2.2.3 Finite Field

A finite field is a field with a finite number of elements, also called a Galois field. The order of a finite field is always a prime or a power of a prime. For each prime power there exist exactly one (up to isomorphism) finite field $GF(p^n)$, often written as $\mathbb{F}_{p^n}$, or simply $\mathbb{F}_q$.

It is worth pointing out that for $K = GF(p^n)$, p is called the characteristic of the field K and is denoted $char(K)$.

2.2.4 Discrete Logarithm Problem (DLP)

The discrete logarithm problem (DLP) is a one-way function based on the difficulty of finding a logarithm in a group. The DLP has been extensively studied and has been the basis of several public key cryptosystems. It is defined as follows.

Given an element g in a group G of order n, and another element y of G, the problem is find x such that $g^x = y$, if such an integer exists.

2.3 Elliptic Curve over Finite Fields

2.3.1 Definition

An elliptic curve E over the field $\mathbb{F}$ is a smooth curve in the so called "long Weierstrass form"

$$E : Y^2 + a_1XY + a_3Y = X^3 + a_2X^2 + a_4X + a_6, a_i \in \mathbb{F} \tag{2.1}$$

Let $E(\mathbb{F})$ denotes the set of points $(x, y) \in \mathbb{F}^2$ that satisfy this equation, along with a "point at infinity" denoted O. We can now define an addition operation for $E(\mathbb{F})$ with all the four fundamental properties of closure, associativity, the identity property, and the inverse property, to construct a group. Figure 2.1 shows a graphic expression of point addition for real numbers. As shown in Figure 2.1, the straight line joining P and Q intersects the curve at one additional point. By reflecting this point in the x-axis, we obtain another point, which we call $P+Q$. We can also add Q to itself, or double it in the same way. In this case, we take the tangent to the curve at Q instead of joining P and Q (a special case of point addition). The group law can be stated as follows: If three points on an elliptic curve lie on a straight line, their sum is zero.

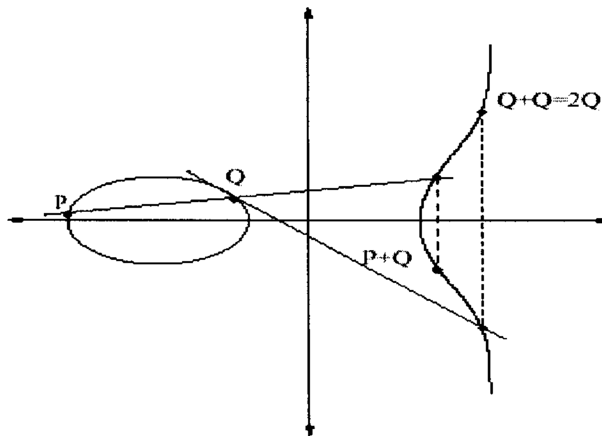

Figure 2.1. Adding points in an elliptic curve

From this geometric definition, we can determine algebraic formulas for the group law. In the next section, we provide an overview of different algebraic formulas (over $\mathbb{F}_p$, $p > 3$) using different variants of point coordinates.

2.3.2 Elliptic Curves over Prime Fields

When $char(\mathbb{F}) \neq 2,3$ the long Weierstrass form in equation (2.1) can be transformed to an equation of an isomorphic curve given in the short Weierstrass form

$$E: y^2 = x^3 + ax + b \quad a,b \in \mathbb{F} \text{ and } 4a^3 + 27b^2 \neq 0$$

using the change of variables

$$X \to x - \left(\frac{a_2}{3} + \frac{a_1^2}{12}\right)$$
$$Y \to y - \left(\frac{a_1 X + a_3}{2}\right)$$

There are different formulas to add two points, depending on what point coordinates are used. In this section, we are concerned only with elliptic curves over fields of characteristic $p > 3$.

2.3.2.1 Affine Coordinates

With affine coordinates, a point is represented as (x, y). Affine coordinates are the simplest representation of a point and are usually used to communicate or to store precomputed points since they need minimum bandwidth and memory in comparison to other coordinate representations. Table 2.2 shows how two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ can be added in affine coordinates to get $P_3 = (x_3, y_3)$.

Addition $P_1 \neq \pm P_2$	**Doubling** $P_1 = P_2$
$\lambda = \frac{y_2 - y_1}{x_2 - x_1}$	$\lambda = \frac{3x_1^2 - 3}{2y_1}$
$x_3 = \lambda^2 - x_1 - x_2$	$x_3 = \lambda^2 - x_1 - x_2$
$y_3 = \lambda(x_1 - x_3) - y_1$	$y_3 = \lambda(x_1 - x_3) - y_1$

Table 2.2. Point addition in affine coordinates: $(x_3, y_3) = (x_2, y_2) + (x_1, y_1)$

Notice that for adding and doubling a point in affine coordinates we need a modular inversion. Since inversion in $GF(p)$ is significantly more expensive in comparison to multiplication, affine coordinates are highly inefficient. One way to avoid modular inversion is to use projective coordinates of which several types have been proposed. In fact, the appropriateness of using projective coordinates is determined by the ratio

$$\frac{\text{time to compute an inverse}}{\text{time to multiply}}.$$

The larger this ratio, the more attractive it is to implement projective coordinates.

2.3.2.2 Standard Projective Coordinates

We can avoid modular inversions in point addition and doubling at the price of more modular multiplications. This can be done by using extra values to represent a point. In standard projective coordinates, each point is represented as (x_P, y_p, z_p).

Converting a point $P = (x_A, y_A)$ in affine coordinates to $P = (x_P, y_p, z_p)$ in standard projective coordinates can be simply done as follows

$$x_P \leftarrow x_A,\ y_P \leftarrow y_A,\ z_P \leftarrow 1$$

However, a modular inversion is required to do the reverse conversion

$$x_A \leftarrow \frac{x_P}{z_P},\ y_A \leftarrow \frac{y_P}{z_P}$$

To avoid inversion, we first convert the point representation from affine coordinates to projective coordinates. After that, we can do all required addition/doubling without any modular inversions. Finally, we can convert the result from projective coordinates to affine coordinate using a modular inversion. Addition and doubling formulas in standard projective coordinates are summarized in Table 2.3.

Addition $P_1 \neq \pm P_2$	**Doubling** $P_1 = P_2$
$u = y_2 z_1 - y_1 z_2$	$w = a z_1^2 + 3 x_1^2$
$v = x_2 z_1 - x_1 z_2$	$s = y_1 z_1$
$A = u^2 z_1 z_2 - v^3 - 2v^2 x_1 z_2$	$B = x_1 y_1 s$
$x_3 = vA$	$h = w^2 - 8B$
$y_3 = u(v^2 x_1 z_2 - A) - v^3 y_1 z_2$	$x_3 = 2hs$
$z_3 = v^3 z_1 z_2$	$y_3 = w(4B - h) - 8 y_1^2 s^2$
	$z_3 = 8s^3$

Table 2.3. Point addition in standard projective coordinates:
$(x_3, y_3, z_3) = (x_2, y_2, z_2) + (x_1, y_1, z_1)$

2.3.2.3 Jacobian Projective Coordinates

It turns out that other projective coordinates require smaller number of field operations to compute the group operation [5]. Jacobian coordinates is one of these coordinates. In fact, Jacobian coordinates is a variant of projective coordinates where triples (x_J, y_J, z_J) represent a point (x_A, y_A) on the elliptic curve. Conversion from affine to Jacobian coordinates can be done

similar to the standard projective coordinates. Conversion from Jacobian to affine coordinate can be accomplished as follows

$$x_A \leftarrow \frac{x_J}{z_J^2},\ y_A \leftarrow \frac{y_J}{z_J^3}$$

We see from Table 2.4 that Jacobian coordinates offer a faster doubling and a slower addition than standard projective coordinates. One way to make a faster addition is to use Chudnovsky projective coordinates.

Addition $P_1 \neq \pm P_2$	**Doubling** $P_1 = P_2$
$U_1 = x_1 z_2^2$	$S = 4x_1 y_1^2$
$U_2 = x_2 z_1^2$	$M = 3x_1^2 + az_1^4$
$S_1 = y_1 z_2^3$	$T = -2S + M^2$
$S_2 = y_2 z_1^3$	$x_3 = T$
$H = U_2 - U_1$	$y_3 = -8y_1^4 + M(S - T)$
$r = S_2 - S_1$	$z_3 = 2y_1 z_1$
$x_3 = -H^3 - 2U_1 H^2 + r^2$	
$y_3 = -S_1 H^3 + r(U_1 H^2 - x_3)$	
$z_3 = z_1 z_2 H$	

Table 2.4. Point addition in Jacobian coordinates: $(x_3, y_3, z_3) = (x_2, y_2, z_2) + (x_1, y_1, z_1)$

2.3.2.4 Chudnovsky Projective Coordinates

Chudnovsky [5] proposed to represent a point (x, y, z) in Jacobian coordinates as (x, y, z, z^2, z^3). The addition formulas in Chudnovsky coordinates remain the same as for Jacobian coordinates given in Table 2.4. The advantages is that we don't have to compute z_1^2, z_2^2, z_1^3 and z_2^3 to obtain the values U_1, U_2, S_1 and S_2 because z_1^2, z_2^2, z_1^3 and z_2^3 are already available. However, we need to compute z_3^2 and z_3^3 to get the result in Chudnovsky coordinates. Therefore, Chudnovsky coordinates require two fewer squarings than Jacobian coordinates to compute point addition. On the other hand, Chudnovsky coordinates require one multiplication more than Jacobian coordinates to compute doubling.

2.3.2.5 Mixed Coordinates

Cohen et al. [6] recommended using mixed coordinates, where the inputs and outputs to point addition/doubling may be in different coordinates. Table 2.5 illustrates how we can add a point represented in Jacobian coordinates with a point represented in affine coordinates (addition in mixed Jacobian-affine coordinates). It also shows doubling in mixed Jacobian-Chudnovsky coordinates where one point is represented in Jacobian coordinates and the other is represented in Chudnovsky coordinates.

Addition (mixed Jacobian-affine) $(x_1,y_1,z_1)+(x_2,y_2)=(x_3,y_3,z_3)$	Doubling (mixed Jacobian-Chudnovsky) $(x_1,y_1,z_1)+(x_2,y_2,z_2,z_2^2,z_2^3)=(x_3,y_3,z_3)$
$A=x_2z_1^2$	$A=x_1z_2^2$
$B=y_2z_1^3$	$B=y_1z_2^3$
$C=A-x_1$	$C=x_2z_1^2-A$
$D=B-y_1$	$D=y_2z_1^3-B$
$x_3=D^2-(C^3+2x_1C^2)$	$x_3=D^2-2AC^2-C^3$
$y_3=D(x_1C^2-x_3)-y_1C^3$	$y_3=D(AC^2-x_3)-BC^3$
$z_3=z_1C$	$z_3=z_1z_2C$

Table 2.5. Point addition in mixed Jacobian-affine and mixed Jacobian-Cudnovsky coordinates

In order to use mixed coordinates we may need to change coordinates. Table 2.6 shows the number of field operation required to convert from one set of coordinates to another. In this table, M denotes field multiplication or squaring cost; and I denotes field inversion cost. However, in the remaining tables in this section, M and S denote, respectively, the cost of field multiplication and field squaring.

From \ To	Affine	Projective	Jacobian	Chudnovsky
Affine	—	—	—	—
Projective	$2M+I$	—	$2M+I$	$2M+I$
Jacobian	$4M+I$	$4M+I$	—	$2M$
Chudnovsky	$4M+I$	$4M+I$	—	—

Table 2.6. Point conversion complexity

The required number of additions and doublings in various coordinates are listed in Table 2.7. From this table we can select an appropriate coordinate system by considering the ratio $\frac{I}{M}$. This factor depends on both the field and its implementation.

Doubling (Arbitrary a)		**General Addition**		**Mixed Coordinates**	
$2A \to A$	$2M+2S+I$	$A+A \to A$	$2M+S+I$	$J+A \to J$	$8M+3S$
$2P \to P$	$7M+5S$	$P+P \to P$	$12M+2S$	$J+C \to J$	$11M+3S$
$2J \to J$	$4M+6S$	$J+J \to J$	$12M+4S$	$C+A \to C$	$8M+3S$
$2C \to C$	$5M+6S$	$C+C \to C$	$11M+3S$		

Table 2.7. Number of additions and doublings, A=affine, P=standard projective, J=Jacobian, C=Chudnovsky

From Table 2.8 we can see that point doubling computation cost can be reduced when $a=-3$. In fact, an elliptic curve $E_{a,b}$ can be transformed into an $\mathbb{F}_q$-isomorphic one $E_{a',b'}$ with $a'=-3$ if and only if $\frac{-3}{a}$ has a fourth root in $\mathbb{F}_q$. This holds for about a quarter of the values of a when $q \equiv 1 \pmod 4$, and half the values when $q \equiv 3 \pmod 4$ [7].

Doubling ($a=-3$)	
$2A \to A$	$2M+2S+I$
$2P \to P$	$7M+3S$
$2J \to J$	$4M+4S$
$2C \to C$	$5M+4S$

Table 2.8. Required computations for doubling when $a=-3$

2.4 Elliptic Curve Discrete Logarithm Problem (ECDLP)

Let E be an elliptic curve over some finite field $\mathbb{F}_q$, and P a point of order n on E. The elliptic curve discrete logarithm problem (ECDLP) on E is to find the integer $k \in [0, n-1]$, if such an integer exists, so that

$$Q = kP\text{, where } kP = \underbrace{P + P + \ldots + P}_{k \text{ times}}.$$

It is believed that the usual discrete logarithm problem over the multiplicative group of a finite field (DLP) and ECDLP are not equivalent problems, and that ECDLP is significantly more difficult than DLP. The main reason is that there is no known subexponential-time algorithm to solve ECDLP in general.

2.5 ECC Security

ECC is widely regarded as the strongest asymmetric algorithm for a given key length, so they are especially attractive for security applications where computational power and integrated circuit space is limited, such as smart cards, PC (personal computer) cards, and wireless devices. Table 2.9 gives the approximate parameter size for comparable strength elliptic curve systems and RSA.

Elliptic curve cryptosystem (Order of base point P)	**RSA (length of modulus n)**
106 bits	512 bits
132 bits	768 bits
163 bits	1024 bits
224 bits	2048 bits
384 bits	7680 bits

Table 2.9. Comparative bit lengths

Certicom, a major commercial proponent of ECC, has sponsored several challenges to the ECC algorithm [8]. The most complex to have been solved was a 109-bit key, which was broken by a team of researchers near the beginning of 2003. The team which broke the key used a

massively parallel attack based on the birthday attack, using over 10,000 Pentium class PCs running continuously for over 540 days. The minimum recommended key size for ECC, 163 bits, is currently estimated to require 10^8 times the computing resources as that required for the 109 bit problem [9].

It is also possible to attack ECC using special-purpose hardware. Van Oorschot and Wiener [10] proposed an attack against a 120 bit EC system using special-purpose hardware. In their 1996 study, they estimated that if $n \approx 10^{36} \approx 2^{120}$, then a machine with $r = 330,000$ processors that could be built for about US $10 million would compute a single discrete logarithm in about 32 days. However , such hardware attacks are still infeasible for $n > 160$.

2.5.1 ECC Attacks

There are two types of attacks, special-purpose and general-purpose, for solving ECDLP. Special-purpose attack algorithms are tailored to perform better for the elliptic curves with a special form. In contrast, the running times of general-purpose attacks depend only on the size of elliptic curve parameters. In the next two sections we briefly overview some of the known general-purpose and special-purpose attacks.

2.5.1.1 General-Purpose Attacks

1. **Exhaustive Search:**

   In exhaustive search, one attempts to solve the problem by trying all possible keys in the key space. This can be done by computing all successive multiples of P : $2P, 3P, 4P, \ldots$. This method takes up to n steps, where n is the order of the point p .

2. **Baby-Step Giant-Step Algorithm:**

   This is a time-memory trade-off version of the exhaustive search method. It requires storage for about $\sqrt{n}$ points, and its running time is roughly $\sqrt{n}$ steps in the worst case.

3. **Pollard's Rho Algorithm:**

   This algorithm is a randomized version of the baby-step giant-step algorithm. It has roughly the same running time ($\frac{\sqrt{\pi n}}{2}$ steps) as the baby-step giant-step algorithm, but is

superior in that it requires a negligible amount of storage. Van Oorschot and Wiener [10] showed how Pollard's Rho algorithm can be parallelized so that when the algorithm is run in parallel on r processors, the expected running time of the algorithm is roughly $\frac{\sqrt{\pi n}}{2r}$ steps. At present, the parallelized version of Pollard's Rho algorithm is the fastest general-purpose method for solving the ECDLP.

4. **Multiple Logarithms:**

   Silverman and Stapleton [11] observed that if a single instance of the ECDLP (for a given elliptic curve and base point P) is solved, then the next instance for the same curve and the same base point can be solved more easily. More precisely, if solving the first instance takes expected time t , solving the second and third instances takes $(\sqrt{2}-1)t$ and $(\sqrt{3}-\sqrt{2})t$, respectively.

2.5.1.2 Special-Purpose Attacks

- **MOV Attack:**

  Menezes, Okamoto and Vanstone (MOV) [12], showed how, under mild assumptions, the ECDLP in an elliptic curve defined over a finite field $\mathbb{F}_q$ can be reduced to the DLP in some extension field $\mathbb{F}_{q^B}$ for some $B > 1$. This reduction is only useful when B is a small number (less than $\log^2(q)$). Balasubramanian and Koblitz [13] showed that for most elliptic curves, B is not a small number. However, for a very special class of elliptic curves (known as supersingular curves), it is known that $B \leq 6$. For these curves the MOV reduction gives a subexponential-time algorithm for solving ECDLP. For this reason supersingular curves are excluded from use in elliptic curve cryptosystems.

- **Prime Field Anomalous Attack:**

  Semaev [14], Smart [15], and Satoh and Araki [16] independently showed that it is easy to solve EDLP for a special class of elliptic curves called anomalous elliptic curves. An anomalous elliptic curve over $\mathbb{F}_q$ is an elliptic curve which has exactly q points.

- **Pollard's Rho Attack for Koblitz Curves:**

  Gallant, Lambert and Vanstone [17], and Wiener and Zuccherato [18] independently showed a way to speed up Pollard's Rho algorithm by a factor of $\sqrt{d}$ for solving ECDLP for elliptic curves over $\mathbb{F}_{q^{Bd}}$. For example for a Koblitz curve over $\mathbb{F}_{2^m}$ Pollard's Rho algorithm can be sped up by a factor of $\sqrt{m}$. In fact this factor is not a concern in practice since it is relatively small.

2.5.2 Attack Countermeasures

Table 2.10 summarizes the known attacks together with their countermeasures. An elliptic curve that satisfies all the countermeasure requirements in this table is considered intractable against all known attacks.

Attack	**Countermeasure**
Pohlig-Hellman	Select n to be prime
Pollard Rho	Select n to be a large number (at least 2^{160})
Multiple logarithms	Select n to be a large number (at least 2^{160})
MOV	Check that n does not divide $q^k - 1$ for all $1 \le k \le 20$
Prime field anomalous	Check that $n \neq q$
Weil descent	Do not use elliptic curves over composite binary fields or over F_{p^m} where p is odd and $m = 5$ or $m = 7$.(Conservative recommendations)

Table 2.10. Summary of ECDLP attacks and their countermeasures

Chapter 3

Software Implementation of ECC over *GF(p)*

3.1 Software Implementation Options

Elliptic curve cryptosystems offer a variety of implementation options. One of the main options is the choice of the underlying finite field. Figure 3.1 shows different finite fields that can be used to implement an elliptic curve cryptosystem.

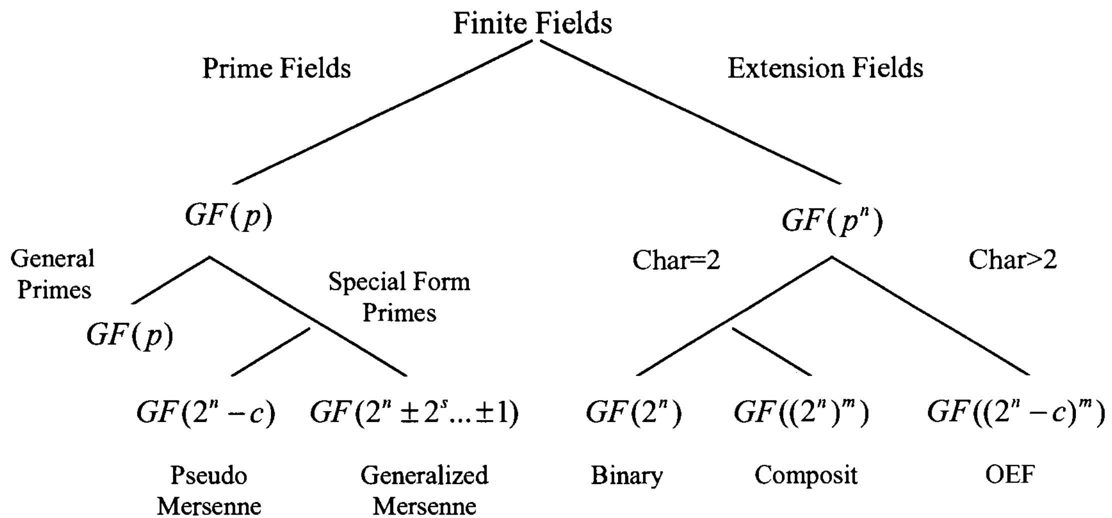

Figure 3.1. Underling finite field options

$GF(2^n)$ and $GF(p)$ are the two most common choices to implement elliptic curve cryptosystems. The case of $GF(2^n)$ or binary fields is especially attractive for hardware design. However, for software implementation of ECC they do not offer the same computational

advantages. This is because in software we have to work with units of data, called words, for computing field operations. For the large values of n, which is required for practical cryptosystems, we need to use several words to represent a field element. Thus, multiplication in $GF(2^n)$ can be very slow [19].

The next choice is $GF(p)$ or prime fields. As mentioned in the previous chapter, field inversion in $GF(p)$ is much slower than field multiplication. Therefore, it is preferred to use projective coordinates to eliminate field inversions for computing point addition and doubling. Similar to the binary fields, field addition and field subtraction in prime fields can be computed easily. So, the remaining problems are to find efficient modular multiplication and reduction algorithms. One approach to speed up reduction is to use $GF(p)$ where p is a Mersenne or Mersenne-like prime [20], e.g., NIST prime fields.

The last choice of the underlying finite fields we consider here is the choice of Optimal Extension Fields (OEF) first introduced in [21]. Optimal extension fields are the fields of the form $GF(p^m)$, $p > 2$. OEFs appear to offer performance advantages, in terms of overall performance and storage memory requirements, over binary and prime finite fields [22]. Field inversion can be implemented efficiently in an OEF. Thus, the ratio I/M is small enough to use affine coordinates which require 33% less storage compared to projective coordinates. On the other hand, we should note that the GHS Weil descent attack may succeed for some elliptic curves over $GF(p^m)$ with $m = 5$ or $m = 7$. However, more research is needed to conclude this with certainty.

The next option is the choice of field representation. The field representation can have a significant impact on elliptic curve cryptosystem performance. Note that the choice of field representation does not appear to affect the system security. For the case of $GF(2^n)$ there are different bases available to represent field elements. Two common families of bases to represent elements of $GF(2^n)$ are polynomial basis and normal basis. These bases specify how a bit string is to be interpreted. In all these bases, field addition and field subtraction are realized by a bit-wise exclusive OR. However, the structure of the modular multiplication and inversion is determined by the choice of bases for the representation.

In a polynomial basis, the basis elements are successive powers of an element α, namely $\alpha^0, \alpha^1, \alpha^2, ..., \alpha^{n-1}$. Polynomial basis is usually used for software implementation of ECC since they can provide relatively faster field multiplication and field inversion [23].

In a Normal Basis (NB), the basis elements are successive exponentiations of an element β, namely $\beta^{2^0}, \beta^{2^1}, \beta^{2^2}, ..., \beta^{2^{n-1}}$. Squaring a field element in NB is accomplished by a simple rotation of the binary vector representation. This can be implemented easily in hardware. However, multiplication in NB is more complicated. NB can be optimized to Optimal Normal Basis (ONB) for some values of n. ONB allows for efficient hardware implementation of modular multiplication.

In $GF(p)$ the elements are the integers between 0 and $p-1$. In software implementation, we store a field element in an array of words. A typical base to represent elements of $GF(p)$ is base 2^w, where w denotes the word size. The advantage of using this base is that it requires a minimum number of words to represent a field element. However, the result of multiplication of two base integers does not fit into a word. This may result in an inefficient field multiplication when multiplication is implemented using a language like C.

The second choice is a base of half the word size, i.e. $2^{w/2}$. The advantage of this base over the word size base is that the result of multiplying two base integers will still fit into a word. However, we need twice as many words to represent a field element using this base. This will result in more iterations to compute even a simple operation such as field addition and the situation is worse for algorithms with non-linear complexity, such as modular multiplication.

The third option is the choice of a suitable elliptic curve. One choice is to use a randomly generated elliptic curve. In the generation process, we should avoid certain classes of weak elliptic curves such as anomalous curves. Fortunately, each of these classes of weak curves is easy to identify. Generating a suitable elliptic curve can be done in a number of different ways. One way to find a curve is a random approach in which the parameters a and b of the elliptic curve are chosen randomly. If it is turn out that the curve is not suitable, another pair of parameters is chosen. The second way to find a suitable elliptic curve is to use the complex multiplication approach [7]. In this approach, first, a good candidate for the group order is found then parameters a and b of the curve is determined. However, generating a suitable elliptic curve can be complex. So to avoid it one can use elliptic curves which have been verifiably generated at random. Randomly generated elliptic curves are the safest choices when choosing elliptic curves.

Their main drawback is that they may not allow an efficient implementation of point multiplication (defined in Section 3.4.3), compared to some special classes of elliptic curves.

The second choice of a suitable elliptic curve is to use certain special classes of elliptic curves. These elliptic curves allow for faster implementation of point multiplication, hence improving the cryptosystem performance. Examples of such elliptic curves are the curves over $GF(p)$ for which the parameter a is equal to -3. As mentioned in the previous chapter, an elliptic curve over $GF(p)$ with $a=-3$ yields a faster algorithm for point doubling when Jacobian coordinates are used. Furthermore, this choice is still quite general since about half of all isomorphism classes of elliptic curves over $GF(p)$ have a representative with $a=-3$ [7].

Other examples of special type of elliptic curves are Koblitz curves first suggested by Koblitz [24]. These are the curves

$$y^2+xy=x^3+x^2+1$$

and

$$y^2+xy=x^3+1$$

over $GF(2^n)$. The primary advantage of Koblitz curves is that point multiplication algorithms can be devised to use Frobenius endomorphism instead of point doubling. This technique can be generalized to use an arbitrary endomorphism but they are generally not efficient [25].

There are other special elliptic curves with different advantages. For example, Montgomery-form elliptic curves [26] are easier to protect against information leakage attacks (attacks, which use observation such as timings or power consumption measurements in order to obtain secretes information).

Figure 3.2 summarizes the possible elliptic curve cryptosystems based on the choice of finite field and elliptic curve.

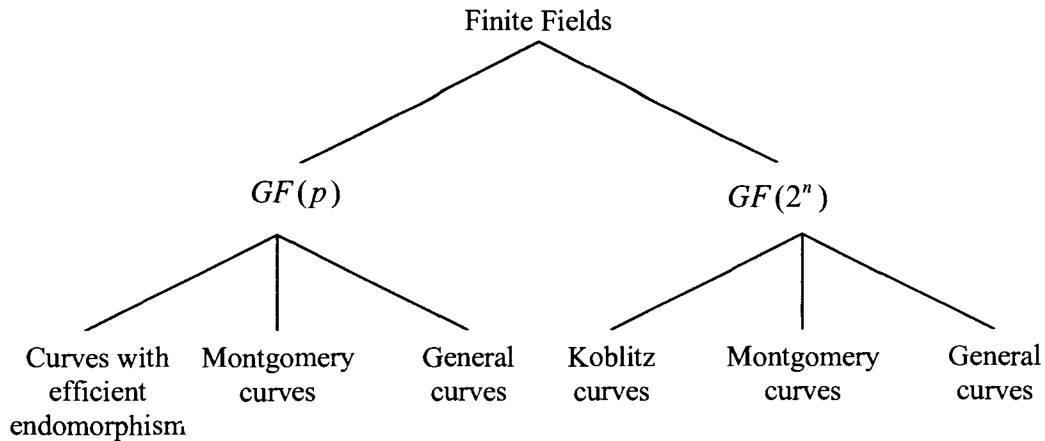

Figure 3.2. Elliptic curve options

There are also other implementation choices such as algorithms for field arithmetic and elliptic curve arithmetic that have to be made. A practical question to ask is whether there is a best set of choices. In fact it is difficult, if not impossible, to find a best set due to the different security considerations, application platforms (software or hardware), computing environments and engineering constraints such as memory, power and bandwidth requirements.

3.2 NIST-Recommended Elliptic Curves over Prime Fields

The NIST (National Institute of Standards and Technology) recommended a certain set of elliptic curves to use. These curves can be divided into two groups: a group of elliptic curves over $GF(2^n)$ and a group of the elliptic curves over $GF(p)$. The NIST elliptic curves over prime fields are listed in Table 3.1. The curves in this table are of the form

$$y^2 = x^3 - 3x + b$$

with an appropriate b chosen randomly and $a = -3$. As mentioned earlier, setting a equal to -3 yields a faster algorithm for point doubling when Jacobian coordinates are used. The primes p for $GF(p)$ were also selected to be a generalized Mersenne prime for which modular reduction can be carried out more efficiently than with general primes. In Table 3.1, the number of points on E defined over $GF(p)$ is nh, where n is a prime, and h is called the co-factor.

$p_{192} = 2^{192} - 2^{64} - 1,\ a = -3,\ h = 1,$

$b =$ 0x 64210519 E59C80E7 0FA7E9AB 72243049 FEB8DEEC C146B9B1

$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF 99DEF836 146BC9B1 B4D22831

$p_{224} = 2^{224} - 2^{96} + 1,\ a = -3,\ h = 1,$

$b =$ 0x B4050A85 0C04B3AB F5413256 5044B0B7 D7BFD8BA 270B3943 2355FFB4

$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFF16A2 E0B8F03E 13DD2945 5C5C2A3D

$p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1,\ a = -3,\ h = 1,$

$b =$ 0x 5AC635D8 AA3A93E7 B3EBBD55 769886BC 651D06B0 CC53B0F6 3BCE3C3E 27D2604B

$n =$ 0x FFFFFFFF 00000000 FFFFFFFF FFFFFFFF BCE6FAAD A7179E84 F3B9CAC2 FC632551

$p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1,\ a = -3,\ h = 1,$

$b =$ 0x 3312FA7 E23EE7E4 988E056B E3782D19 181D9C6E FE814112 0314088F 5013875A C656398D 8A2ED19D 2A85C8ED D3EC2AEF

$n =$ 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF C7634D81 F4372DDF 581A0DB2 48B0A77A ECEC196A CCC52973

$p_{521} = 2^{521} - 1,\ a = -3,\ h = 1,$

$b =$ 0x 00000051 953EB961 8E1C9A1F 929A21A0 B68540EE A2DA725B 99B315F3 B8B48991 8EF109E1 56193951 EC7E937B 1652C0BD 3BB1BF07 3573DF88 3D2C34F1 EF451FD4 6B503F00

$n =$ 0x 000001FF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFA 51868783 BF2F966B 7FCC0148 F709A5D0 3BB5C9B8 89C47AE BB6FB71E 91386409

Table 3.1. NIST-recommended elliptic curves over prime fields

Table 3.2 shows another elliptic curve similar to the NIST curves. Parameter a for this curve is -3 and p is a generalized Mersenne prime

$p_{128} = 2^{128} - 2^{97} - 1,\ a = -3,\ h = 1,$
$b = 0x$ $E87579C1$ $1079F43D$ $D824993C$ $2CEE5ED3$
$n = 0x$ $FFFFFFFF$ 00000000 $75A30D1B$ $9038A115$

Table 3.2. An elliptic curve similar to the NIST-recommended elliptic curves

3.3 Prime Field Arithmetic

Efficient implementation of finite field arithmetic operations is crucial to achieve an efficient implementation of ECC. These operations include modular addition, modular subtraction, modular multiplication, modular squaring, modular reduction, and modular inversion. The field operations of modular addition and modular subtraction are relatively fast and easily implemented. On the other hand, field inversion in $GF(p)$ is very slow and is usually avoided by using point projective coordinates representation. Modular multiplication, squaring and reduction are also time consuming operations and should be fast to obtain adequate cryptosystem performance. In this section we will present algorithms for arithmetic in $GF(p)$. For simplicity, we assume that the implementation platform has a 32-bit architecture.

3.3.1 Field Representation

As mentioned earlier, two different bases to represent field elements in $GF(p)$ are the base of the word size and the base of half the word size. Let $m = \lceil \log_2(p) \rceil$ and $t = \lceil m/32 \rceil$. In base 2^{32} every integer d can be represented as

$$d = \sum_{i=0}^{t-1} a_i 2^{32i} .$$

Therefore, using this base every element of $GF(p)$ can be stored in an array of length t

$$(a_{t-1},...,a_2,a_1,a_0),\text{ where } 0 \le a_i < 2^{32}.$$

We can also represent an integer in the base of half the word size, namely in base 2^{16}

$$d = \sum_{i=0}^{2t-1} a_i 2^{16i}.$$

In this case, we need an array of length $2t$ to store a field element

$$(a_{2t-1},...,a_2,a_1,a_0),\text{ where } 0 \le a_i < 2^{16}.$$

The problem of using the base of the word size is that the result of multiplication of two a_i's does not fit in a word. In contrast to the base of the word size, in the base of half the word size, multiplication of two a_i's can be performed more easily in a language like C. However, we need twice as many iterations to carry out linear operations like field addition. For operations with non-linear complexity such as field multiplication, we need even more iterations and the situation gets worse the larger the finite field used.

The problem of word multiplication in the base of word size can be alleviated by implementing the word multiplication function in a small amount of assembly code.

3.3.2 Addition and Subtraction

Addition and subtraction are the simplest finite field operations. Algorithm 1 [27] shows how to add two elements a and b in $GF(p)$ by first adding the corresponding words from index 0 to index $t-1$ and then subtracting p if the result is greater than $p-1$. Notice that in Step 2 of the algorithm we need to consider the carry bit of the previous word addition. In the C language, we do not necessarily have access to the carry bit so we may need to write code to cope with the carry being needed. However, processors such as the Intel Pentium family offer an "add-with-carry" instruction. Therefore, one can use this instruction by including code written in assembly language inside the program to speed up the field addition operation. Modular subtraction

(Algorithm 2 [27]) can be implemented in a similar way as modular addition. However, the carry bit is replaced with a borrow bit. Similar to the addition operation, we are able to use processor specific instructions to speed up the subtraction operation.

Algorithm 1. Modular addition

Input: A modulus p , and integers $a,b \in [0, p-1]$.

Output: $c = (a+b) \bmod p$.

1. $c_0 \leftarrow \text{Add}(a_0, b_0)$.
2. For i from 1 to $t-1$ do $c_i \leftarrow \text{Add_with_carry}(a_i, b_i)$.
3. If the carry bit is set then subtract p from $(c_{t-1}, \ldots, c_2, c_1, c_0)$.
4. If $c \geq p$ then $c \leftarrow c - p$.
5. Return (c).

Algorithm 2. Modular subtraction

Input: A modulus p , and integers $a,b \in [0, p-1]$.

Output: $c = (a-b) \bmod p$.

1. $c_0 \leftarrow \text{Subtract}(a_0, b_0)$.
2. For i from 1 to $t-1$ do: $c_i \leftarrow \text{Subtract_with_carry}(a_i, b_i)$.
3. If the carry bit is set then add p to $(c_{t-1}, \ldots, c_2, c_1, c_0)$.
4. Return (c).

It is worth noting that there are different ways to implement these algorithms in a language like C. In fact, there is a trade-off between memory and speed for implementing even these simple algorithms. Tables 3.4, 3.5, and 3.6 show different implementations of Steps 1 and 2 of Algorithm

1 in the C language and Table 3.7 compares their speed and code size. The definitions used in the C Programs are listed in Table 3.3.

```
typedef unsigned long    NumWord;

typedef NumWord          Num      [t];
```

Table 3.3. Definitions used in the C programs

```
void numAdd1(Num a , Num b , Num result ){
    int i ;
    wordAdd(a[0], b[0], &result[0]);
    for (i =1; i <t ; i ++){
        wordAddCarry(a[i ], b[i ], &result[0]);
    }
}
```

Table 3.4 First implementation of Steps 1 and 2 of Algorithm 1

```
void numAdd2(Num a , Num b , Num result ){
    int i ; NumWord temp ;
    temp = a[0]+b[0];

    if (temp < a[0]){
        result[0] = temp ; temp = a[1]+b[1]; temp ++ ;
    }else{
        result[0] = temp ; temp = a[1]+b[1];
    }

    for (i = 1; i < (t - 1); ){
        if ((temp < a[i ]) || ((temp == a[i ]) & &(b[i ]))){
            result[i ++] = temp ;
            temp = a[i ]+b[i ];
            temp ++ ;
        }else{
            result[i ++] = temp ;
            temp = a[i ]+b[i ];
        }
    }

    carryFlag = (temp < a[i ]) || ((temp == a[i ]) & &(b[i ]))?1 : 0;
    result[i ] = temp ;
}
```

Table 3.5. Second implementation of Steps 1 and 2 of Algorithm 1

```
void numAdd3(Num a, Num b, Num result){
    int i = 1; NumWord temp;

    temp = a[0]+b[0];
    if(temp < a[0]){
        result[0] = temp; temp = a[1]+b[1]; temp++;
    }else{ result[0] = temp; temp = a[1]+b[1];}

    if((temp < a[1]) || ((temp == a[1]) && (b[1]))){
        result[1] = temp; temp = a[2]+b[2]; temp++;
    }else{ result[1] = temp; temp = a[2]+b[2];}

    if((temp < a[2]) || ((temp == a[2]) && (b[2]))){
        result[2] = temp; temp = a[3]+b[3]; temp++;
    }else{ result[2] = temp; temp = a[3]+b[3];}

                                        ⋮

    if((temp < a[t-2]) || ((temp == a[t-2]) && (b[t-2]))){
        result[t-2] = temp; temp = a[t-1]+b[t-1]; temp++;
    }else{ result[t-2] = temp; temp = a[t-1]+b[t-1];}

    carryFlag = (temp < a[t-1]) || ((temp == a[t-1]) && (b[t-1])) ? 1 : 0;
    result[t-1] = temp;
}
```

Table 3.6. Third implementation of Steps 1 and 2 of Algorithm 1

Note that in the third implementation, we used loop unrolling technique to speed up the modular addition. Table 3.7 presents the code size and timing results for the addition algorithm

implementations over $GF(p_{192})$. To get the results in Table 3.7, we used the GCC compiler to compile the code for a Pentium II 448.81 MHz workstation.

	Code size (bytes)	**Timings (in μs)**
First Implementation (Table 3.4)	107	0.7
Second Implementation (Table 3.5)	407	0.4
Third Implementation (Table 3.6)	619	0.3

Table 3.7. Speed and memory trade-off for implementing modular addition

3.3.3 Multiplication and Squaring

The most challenging parts of implementing arithmetic in $GF(p)$ are modular multiplication, reduction, and Inversion. Since there is no satisfactory algorithm for modular inversion in $GF(p)$, it is often avoided by using projective coordinates for representing elliptic curve points. In the next section we will show how modular reduction can be carried out using a few field additions and subtractions when p is a generalized Mersenne prime. The remaining operation in this case is then modular multiplication. Hence, it is very important to implement it as efficient as possible.

The most crucial part of modular multiplication implementation is the word multiplication implementation. When a base of word size is used for representing field elements, the result of multiplying two coefficients does not fit into a word. For example, in a language like C, we do not have a command for 32×32 multiplication. In this case, we can write a function which may use an algorithm like Algorithm 3, to implement the operation of multiplying two full word size integers.

Algorithm 3. Word Multiplication

Input: Integers a and b $0 \le a,b < 2^{32}$

Output: Integers u and v such that $0 \le u,v < 2^{32}$ and $u2^{32} + v = ab$

1. $a_l \leftarrow a \,\&\, (2^{16} - 1)$, $a_h \leftarrow (a >> 16)$.
2. $b_l \leftarrow b \,\&\, (2^{16} - 1)$, $b_h \leftarrow (b >> 16)$.
3. $u \leftarrow a_h b_h$, $v \leftarrow a_l b_l$.
4. $c \leftarrow a_h b_l$, $c_1 \leftarrow (c << 16)$, $c_2 \leftarrow (c >> 16)$.
5. $d \leftarrow a_l b_h$, $d_1 \leftarrow (d << 16)$, $d_2 \leftarrow (d >> 16)$.
6. Add(v, c_1), Add-with-carry(u, c_2).
7. Add(v, d_1), Add-with-carry(u, d_2).

However, it is highly recommended to use a 32×32 multiply instruction (when the processor provides such an instruction) or to implement the function in a small amount of assembly language. As shown in Chapter 4, the trouble of adding a few lines of code in assembly language into the program for every target architecture, is a small price to pay for the large increase in speed which results.

The next step for implementing efficient modular multiplication is to choose a suitable multiplication algorithm. One of the simplest modular multiplication algorithms is the classical multiplication algorithm (Algorithm 4) [27].

Algorithm 4. Classical integer multiplication

Input: Integers $a,b \in [0, p-1]$.

Output: $c = ab$

1 $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$.

2 For k from 0 to $2(t-1)$ do

2.1 For each element of $\{(i,j) \mid i+j=k,\ 0 \le i,j < t\}$ do

$(uv) \leftarrow a_i b_j$.

$r_0 \leftarrow$ Add(r_0,v), $r_1 \leftarrow$ Add-with-carry(r_1,u), $r_2 \leftarrow$ Add-with-carry(r_2,0).

2.2 $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$.

3 $c_{2t-1} \leftarrow r_0$.

4 Return (c).

To multiply two t-word field elements using this algorithm, one needs about t^2 word operations. Therefore, the time complexity of multiplying two t-word field elements using this algorithm is $O(t^2)$.

For the cases where word multiplication is complex (i.e. when word multiplication is written in a language like C), the Karatsuba multiplication algorithm [28] can be employed. This algorithm proceeds as follows. Assume that we want to multiply two integers $a,b \in [0, p-1]$ and suppose $t = 2m$ is even (if not, we add a zero word at the left end). We can write

$$a = a_1 2^m + a_0$$

$$b = b_1 2^m + b_0$$

with m-word integers a_1, a_0, b_1, b_0. The product is given by

$$a.b = a_1 b_1 2^{2m} + (a_1 b_0 + a_0 b_1) 2^m + a_0 b_0 .$$

So we need to compute the integers $a_1 b_1$, $a_1 b_0 + a_0 b_1$, $a_0 b_0$. The idea of the Karatsuba algorithm is that this can be done with three rather than four multiplications, because $a_1 b_0 + a_0 b_1$ can be computed using only one multiplication if we use the results of the $a_1 b_1$ and $a_0 b_0$ multiplications

$$a_1 b_0 + a_0 b_1 = (a_1 + a_0)(b_1 + b_0) - a_1 b_1 - a_0 b_0 .$$

if $T(t)$ denotes the time it takes to multiply two t-word integers with the Karatsuba algorithm, then

$$T(t) = 3T(t/2) + O(t) .$$

This equation can be solved, giving a time complexity of $O(n^{1.585})$ which is much better than the time complexity of classical multiplication algorithm. However, the Karatsuba algorithm is recursive and more complex. There are also faster modular multiplication algorithms in terms of time complexity. For example, using a Fourier Transform techniques we can reach time complexity of $O(n \ln(n) \ln(\ln(n)))$ [29]. However, for the size of integers used in elliptic curve cryptography, it is more efficient in practice to use the classical multiplication algorithm [22].

We can use all these modular multiplication methods to compute modular squaring. However, because modular squaring is faster than modular multiplication, it is often implemented separately to speed up the cryptosystem. For example, in algorithm 4, when both inputs, a and b, are equal we do not need to compute both $a_i b_j$ and $a_j b_i$ as they are equal. Therefore, for squaring we can use Algorithm 5 [27] which needs approximately 50% fewer word multiplications.

Algorithm 5. Classical integer squaring

Input: Integer $a \in [0, p-1]$.

Output: $c = a^2$

1. $r_0 \leftarrow 0$, $r_1 \leftarrow 0$, $r_2 \leftarrow 0$.
2. For k from 0 to $2(t-1)$ do
   2.1 For each element of $\{(i,j) \mid i+j=k,\ 0 \le i,j < t\}$ do
      2.1.1 $(uv) \leftarrow a_i a_j$.
      2.1.2 If $(i < j)$ then $(uv) \ll 1$, $r_2 \leftarrow$ Add-with-carry(r_2,0).
      2.1.3 $r_0 \leftarrow$ Add(r_0,v), $r_1 \leftarrow$ Add-with-carry(r_1,u), $r_2 \leftarrow$ Add-with-carry(r_2,0).
   2.2 $c_k \leftarrow r_0$, $r_0 \leftarrow r_1$, $r_1 \leftarrow r_2$, $r_2 \leftarrow 0$.
3. $c_{2t-1} \leftarrow r_0$.

Return (c).

3.3.4 Modular Reduction

Modular reduction can be sped up using primes with a special form. The Mersenne primes (primes of the form $p = 2^k - 1$), are good examples. Modular reduction by a Mersenne prime can be carried out using one modular addition. Unfortunately, Mersenne primes are very rare so we are not able to use them for most cases. However, we can use generalized forms of Mersenne primes for which two types are given below.

The first type has the form $2^k - c$ where c is a small integer for which $0 < |c| < 2^{\lfloor k/2 \rfloor}$. A prime with this form is called pseudo-Mersenne prime. Modular reductions by pseudo-Mersenne primes can be efficiently computed using a few multiplications by a small integer c. The second type of such generalized forms is generalized-Mersenne primes. A generalized-Mersenne

prime has the form $p = f(2^k)$ where $f(t)$ is a low-degree polynomial with small integer coefficients. Modular reduction by generalized-Mersenne primes can be very fast since it requires a small number of modular addition/subtraction and some bit shifts. In practice, k is a multiple of the word size to eliminate the need for shifting bits.

In the NIST standard, the primes are taken to be generalized-Mersenne primes. To make modular reduction faster, the coefficients of $f(t)$ for the NIST primes are 1. Furthermore, k is a multiple of the word size for four (out of five) NIST primes. Therefore, modular reduction for the NIST primes can be done efficiently using Algorithms 7-11 [27]. Notice that $p_{128} = 2^{128} - 2^{97} - 1$ is not a NIST prime but it has the same properties as the NIST primes. A fast reduction algorithm for p_{128} is given in Algorithm 6.

Algorithm 6. Fast reduction modulo $p_{128} = 2^{128} - 2^{97} - 1$

Input: Integer $c = (c_7, \ldots, c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{128}^2$.

Output: $c \bmod p_{128}$

1. Define 128-bit integers: $s_1 = (c_3, c_2, c_1, c_0)$, $s_2 = (c_7, c_6, c_5, c_4)$, $s_3 = (c_4, c_7, c_6, c_5)$, $s_4 = (c_5, 0, c_7, c_6)$, $s_5 = (c_6, 0, 0, c_7)$, $s_4 = (c_7, 0, 0, 0)$.
2. Return ($s_1 + s_2 + 2s_3 + 4s_4 + 8s_5 + 16s_6 \bmod p_{128}$).

Algorithm 7. Fast reduction modulo $p_{192} = 2^{192} - 2^{64} - 1$

Input: Integer $c = (c_5, c_4, c_3, c_2, c_1, c_0)$ where each c_i is a 64-bit word, and $0 \le c < p_{192}^2$.

Output: $c \bmod p_{192}$.

1. Define 192-bit integers: $s_1 = (c_2, c_1, c_0)$, $s_2 = (0, c_3, c_3)$, $s_3 = (c_4, c_4, 0)$, $s_4 = (c_5, c_5, c_5)$.
2. Return ($s_1 + s_2 + s_3 + s_4 \bmod p_{192}$).

Algorithm 8. Fast reduction modulo $p_{224} = 2^{224} - 2^{96} + 1$

Input: Integer $c = (c_{13}, ..., c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{224}^2$.

Output: $c \bmod p_{224}$.

1. Define 224-bit integers: $s_1 = (c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,

   $s_2 = (c_{10}, c_9, c_8, c_7, 0, 0, 0)$, $s_3 = (0, c_{13}, c_{12}, c_{11}, 0, 0, 0)$,

   $s_4 = (c_{13}, c_{12}, c_{11}, c_{10}, c_9, c_8, c_7)$, $s_5 = (0, 0, 0, 0, c_{13}, c_{12}, c_{11})$.

2. Return ($s_1 + s_2 + s_3 - s_4 - s_5 \bmod p_{224}$).

Algorithm 9. Fast reduction modulo $p_{256} = 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1$

Input: Integer $c = (c_{15}, ..., c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{256}^2$.

Output: $c \bmod p_{256}$.

1. Define 256-bit integers: $s_1 = (c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,

   $s_2 = (c_{15}, c_{14}, c_{13}, c_{12}, c_{11}, 0, 0, 0)$, $s_3 = (0, c_{15}, c_{14}, c_{13}, c_{12}, 0, 0, 0)$,

   $s_4 = (c_{15}, c_{14}, 0, 0, 0, c_{10}, c_9, c_8)$, $s_5 = (c_8, c_{13}, c_{15}, c_{14}, c_{13}, c_{11}, c_{10}, c_9)$,

   $s_6 = (c_{10}, c_8, 0, 0, 0, c_{13}, c_{12}, c_{11})$, $s_7 = (c_{11}, c_9, 0, 0, c_{15}, c_{14}, c_{13}, c_{12})$,

   $s_8 = (c_{12}, 0, c_{10}, c_9, c_8, c_{15}, c_{14}, c_{13})$, $s_9 = (c_{13}, 0, c_{11}, c_{10}, c_9, 0, c_{15}, c_{14})$.

2. Return ($s_1 + 2s_2 + 2s_3 + s_4 + s_5 - s_6 - s_7 - s_8 - s_9 \bmod p_{256}$).

Algorithm 10. Fast reduction modulo $p_{384} = 2^{384} - 2^{128} - 2^{96} + 2^{32} - 1$

Input: Integer $c = (c_{23}, \ldots, c_2, c_1, c_0)$ where each c_i is a 32-bit word, and $0 \le c < p_{384}^2$.

Output: $c \bmod p_{384}$.

1. Define 384-bit integers: $s_1 = (c_{11}, c_{10}, c_9, c_8, c_7, c_6, c_5, c_4, c_3, c_2, c_1, c_0)$,

   $s_2 = (0,0,0,0,0,c_{23},c_{22},c_{21},0,0,0,0)$,

   $s_3 = (c_{23},c_{22},c_{21},c_{20},c_{19},c_{18},c_{17},c_{16},c_{15},c_{14},c_{13},c_{12})$,

   $s_4 = (c_{20},c_{19},c_{18},c_{17},c_{16},c_{15},c_{14},c_{13},c_{12},c_{23},c_{22},c_{21})$,

   $s_5 = (c_{19},c_{18},c_{17},c_{16},c_{15},c_{14},c_{13},c_{12},c_{20},0,c_{23},0)$, $s_6 = (0,0,0,0,c_{23},c_{22},c_{21},c_{20},0,0,0,0)$,

   $s_7 = (0,0,0,0,0,0,c_{23},c_{22},c_{21},0,0,c_{20})$,

   $s_8 = (c_{22},c_{21},c_{20},c_{19},c_{18},c_{17},c_{16},c_{15},c_{14},c_{13},c_{12},c_{23})$,

   $s_9 = (0,0,0,0,0,0,0,c_{23},c_{22},c_{21},c_{20},0)$, $s_{10} = (0,0,0,0,0,0,0,c_{23},c_{23},0,0,0)$.
2. Return ($s_1 + 2s_2 + s_3 + s_4 + s_5 + s_6 + s_7 - s_8 - s_9 - s_{10} \bmod p_{384}$).

Algorithm 11. Fast reduction modulo $p_{521} = 2^{521} - 1$

Input: A binary integer $c = (c_{1041}, \ldots, c_2, c_1, c_0)$.

Output: $c \bmod p_{521}$.

1. Define 521-bit integers: $s_1 = (c_{1041}, \ldots, c_{514}, c_{513}, c_{512})$, $s_1 = (c_{511}, \ldots, c_2, c_1, c_0)$.
2. Return ($s_1 + s_2 \bmod p_{521}$)

Modular reduction modulo the order of the base point is also required for implementing the Elliptic Curve Digital Signature Algorithm (ECDSA). The order of the base point for four NIST elliptic curves is a pseudo-Mersenne prime. However, to implement modular reduction modulo

the order of the base point for other elliptic curves, i.e. elliptic curves defined over $GF(p_{256})$ and $GF(p_{128})$, we need to use a general modular reduction algorithm such as the Barrett reduction algorithm [7].

To compute ($x \bmod p$) for $2^{w(t-1)} < p < 2^{wt}$, the Barrett algorithm needs μ precomputed where $\mu = \left\lfloor \frac{2^{2wt}}{p} \right\rfloor$. We can also use Montgomery reduction but it requires the operands to be converted to a special Montgomery form [7].

Algorithm 12. Barrett reduction

Input:.$b = 2^w$ where w is the word size, x , p and μ such that $x < b^{2t}$ and $\mu = \lfloor b^{2t}/p \rfloor$.

Output: $z = x \pmod p$.

1. $q_0 = \lfloor x/b^{t-1} \rfloor$.
2. $q = \lfloor (\mu q_0)/b^{t+1} \rfloor$.
3. $r_1 = x \pmod{b^{t+1}}$, $r_2 = qp \pmod{b^{t+1}}$.
4. $z \leftarrow r_1 - r_2$.
5. If $z < 0$ then $z \leftarrow z + b^{t+1}$.
6. While $z \geq p$ do: $z \leftarrow z - p$.
7. Return z .

3.3.5 Modular Inversion

As mentioned before, modular inversion is a very slow operation in $GF(p)$ so it is often avoided by using projective coordinates to represent elliptic curve points. However, a modular inversion is required in some operations such as converting the projective coordinates to affine coordinates, digital signature generation, and digital signature verification.

A general algorithm for computing modular inversion is the Extended Euclidean Algorithm (EEA). To find the inversion of $a \in GF(q)$, the EEA computes the coefficients s and t such that

$$as + qt = \gcd(a, q) = 1$$

The parameter s will be the inversion of a in $GF(q)$. Note that a and q must be relatively prime in order for the inverse to be exist.

However, the EEA involves integer divisions, which are quite slow. To avoid divisions, we can use the binary Euclidean algorithm, a variant of the EEA algorithm. The binary Euclidean algorithm only requires shifts and additions. However, it has more iterations.

To compute $a^{-1} \bmod p$, the binary Euclidean algorithm (Algorithm 13) [27], maintains the equations $Aa + dp = u$ and $Ca + ep = v$ and decrease the values v and u in every iterations until u becomes 0, in which case $v = 1$ and $Ca + ep = 1$; hence $C = a^{-1} \bmod p$.

Algorithm 13. Binary inversion algorithm

Input: Prime p, $a \in [1, p-1]$.

Output: $a^{-1} \bmod p$.

1 $u \leftarrow a$, $v \leftarrow p$, $A \leftarrow 1$, $C \leftarrow 0$.

2 While $u \neq 0$ do:

2.1 While u is even do:

$u \leftarrow u/2$. If A is even then $A \leftarrow A/2$; else $A \leftarrow (A + p)/2$.

2.2 While v is even do:

$v \leftarrow v/2$. If C is even then $C \leftarrow C/2$; else $C \leftarrow (C + p)/2$.

2.3 If $u \geq v$ then: $u \leftarrow u - v$, $A \leftarrow A - C$; else: $v \leftarrow v - u$, $C \leftarrow C - A$.

3 Return ($C \bmod p$).

3.4 Arithmetic on Elliptic Curves over $GF(p)$

To implement digital signature algorithms we need to implement elliptic curve point operations, namely point addition/subtraction, point doubling and point multiplication. In this section we explain arithmetic on elliptic curves over a field of prime characteristic.

3.4.1 Addition and Subtraction

In the previous chapter we showed how to add two points P and Q , $P \neq -Q$, in different coordinates. We also showed the addition formulas for these coordinates. For point subtraction we can use the same formulas due to the fact that $P - Q = P + (-Q)$. Therefore, subtraction can be performed by first computing the negation of point Q and then adding the result to P . The negation of the point $Q = (x, y)$ in affine coordinates is $-Q = (x, -y)$. Similar to the affine coordinates, the negation can be computed by simply negating the y -coordinate in the other coordinates defined in this thesis. For example, the negation of the point $Q = (x, y, z)$ in projective coordinates is $-Q = (x, -y, z)$. Therefore, to negate a point we require only a modular subtraction, which is a very fast operation in comparison to modular multiplication and inversion. Hence, point subtraction has the same cost as point addition.

Another way to compute point subtraction is to use slightly different formulas than those used for point addition. For example, assume that we want to compute $P - Q$, where the points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ are represented in affine coordinates. Table 3.8 shows the point subtraction formula, which is obtained by replacing y_2 with $-y_2$ in the point addition formula. In this case, we don't need to compute point negation. However, point subtraction would require a separate function than point addition.

Addition $(x_3,y_3)=(x_1,y_1)+(x_2,y_2)$	**Subtraction** $(x_3,y_3)=(x_1,y_1)-(x_2,y_2)$
$\lambda=\frac{y_2-y_1}{x_2-x_1}$ $x_3=\lambda^2-x_1-x_2$ $y_3=\lambda(x_1-x_3)-y_1$	$\lambda=\frac{y_2+y_1}{x_1-x_2}$ $x_3=\lambda^2-x_1-x_2$ $y_3=\lambda(x_1-x_3)-y_1$

Table 3.8. Point subtraction formula in affine coordinates

3.4.2 Point Doubling

As mentioned in the previous chapter, similar to point addition, we have different point doubling formulas in different coordinates. In each coordinate system, we have different formulas for point addition and point doubling since the usual point addition formulas cannot be used for point doubling. With the usual algorithms, point doubling is often much faster than point addition. However, for some cases, point doubling can be even slower than point addition. For example, when affine coordinates are used, point doubling requires two modular squarings, one modular multiplication and one modular inversion, while point addition requires one modular squaring, one modular multiplication and one modular inversion.

As Liardet and Smart [30] and Joye and Quisquater [31] have proposed, a single formula can be used for both point addition and point doubling by using special representation of points of certain elliptic curves over prime fields. Although, this approach induces some performance penalty, it reduces information leakage, as the secret key may be visible by tracing the power consumption.

3.4.3 Point Multiplication

Elliptic curve cryptographic schemes require the computation of kP defined as repeated addition via

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

where P is an elliptic curve point, k is an integer $k \in [1, n-1]$, and n is the order of the point P. The operation is known as scalar or point multiplication, and dominates the execution time of elliptic curve cryptographic schemes. In this section, we describe the methods for computing point multiplication.

3.4.3.1 Random Point Multiplication

When the point P is not known a priori, the point multiplication kP is called random point multiplication. There are various methods for speeding up this calculation. The two basic operations typically used in point multiplication methods are point doubling and point addition. Point subtraction can also be used since it typically has the same computational cost as addition. Most of the random point multiplication methods require approximately the same number of doubling operations, but they require different numbers of additions and/or subtractions.

The simplest (and oldest) efficient method for computing point multiplication is the binary method. This method uses the binary representation of k to compute kP, as shown in Algorithm 14 [7].

Algorithm 14. Binary method for point multiplication

Input: An integer $k = \sum_{0 \le i < l} k_i 2^i$, $k_i \in \{0,1\}$, $P \in E(\mathbb{F}_p)$.

Output: kP.

1. $Q \leftarrow o$.
2. For i from $l-1$ down to 0 do
   2.1 $Q \leftarrow 2Q$.
   2.2 If $k_i = 1$ then $Q \leftarrow Q + P$.
3. Return (Q).

This method requires $(l-1)$ point doublings and on average $l/2$ point additions. Although there are other more efficient methods for computing point multiplication, the binary method may be used for simplicity and minimizing temporary memory [32]. In this section, we describe how most of the known random point multiplication methods can be obtained by using different integer representations in the sliding window algorithm.

3.4.3.1.1 Integer Representations

To compute the point multiplication kP, the integer k can be represented as

$$k = \sum_{0 \le i < l} b_i 2^i ,$$

where $b_i \in B \cup \{0\}$ and B is a set of nonzero integers including 1. As mentioned earlier, elliptic curve subtraction has the same cost as addition, so the elements of B can be negative as well. There are various integer representations used in point multiplication methods. In this section, some of these are described.

3.4.3.1.1.1 Binary Representation

The binary representation is the simplest integer representation and has $B = \{1\}$. The expected weight (number of nonzero digits) of the binary representation of an integer of length l is $\frac{l}{2}$. Hence, the average density (the number of nonzero digits over the length of the integer) of this representation is $\frac{1}{2}$.

3.4.3.1.1.2 The Non-Adjacent Form Representation

The Non-Adjacent Form (NAF) is a signed binary representation of an integer with $B = \{1, -1\}$. In this representation, at most one of any two consecutive digits is nonzero. Moreover, each positive integer has a unique NAF representation with expected weight $\frac{l}{3}$. Thus, adding a negative digit to the binary representation reduces the average density from $\frac{1}{2}$ to $\frac{1}{3}$. The NAF representation of an integer can be efficiently computed using Algorithm 15.

Algorithm 15. Computing the NAF of a positive integer k

Input: A positive integer k.

Output: NAF(k).

1. $i \leftarrow 0$.
2. While $k \geq 1$ do
   - 2.1 If k is odd then $k_i \leftarrow 2 - (k \bmod 4)$, $k \leftarrow k - k_i$.
   - 2.2 Else $k_i \leftarrow 0$.
   - 2.3 $k \leftarrow k/2$, $i \leftarrow i + 1$.
3. Return $((k_{i-1}, k_{i-2}, \ldots, k_1, k_0))$.

3.4.3.1.1.3 The m-ary and Signed m-ary Representations

The m-ary representation (integer representation in base 2^w) is a generalized form of the binary representation. In the m-ary representation, $B = \{1,2,3,...,2^w - 1\}$. The average density of this representation is $\frac{1}{w}(1-\frac{1}{2^w})$, i.e., $(1-\frac{1}{2^w})$ is the probability that a digit in base 2^w representation is nonzero.

The signed m-ary representation (also called width-(w+1) NAF or wNAF) is a generalized form of the NAF representation. In the signed m-ary representation, $B = \{\pm1, \pm3, ..., \pm(2^w - 1)\}$. The average density of the signed m-ary representation is $\frac{1}{w+2}$ as $k \to \infty$ [33].

Algorithm 16. Computing the signed m-ary representation of a positive integer k

---

Input: A positive integer k .

Output: signed m-ary of k .

1. $i \leftarrow 0$.
2. while $k > 0$ do
   - 2.1 if k is odd then
     - 2.1.1 $b \leftarrow k \bmod 2^{w+1}$.
     - 2.1.2 If $b \geq 2^w$ then
       - 2.1.2.1 $b \leftarrow b - 2^{w+1}$.
     - 2.1.3 $k \leftarrow k - b$.
   - 2.2 Else
     - 2.2.1 $b \leftarrow 0$.
   - 2.3 $b_i \leftarrow b$, $i \leftarrow i + 1$.
   - 2.4 $k \leftarrow k/2$.
3. Return $((b_{i-1},...,b_1,b_0))$.

3.4.3.1.1.4 The KT and Katti Representations

Both the KT [34] and Katti [35] representations use the same set B as the NAF representation. The advantage of these two representations over NAF is that they have a higher average length of zero runs. The average length of the zero runs for the KT and Katti representations is the same [35], but the KT representation is more complex to obtain. The Katti representation can be obtained in three steps as shown below.

1. Convert the binary representation of $k = \sum_{0 \le i < l} b_i 2^i$ into

$$X = \sum_{0 \le i \le l} c_i 2^i \text{, where } c_i = \begin{cases} b_{i-1} - b_i & \text{if} \quad 0 < i < l \\ b_{i-1} & \text{if} \quad i = l \\ -b_0 & \text{if} \quad i = 0 \end{cases}.$$

2. Convert X into Y by going from left to right and replacing $1,-1$ by $0,1$ and $-1,1$ by $0,-1$.

3. Convert Y into Z by going from left to right and replacing $1,0,-1$ by $0,1,1$ and $-1,0,1$ by $0,-1,-1$.

3.4.3.1.1.5 A Simple Signed Binary Representation

A Simple Signed Binary Representation (SSBR) can be obtained from the binary representation of an integer simply by performing Step 1 of the Katti representation algorithm given in the previous section. This will be used with our proposed windowing algorithms to obtain efficient methods for computing single and multiple point multiplication. To convert a binary representation to SSBR, digits are scanned from left to right (i.e., from the most significant position to the least significant position). Therefore, a binary representation can be converted to SSBR in a pipelined fashion in the algorithms presented in the following sections.

3.4.3.1.2 Windowing Algorithms

In this section, we describe the Sliding Window Algorithm (SWA). Then we introduce a generalized sliding window algorithm. These algorithms are combined with integer representations to obtain point multiplication methods.

3.4.3.1.2.1 The Sliding Window Algorithm

The sliding window algorithm (SWA) is a well-known algorithm for point multiplication. In the precomputation stage of this algorithm, bP for all $b \in \{1,3,5,...,2^w - 1\}$, where w is a small positive integer called the window size, is computed and stored. Algorithm 17 is a modified sliding window algorithm (in the SWA, $s = 0$ and $e = h$ so s does not have to be determined as shown in line 3.2.3). For the representations described in the previous section (except for the m-ary representations), s in line 3.2.3 is always equal to 0, thus there is no need to compute s. If the SWA is used with the m-ary representation (m-ary method) we need to precompute and store all bP where $b \in \{1,2,3,4,...,2^w - 1\}$. However, employing s in the algorithm (line 3.2.3) results in a modified m-ary method which requires 50% less memory than the m-ary method to store precomputed points [7].

Algorithm 17. The modified sliding window algorithm

INPUT: A point P, an integer $k = \sum_{0 \le i < l} b_i 2^i$, $b_i \in B \cup \{0\}$, window size w.

OUTPUT: kP.

1. In the precomputation stage: Compute bP for all $b \in \{1, 3, ..., 2^w - 1\}$.
2. $i \leftarrow l - 1$, $Q \leftarrow o$.
3. While $i \ge 0$ do
   3.1 If $b_i = 0$ then $Q \leftarrow 2Q$, $i \leftarrow i - 1$.
   3.2 Else
      3.2.1 Let t be the least integer such that $i - t + 1 \le w$ and $b_t \ne 0$,
      3.2.2 $h \leftarrow (b_i b_{i-1} ... b_t)_2$.
      3.2.3 Let s, e be such that $h = 2^s e$, e odd.
      3.2.4 If $e > 0$ then $Q \leftarrow 2^{i-t-s+1} Q + eP$,
      3.2.5 Else $Q \leftarrow 2^{i-t-s+1} Q - (-e)P$,
      3.2.6 $Q \leftarrow 2^s Q$.
      3.2.7 $i \leftarrow t - 1$.
4. Return (Q).

Table 3.9 presents a classification of PM methods based on the sliding window algorithm with integer representations. For example, the binary representation with a window size of 1 gives the binary method. The methods in Table 3.9 are from [7].

Method	Window Size	Integer Representation	Set of all multipliers b for which b.P must be stored	Estimated average number of curve operations
Binary	1	Binary	$\{1\}$	$l+\frac{l}{2}-2$
NAF	1	NAF	$\{1\}$	$l+\frac{l}{3}-2$
Modified m-ary	1	m-ary	$\{1,3,...,2^w-1\}$	$l+\frac{l}{w}(1-\frac{1}{2^w})+2^{w-1}-2$
Signed m-ary	1	Signed m-ary	$\{1,3,...,2^w-1\}$	$l+\frac{l}{w+2}+2^{w-1}-2$
Sliding Window	w	binary	$\{1,3,...,2^w-1\}$	$l+\frac{l}{w+1}+2^{w-1}-2$
Sliding Window	w	NAF	$\{1,3,...,\frac{2}{3}(2^w-(-1)^w)-1\}$	$l+\frac{l}{w+\frac{4}{3}}+\frac{2^w}{3}-2$

Table 3.9. A classification of PM methods based on window size and integer representation

The KT and Katti representations are good choices for use with the SWA because they have a higher average length of zero runs. However, the Katti representation can be modified for use with the sliding window algorithms to improve performance. The modified Katti representation can be obtained in two steps as follows.

1. Convert the binary representation of $k=\sum_{0\le i<l} b_i 2^i$ into

$$X=\sum_{0\le i\le l} c_i 2^i \text{, where } c_i=\begin{cases} b_{i-1}-b_i & if \quad 0<i<l \\ b_{i-1} & if \quad i=l \\ -b_0 & if \quad i=0 \end{cases}.$$

2. Convert X to Y by going from left to right and replacing the sequences $1\ \underbrace{00..0}_{0\le i<w\ zeros}\ -1$ and $-1\ \underbrace{00..0}_{0\le i<w\ zeros}\ 1$ with $0\ \underbrace{11..1}_{0\le i<w\ zeros}\ 1$ and $0\underbrace{-1-1..-1}_{0\le i<w\ zeros}-1$, respectively, if these sequences are at the beginning of a window.

Step 2 is a generalization of Steps 2 and 3 in the Katti representation algorithm. In this step, we try to move windows to the right to reduce the number of windows. Notice that after doing step 2 for one window, it need not be repeated for that window. Simulation shows that for window size w, the average window density (the number of windows for an integer k) of the Katti representation is about $\frac{1}{w+1.5}$ as $k \to \infty$, while for the modified Katti representation it is $\frac{1}{w+2}$ as $k \to \infty$. Furthermore, the simulation shows that the modified Katti representation always results in at worst the same average window density as the Katti representation. To obtain these results, we used several million random numbers for each value of w, where $0 < w < 6$.
As an example, let $k = (1011011101010111111001011001 0001)$ and $w = 4$. The first step of the Katti algorithm (which is the same as the first step of the modified Katti algorithm), converts k into X as given below

$$X = (\text{1-110-1100-11-11-1100000-101-110-101-1001-1})$$

Steps 2 and 3 of the Katti representation convert X into Y and Z, respectively

$$Y = (\text{1100-1000-10-10-100000-100110-1001001})$$

$$Z = (\text{1100-1000-10-10-100000-10010110010001})$$

For the case $w = 4$, Z can be split into 8 windows with length at most w as follows (the windows are underlined)

$$Z = (\ \underline{11}00\underline{-1}000\underline{-10-1}0\underline{-1}00000\underline{-1001}0\underline{11}00\underline{1}000\underline{1}\ )$$

The modified Katti algorithm converts X into W which results in 6 windows as shown below

$$W = (\, \underline{110{-}1}\,0\,\underline{1111}\,0\,\underline{{-}1{-}11}\,000000\,\underline{{-}1{-}1{-}11}\,00\,\underline{{-}1{-}1{-}1}\,000\underline{1}\,)$$

This example shows how Step 2 of the modified Katti algorithm can reduce the number of windows.

3.4.3.1.2.2 The Generalized Sliding Window Algorithm

In the precomputation stage of the sliding window algorithm, 2^{w-1} points are stored. In some cases, there is sufficient memory to store m, $2^{w-1} < m < 2^{w}$, points. Since the SWA can only use 2^{w-1} points, the remainder of the available memory is wasted. However, in the generalized sliding window algorithm, we can store any number of points (up to the amount of available memory) to improve the efficiency of point multiplication.

In the precomputation stage of the proposed algorithm, bP for all $b \in \{1, 3, ..., 2m-1\}$, where m is the maximum number of points we want to store, are computed and stored. Note that the algorithm scans the integer representation from left to right. It initializes the window size to 0 and increases it until the absolute value of h (the number inside the window) is not greater than $(2m-1)$. Next hP is added to the last result, and this step is repeated starting from the next nonzero bit after the window.

Algorithm 18. The proposed generalized sliding window algorithm

---

INPUT: A point P, an integer $k = \sum_{0 \le i < l} b_i 2^i$, $b_i \in B \cup \{0\}$.

OUTPUT: kP

1. In the precomputation stage: Compute bP for all $b \in \{1,3,\ldots,2m-1\}$, where m is the maximum number of points we want to store
2. $i \leftarrow l-1$, $Q \leftarrow o$.
3. While $i \ge 0$ do
   - 3.1 If $b_i = 0$ then $Q \leftarrow 2Q$, $i \leftarrow i-1$.
   - 3.2 Else
     - 3.2.1 Let t be the least integer such that $|(b_i b_{i-1} \ldots b_t)_2| < 2m$ and $b_t \ne 0$,
     - 3.2.2 $h \leftarrow (b_i b_{i-1} \ldots b_t)_2$.
     - 3.2.3 Let s , e be such that $h = 2^s e$, e odd.
     - 3.2.4 If $e > 0$ then $Q \leftarrow 2^{i-t-s+1} Q + eP$,
     - 3.2.5 Else $Q \leftarrow 2^{i-t-s+1} Q - (-e)P$,
     - 3.2.6 $Q \leftarrow 2^s Q$.
     - 3.2.7 $i \leftarrow t-1$.
4. Return (Q).

A very simple method for point multiplication can be obtained by combining this algorithm and the SSBR representation. The estimated average number of curve operations using this method is $l + \frac{l}{w+2} + 2^{w-1} - 2$ with $B = \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$. Note that this is as good as the best method in Table 3.9, but the SSBR representation is simpler and allows the integer to be scanned from left to right.

3.4.3.2 Double Point Multiplication

Digital signature verification requires the computation of $gP + hQ$ for two points P and Q. The most obvious way of performing this computation is to compute gP and hQ separately and then add them together. If we use the binary method to compute gP and hQ then we would need about $2l$ doublings and on average l additions. Algorithm 19 [36] shows how the number of doublings can be reduced to l by performing the point multiplications gP and hQ simultaneously.

Algorithm 19. Simple double point multiplication

Input: positive integers $g = \sum_{i=0}^{l-1} g_i 2^i$ and $h = \sum_{i=0}^{l-1} h_i 2^i$, where $g_i, h_i \in \{0,1\}$, $P, Q \in E(\mathbb{F}_p)$.

Output: $gP + hQ$.

1. $A \leftarrow o$.
2. For i from $l-1$ down to 0 do
   - 2.1 $A \leftarrow 2A$.
   - 2.2 If $g_i = 1$ then $A \leftarrow A + P$.
   - 2.3 If $h_i = 1$ then $A \leftarrow A + Q$.
3. Return (A).

Algorithm 19 uses the binary representation of integers g and h to compute $gP + hQ$. However, by using Algorithm 20 (a generalization of Algorithm 19), we can employ other integer representations such as signed m-ary representation to reduce the number of additions.

Algorithm 20. A generalization of the simple double point multiplication algorithm

---

Input: positive integers $g = \sum_{i=0}^{l-1} g_i 2^i$ and $h = \sum_{i=0}^{l-1} h_i 2^i$, where $g_i, h_i \in \{0,1\}$, $P, Q \in E(\mathbb{F}_p)$.

Output: $gP + hQ$.

1. Precomputation stage: Compute mP and mQ for all positive integers m , $m \in B$.
2. $A \leftarrow o$.
3. For i from $l-1$ down to 0 do

   3.1 $A \leftarrow 2A$.

   3.2 If $g_i \neq 0$ then:

      3.2.1 If $g_i > 0$ then: $A \leftarrow A + g_i P$

      3.2.2 Else: $A \leftarrow A - (-g_i)P$

   3.3 If $h_i \neq 0$ then:

      3.3.1 If $h_i > 0$ then: $A \leftarrow A + h_i Q$

      3.3.2 Else: $A \leftarrow A - (-h_i)Q$

4. Return (A).

It is also possible to extend Algorithm 20 by combining it with the generalized sliding window algorithm to get Algorithm 21. This algorithm computes multiple point multiplication, namely $\sum_{1 \le i \le d} k_i P_i$ where k_i is a positive integers and $P_i \in E(\mathbb{F}_p)$.

Algorithm 21. A generalization of Algorithm 20

Input: : Points $P_1, P_2, ..., P_d$, and integers $k_1, k_2, ..., k_d$ where $k_i = \sum_{0 \le j < l} b_{i,j} 2^j$, $b_{i,j} \in B \cup \{0\}$.

Output:. $A = \sum_{1 \le i \le d} k_i P_i$

1. $A \leftarrow o$
2. For i from 1 to d do: $t_i \leftarrow l - 1$
3. For j from $l - 1$ down to 0 do:
   - 3.1 $A \leftarrow 2A$.
   - 3.2 For i from 1 to d do:
     - 3.2.1 if j is the smallest integer for which $|(b_{i,t_i} ... b_{i,j})_2| < 2m$ and $b_{i,j} \neq 0$ then
       - 3.2.1.1 $h = (b_{i,t_i} ... b_{i,j})_2$,
       - 3.2.1.2 If ($h > 0$) then $A \leftarrow A + hP_i$,
       - 3.2.1.3 Else do: $A \leftarrow A - (-h)P_i$
       - 3.2.1.4 $t_i \leftarrow j - 1$.
4. Return (A).

Similar to single point multiplication, we can combine this extended algorithm with the SSBR representation to obtain an efficient method for multiple point multiplication. In fact, this method has the same performance as Möller's method [33]. The advantage of our proposed method over Möller's method is that the proposed method uses a simpler integer representation (SSBR). To compute d point multiplications simultaneously, both Möller's method and the proposed method store $d \times m$ points in their precomputation stages. However, for the proposed algorithm, m can be any positive integer, while for Möller's method m should be of the form 2^w. Therefore, another advantage of the proposed method is that it may use the available memory more efficiently.

3.4.3.3 Fixed Point Multiplication

For elliptic curve signature generation, we are required to compute kP for a fixed point P. Since P is fixed, we can significantly speed up the computation of kP by precomputing and storing a table of multiples of P. For example, by precomputing the points $2P, 2^2P, \ldots, 2^{l-1}P$, we can eliminate all the doubling operations required in the binary method. In this case, the binary method requires only $l/2$ point additions (on average). In fact, this method is the simplest example of the BGMW method proposed in [37]. In general, the BGMW method uses the 2^w-ary representation of k as follows. Let $k = (k_{d-1}, \ldots, k_1, k_0)_{2^w}$ where $d = \lceil l/w \rceil$, and $Q_j = \sum_{i:k_i=j} 2^{wi}P$ then

$$kP = \sum_{i=0}^{d-1} k_i(2^{wi}P) = \sum_{j=1}^{2^w-1} (j \sum_{i:k_i=j} 2^{wi}P) = \sum_{j=1}^{2^w-1} jQ_j$$

The BGMW method uses the key point that $\sum_{j=1}^{2^w-1} jQ_j$ can be computed efficiently as

$$\sum_{j=1}^{2^w-1} jQ_j = Q_{2^w-1} + (Q_{2^w-1} + Q_{2^w-2}) + \ldots + (Q_{2^w-1} + Q_{2^w-2} + \ldots + Q_1)$$

As we can see, this method does not need any point doublings. However, it requires $(((d(2^w-1)/2^w)-1)+(2^w-2))$ point additions on average. It also needs d points to be stored in the precomputation stage. Notice that for fixed point multiplication, the table of multiples of P is computed once in the precomputation stage and is used every time we want to compute kP for a random value of k.

Algorithm 22. The BGMW method

Input: Window width w, $d=\lceil l/w \rceil$, $k=(k_{d-1},...,k_1,k_0)_{2^w}$, $P \in E(\mathbb{F}_p)$.

Output: kP.

1. In the precomputation stage: Compute $P_i = 2^{wi}P$, $0 \le i \le d-1$.
2. $A \leftarrow o$, $B \leftarrow o$.
3. For j from $2^w - 1$ down to 1 do

   3.1 For each i for which $k_i = j$ do $B \leftarrow B + P_i$. {Add Q_j to B}

   3.2 $A \leftarrow A + B$.
4. Return (A).

The disadvantage of this method is that it uses a lot of memory because it works best when 2^w is small compared to l, but small values of 2^w requires more storage.

The second approach to fixed point multiplication is to use the Lim-Lee (LL) method [38]. The LL method first divides the multiplier k of l bits into hv subblocks of b bits as shown in Figure 3.3, where

$$k=\sum_{u=0}^{l-1} 2^u e_u = \sum_{i=0}^{h-1}\left(\sum_{j=0}^{v-1} k_{i,j} 2^{bj}\right)2^{ia}$$

$$a=\lceil l/h \rceil,\ b=\lceil a/v \rceil, \text{ and } k_{i,j}=\sum_{t=0}^{b-1} 2^t e_{ia+jb+t}.$$

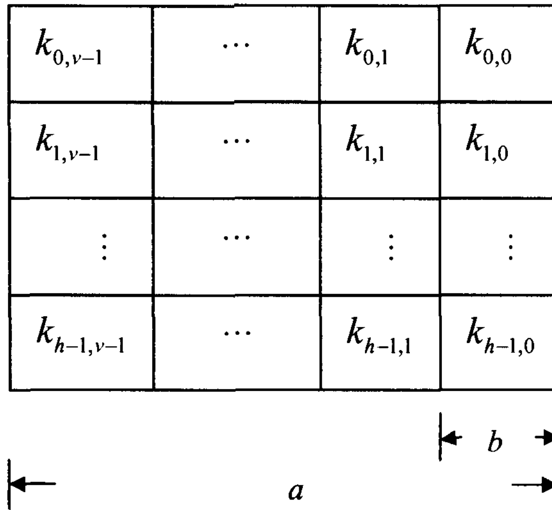

Figure 3.3. Partition of the multiplier k

The LL method is given in Algorithm 23. Fixed point multiplication using the LL method requires $(b-1)$ point doublings and on average $(a-1-\frac{a+(ah-l)}{2^h})$ point additions. This method stores $(2^h-1)v$ points in its precomputation stage.

Algorithm 23. LL method

Input: Multiplier $k = \sum_{i=0}^{l-1} 2^i e_i$ and block sizes h and v.

Output: kP.

1. In the precomputation stage: Compute $PP[I][j] = \sum_{i=0}^{h-1} 2^{ia+jb} b_i P$ for $0 \le j < v$ and $0 < I < 2^h$, where $I = \sum_{i=0}^{h-1} 2^i b_i$, $b_i \in \{0,1\}$.
2. $a \leftarrow \lceil l/h \rceil$, $b \leftarrow \lceil a/v \rceil$.
3. $Q \leftarrow o$.
4. For t from $b-1$ down to 0 do
   - 4.1 $Q \leftarrow 2Q$.
   - 4.2 For j from $v-1$ down to 0 do
     - 4.2.1 $I \leftarrow 0$.
     - 4.2.2 For i from $h-1$ down to 0 do
       - 4.2.2.1 $I \leftarrow 2I + e_{ia+jb+t}$.
     - 4.2.3 $Q \leftarrow Q + PP[I][j]$.
5. Return (Q).

3.5 Hash Function

A hash function H takes a variable size message M as input and produces a hash or message digest h as output (i.e. $h = H(M)$). The hash function with this definition has various applications. A simple example of such a function is used to generate the parity bit appended to 7-bit numbers (e.g., ASCII characters) such that the total number of 1s is always even (even parity) or odd (odd parity).

In cryptography, hash functions typically have certain additional properties.

1. The input can be of any length.
2. The output has a fixed length.
3. $H(M)$ is relatively easy to compute for any given M.
4. $H(M)$ is a one-way function: given h, it should be computationally infeasible to find an M such that $h = H(M)$.
5. $H(M)$ is collision free: given a message M_1, it is computationally infeasible to find a message $M_2 \neq M_1$ such that $H(M_1) = H(M_2)$.

Note that Properties 4 and 5 ensure security. SHA-1 (Secure Hash Algorithm), published by the National Institute of Standards and Technology (NIST), is a good example of a secure hash function. NIST has also published three additional variations of SHA, each with longer digests. They are named after the digest lengths (in bits), SHA-256, SHA-384, and SHA-512. Table 3.10 summarizes the SHAs variations.

Algorithm	**Message Size (bits)**	**Security (bits)**
SHA-1	$< 2^{64}$	80
SHA-256	$< 2^{64}$	128
SHA-384	$< 2^{128}$	192
SHA-512	$< 2^{128}$	256

Table 3.10. SHA properties

Figure 3.4 shows how hash functions are used to generate or verify a signature. As shown in the figure, the digital signature algorithm takes a message digest instead of the message as the input. This is because the message digest is small compared to the message itself. Furthermore, a message digest can be made public since it does not reveal the contents of the message from which it is derived.

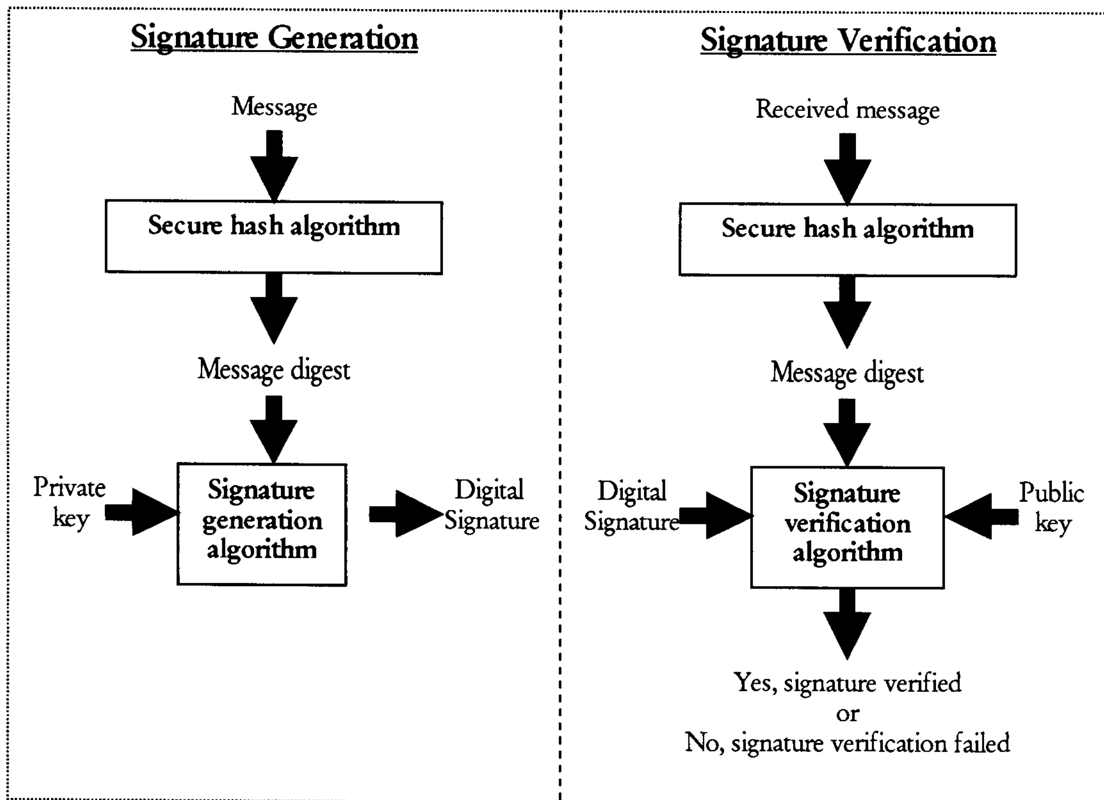

Figure 3.4. Using a hash function in signature generation and verification algorithms

3.6 Elliptic Curve Digital Signature Algorithm

The Elliptic Curve Digital Signature Algorithm (ECDSA) is a variant of the Digital Signature Algorithm [39] (DSA) which operates on elliptic curve groups. In order to sign a message, entity A needs to make public the system parameters such as:

- Prime p
- Elliptic curve $E_{a,b}$

- Base point P
- Order n of the base point P

Let the integer d, $d \in [1, n-1]$, be A's private key and the point $Q = dP$ its public key. In order to sign message M, A does the following.

1. Generate a random number $k \in [1, n-1]$.
2. Compute $R = kP$.
3. Compute $r = x \bmod n$ where x is the x-coordinate of R. If $r = 0$ then go to step 1.
4. Compute $e = H(M)$, where H is a cryptographic hash function such as SHA-1.
5. Compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ then go to step 1.
6. The signature on M is (r, s).

The entity B can verify A's signature (r, s) on M as follows:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute $e = H(M)$, where H is the same hash function used for generating the signature
3. Compute $w = s^{-1} \bmod n$, $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
4. Compute $X = u_1 P + u_2 Q$, and verify that $X \neq o$.
5. Let $v = x \bmod n$, where x is the x-coordinate of X.
6. Accept the signature if and only if $v = r$.

The requirement $v = r$ holds because if the signature (r, s) on a message M is indeed generated by A then

$$k = s^{-1}(e + dr) = s^{-1}e + s^{-1}rd = we + wrd = u_1 + u_2 d \pmod{n},$$

and thus

$$X = u_1 P + u_2 Q = (u_1 + u_2 d)P = kP \Rightarrow v = r.$$

Chapter 4

Implementation Results

This chapter contains the timing results on a Pentium II 448.81 MHz workstation running the Linux operating system for the operations required to implement ECDSA. All these operations have been written entirely in C in three levels as depicted in Figure 4.1. In Level 1, the finite field operations such as modular addition and modular multiplication are implemented. In Level 2, the operations from Level 1 are used to implement elliptic curve operations including point addition, point doubling and point multiplication. In Level 3, we employ the operations in Level 1 and Level 2 together with the SHA to implement digital signature generation and digital signature verification algorithms. Finally, we used GCC compiler to compile the C program.

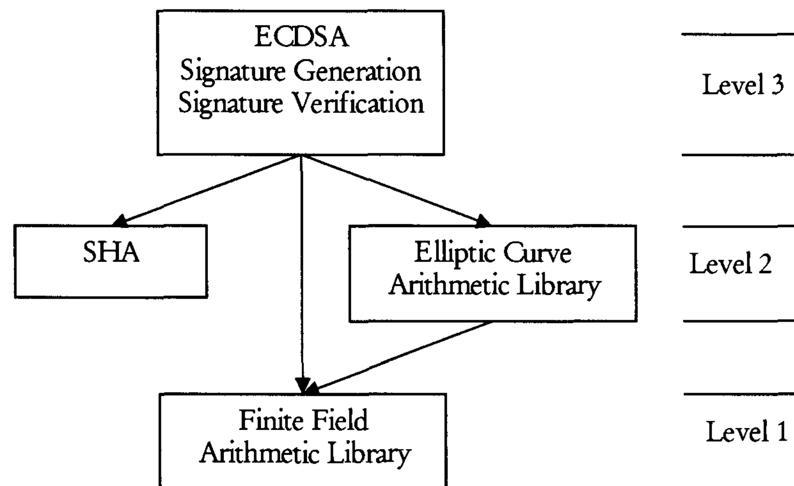

Figure 4.1. Implementation architecture

To obtain the timing results we used the RDTSC (Read-Time Stamp Counter) instruction provided by Pentium processors. This instruction allows the programmer to access a time-stamp

counter. The time-stamp counter is a 64-bit register which is incremented every clock cycle. Therefore, to measure the execution time of a function like $f(a)$, we can get the CPU clock cycle before and after invoking $f(a)$. Then, the execution time of $f(a)$ in μs can be computed by dividing the number of clock cycles by the CPU clock frequency in MHZ. Table 4.1 shows an example C code to obtain execution time of $f(a)$ on a Pentium II 400 MHZ. The "testFunc" function given in Table 4.1 returns the execution time of $f(a)$ for the input $a = 100$ in μs. Note that to get an average execution time of $f(a)$, we can use "testFunc" for several randomly chosen values of a.

```
flaot testFunc ( ){
        long x ,y ;
        int a;

        a = 100;
        __asm__ volatile ("rdtsc" : "=A" (y ));
        f(a);
        __asm__ volatile ("rdtsc" : "=A" (x ));

        return ((x - y) / 400);
}
```

Table 4.1. Measuring the execution time of $f(a = 100)$ in μs on a Pentium II 400 MHZ using the C language

4.1 Finite Field Arithmetic Timing

Two different functions have been implemented for most of the modular arithmetic operations. One function was optimized to minimize the code size and the other optimized to maximize the speed. To decrease the code size we used the following techniques:

- Reusability: Writing a frequently executed sequence of instructions as a function and invoking that function in the places where needed.

- Factoring: If an expression is carried out both when a condition is met and otherwise, it can be written just once outside of the condition statement.

To increase the speed we used these techniques

- Loop unrolling: Full or partial transformation of a loop into straight code. This technique attempts to reduce the overhead inherent in testing a loop condition.
- Function inlining: Putting the body of a function inside the code rather than invoking it from another location, thus eliminating the overhead of a function call.
- Condition eliminating: Replacing conditional instructions with unconditional instructions, thus allowing prefetching of instructions.

Note that although the word "optimization" shares the same root as "optimal", it is difficult, if not impossible, to find a truly optimal system.

4.1.1 Modular Addition and Subtraction Timing

Table 4.2 compares the timing results of two different function implementations of modular addition (Algorithm 1) and modular subtraction (Algorithm 2). As shown in Table 4.2, Function 1 is about 45% faster than Function 2. However, as mentioned in the previous chapter, Function 1 has a larger code size compared to Function 2.

Modular Addition and Subtraction		$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Addition (Algorithm 1)	Function 1	0.35	0.50	0.58	0.67	0.97
	Function 2	0.51	0.75	0.85	0.97	1.41
Subtraction (Algorithm 2)	Function 1	0.32	0.46	0.56	0.63	0.93
	Function 2	0.46	0.70	0.82	0.93	1.38

Table 4.2. Timing (in μs) for modular addition and subtraction including reduction (without compiler optimization)

It is also possible to do more optimization by enabling compiler optimization options. Without using these options, the compiler's goal is to reduce the cost of compilation and to allow

debugging. Since these goals are not important in our case we chose the "Os" option from the available compiler optimization options to both decrease the code size and timing. By enabling this option, GCC performs nearly all supported optimizations that do not typically increase code size. It is also possible to get faster execution using other optimization options such as "O3". However, these options may result in increased code size. Unlike Table 4.2, Table 4.3 presents the timing for the modular addition and subtraction functions optimized using the compiler "Os" option. Table 4.3 shows 60-110% timing improvement using the compiler "Os" option. Note that this compiler option was used to obtain the timing results in the remaining of this chapter.

Modular Addition and Subtraction		$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Addition (Algorithm 1)	Function 1	0.22	0.30	0.34	0.37	0.50
	Function 2	0.32	0.42	0.49	0.54	0.77
Subtraction (Algorithm 2)	Function 1	0.19	0.26	0.29	0.32	0.44
	Function 2	0.27	0.37	0.44	0.50	0.72

Table 4.3. Timing (in μs) for modular addition and subtraction including reduction (with compiler optimization)

4.1.2 Modular Reduction Timing

Table 4.4 compares Barrett reduction with the fast reduction algorithms. As we can see from this table, fast reduction algorithms are much faster than the Barrett reduction algorithm, particularly for large finite field sizes. However, we need Barrett reduction for reduction modulo the base point order in the Elliptic Curve Digital Signature Algorithm (ECDSA).

Modular Reduction	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Barrett Reduction (Algorithm 12)	14.35	25.63	31.63	40.7	76.39
Fast Reduction (Algorithms 6-10)	4.24	1.27	1.56	5.66	6.06

Table 4.4. Timing (in μs) for modular reduction

4.1.3 Modular Multiplication and Squaring Timing

Before implementing modular multiplication and squaring, we need to implement a word multiplication function. This simple function takes two words as input and returns two words (the product of the two input words), as output. Although this is a very simple function to implement, we will show later that it has a direct effect on the overall system performance. Table 4.5 shows the timing for two implementations of this operation. To optimize Function 1 and Function 2, we used the same techniques mentioned in Section 4.1. As shown in Table 4.5, Function 1, which was optimized for speed, is 2.3 times faster than Function 2, which was optimized for code size.

Word Multiplication (Algorithm 3)	**Timing**
Function 1	0.10
Function 2	0.23

Table 4.5. Timing (in μs) for word multiplication functions

The Barrett reduction timing in Table 4.4 was obtained using the slow word multiplication function (Function 2 in Table 4.5). Table 4.6 shows the Barrett reduction timing using the fast word multiplication function (Function 1 in Table 4.5). By comparing the numbers in Table 4.6 with the numbers in Table 4.4, we can see that the new Barrett reduction function is about 1.5 times faster than the previous implementation due to the use of the fast word multiplication function.

Barrett Reduction (Algorithm 12)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
	9.85	17.33	21.23	27.96	50.63

Table 4.6. Timing (in μs) for Barrett reduction function using the fast word multiplication function

Table 4.7 compares the timing of the classical multiplication algorithm with that of the Karatsuba multiplication algorithm. We have implemented Karatsuba algorithm only for $GF(p_{128})$ and

$GF(p_{256})$ since this algorithm gives the best performance when the number of words used to represent a field element is a power of 2. As we can see from Table 4.7, the Karatsuba algorithm is faster than the classical algorithm when the slow word multiplication function is used. This is due to the fact that the Karatsuba algorithm needs a smaller number of word multiplication operations (which is a slow operation in this case), compared to the classical algorithm.

Modular Multiplication	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Classical (Algorithm 4)	11.32	15.82	21.37	31.41	63.58
Karatsuba	9.25	—	—	22.11	—

Table 4.7. Timing (in μs) for classical and Karatsuba multiplication (including fast reduction) using the slow word multiplication function (Function 2)

Table 4.8 shows a 30-44% speed-up in the classical multiplication using the fast word multiplication function. On the other hand, when the fast word multiplication function is used, the Karatsuba function is slower than its previous implementation (for which the timing results are given in Table 4.7). From Tables 4.7 and 4.8 we can conclude that the Karatsuba algorithm may be preferable if the word multiplication operation is very slow (compared to the word addition operation).

As mentioned before, modular squaring is a special case of modular multiplication and can be performed more efficiently since both operands are equal. For example, classical modular squaring requires about 50% fewer word multiplication operations than classical modular multiplication. As expected, Table 4.9 shows that classical modular squaring is about 16-38% faster than classical modular multiplication.

Modular Multiplication	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Classical (Algorithm 4)	8.71	11.07	14.84	23.09	44.86
Karatsuba	10.50	—	—	25.67	—

Table 4.8. Timing (in μs) for classical and Karatsuba multiplication (including fast reduction) using the fast word multiplication function (Function 1)

Classical squaring (Algorithm 5)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Using the Slow Word Multiplication Function	9.18	11.46	15.61	23.56	45.95
Using the Fast Word Multiplication Function	7.52	8.57	11.76	18.65	35.30

Table 4.9. Timing (in μs) for classical squaring (including fast reduction)

4.1.4 Modular Inversion Timing

Before implementing the binary inversion algorithm, we need to implement a right shift function. Table 4.10 shows the timing results for two different implementations of the right shift algorithm. these two implementations are optimized for code size and speed using the optimization techniques mentioned in Section 4.1. The first row and the second row of Table 4.11 show the timing results of modular inversion implementation using the slow shift function (Function 2 in Table 4.10), and the fast shift function (Function 1 in Table 4.10), respectively. As seen in Table 4.11, a 13-25% speed-up can be achieved by using the fast shift function.

Right Shift Functions	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Function 1	0.078	0.094	0.087	0.107	0.129
Function 2	0.11	0.16	0.19	0.21	0.28

Table 4.10. Timing (in μs) for two implementations of right shift function

Modular Inversion (Algorithm 13)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Using the Slow Shift Function	115.8	231.5	329.7	396.9	862.4
Using the Fast Shift Function	102.6	205.1	259.0	350.7	690.6

Table 4.11. Timing (in μs) for inversion functions

4.2 Random Point Multiplication Timing

Point multiplication is the most crucial operation in an elliptic curve cryptosystem. Table 4.12 compares the binary point multiplication method with the NAF point multiplication method. As Table 4.12 shows, the NAF method is about 14% faster than the binary method. This is because the NAF method requires a smaller number of point additions and/or subtractions compared to the binary method.

Point Multiplication	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Binary Method (Algorithm 14)	17.53	36.86	58.25	99.90	295.10
NAF Method (Algorithms 15, 17)	15.51	32.41	51.13	87.53	257.21

Table 4.12. Timing (in ms) for the binary and NAF point multiplication methods using the slow word multiplication function

To obtain the results in Table 4.12, we used the slow word multiplication function. However, using the fast word multiplication function, we can achieve a significant improvement as shown in Table 4.13. From these results, we can see that the word multiplication function is one of the most crucial operations. Therefore, it is very important to make this function as fast as possible (perhaps by rewriting it using assembly language).

Point Multiplication	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Binary Method (Algorithm 14)	14.75	27.64	43.13	75.77	216.19
NAF Method (Algorithms 15, 17)	13.04	24.36	37.88	66.54	188.75

Table 4.13. Timing (in ms) for binary and NAF point multiplication methods using the fast word multiplication function

Another approach to speed up the random point multiplication operation is to use the proposed generalized sliding window method (Algorithm 18). In the pre-computation stage of this method we can store m points. Table 4.14 shows the timing results of the proposed generalized sliding window method for $m \in \{2,4,6,8,10\}$. Figure 4.2 compares the results of the general sliding window method with that of the binary method for $m \in \{2,4,6,8,10\}$. The y-axis represents the speed improvement percentage of the proposed method over the binary method. As shown in Figure 4.2, for each finite field, there is a maximum speed improvement achievable using the proposed generalized sliding window method. However, the maximum improvement occurs for different values of m . For example, for the case of $GF(p_{128})$ we get the best result with $m = 4$.

#Points Need to be Stored	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
2	12.37	23.10	35.89	62.41	176.64
4	12.03	22.42	34.77	60.24	169.20
6	12.06	22.50	34.73	59.98	167.73
8	12.17	22.68	34.97	59.98	167.07
10	12.36	23.06	35.40	60.47	167.68

Table 4.14. Timing (in ms) for the proposed generalized sliding window method (Algorithm 18)

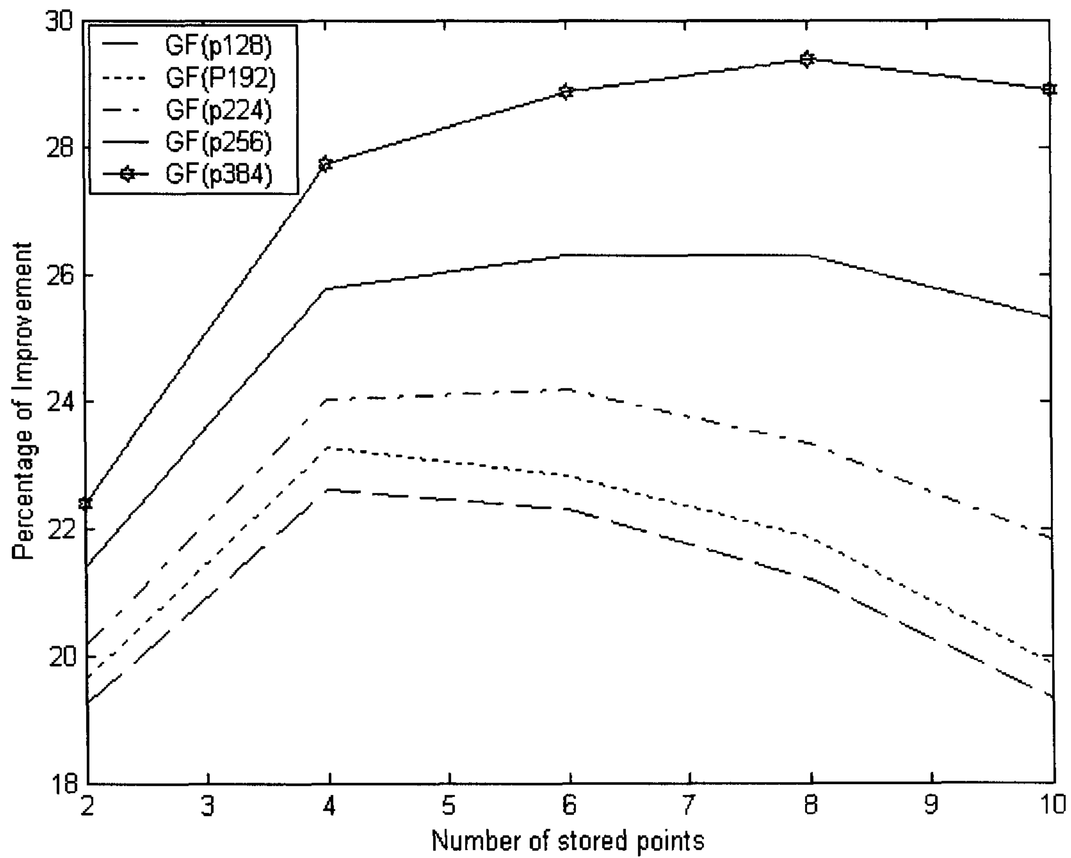

Figure 4.2. The percentage speed improvement of the proposed generalized sliding window method over the binary method

4.3 Fixed Point Multiplication

As mentioned before, in the digital signature generation algorithm, the base point can be considered as a fixed point. In this case, the point multiplication can be substantially sped up using a precomputed table. Tables 4.15 and 4.16 show the timing results for the LL algorithm (Algorithm 23) for different b sizes. For the results in both of these tables, the parameter v is equal to 1. The tables also indicate the number of points required to be stored in the pre-computed table. As we can see from Table 4.15, we need to store 255 and 4095 points for implementing fixed point multiplication over $GF(p_{256})$ and $GF(p_{384})$, respectively. However, by setting b equal to 64 (Table 4.16), we only need to store 15 and 63 points, respectively. In other words, to make fixed point multiplication two times faster, the number of points required to be store will be squared.

Fixed Point Multiplication (Algorithm 23)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Timing	5.05	6.41	8.73	13.66	25.98
#points required to be stored	15	63	127	255	4095

Table 4.15. Timing (in ms) for fixed point multiplication ($v=1$, $b=32$)

Fixed Point Multiplication (Algorithm 23)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Timing	8.96	11.94	25.25	26.31	51.63
#points required to be stored	3	7	15	15	63

Table 4.16. Timing (in ms) for fixed point multiplication ($v=1$, $b=64$)

4.4 Double Point Multiplication Timing

Double point multiplication is the most time consuming operation in digital signature verification. In the previous chapter we described some algorithms to perform double point multiplication (and multiple point multiplication in general). Table 4.17 shows the timing results for Algorithm 19. We chose this algorithm because it is simpler to implement compared to the other double point multiplication algorithms.

Double Point Multiplication Method (Algorithm 19)	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
	20.89	39.23	61.55	109.55	307.57

Table 4.17. Timing (in ms) for double point multiplication (Algorithm 19)

4.5 ECDSA Timing

The Elliptic Curve Digital Signature Algorithm (ECDSA) consists of two parts: digital signature generation and digital signature verification. To compute point multiplication in digital signature generation and digital signature verification we used the LL fixed multiplication method (Algorithm 23) and the simple double point multiplication method (Algorithm 19), respectively. In the LL method, the parameter b was set to 64 for $GF(p_{256})$ and $GF(p_{384})$, and was set to 32 for $GF(p_{128})$, $GF(p_{192})$ and $GF(p_{224})$. Note that we used the fast modular addition and subtraction function (Function 1 in Table 4.2), the fast word multiplication function (Function 1 in Table 4.5) and the fast right shift function (Function 1 in Table 4.10) in order to get the timing results given in Table 4.18. As expected, Table 4.18 shows that elliptic curve digital signature verification is significantly slower than digital signature generations. This is because digital signature verification uses double point multiplication which is significantly slower that fixed point multiplication (which is used by digital signature generation).

ECDSA	$GF(p_{128})$	$GF(p_{192})$	$GF(p_{224})$	$GF(p_{256})$	$GF(p_{384})$
Digital signature generation	5.67	7.48	10.03	27.50	54.24
Digital signature verification	19.37	35.73	56.16	98.90	276.26

Table 4.18. Timing (in ms) for digital signature generation and verification

4.6 Summary

One of the most challenging issues in implementing digital signature algorithm is the trade-off between memory and speed. For example, most of the speed optimization techniques such as loop unrolling and function inlining typically increase code size. Therefore, these techniques should be used carefully in restricted environments where memory and computational power are limited. One approach in this case can be to use the speed optimization techniques for a set of crucial functions including word multiplication, modular multiplication and point multiplication.

Another challenging issue is to choose a suitable algorithm. As shown earlier, fast modular reduction is much faster than Barrett modular reduction. The choice of a fast modular multiplication algorithm depends on the performance of the word multiplication function. In fact, when the word multiplication function is very slow (compared to the word addition function), and when the finite field size in bits is a power of the word size (power of 32 in our case), the Karatsuba multiplication algorithm is faster than the classical multiplication algorithm.

Chapter 5

Conclusion and Suggestions for Future Work

5.1 Conclusion

This thesis studied methods to improve random and multiple point multiplication operations in order to speed up Elliptic Curve Cryptography (ECC). It also presented a software implementation of the NIST-recommended elliptic curves over prime fields. Although the C program was compiled for a 32-bit architecture workstation, the program can also be used for a 16-bit architecture. This can be done by setting the program parameters and modifying a few functions.

The main contributions of this thesis are as follows:

- In Chapter 3, various integer representations including a simple signed binary representation were presented. We then showed how a modified Sliding Window Algorithm (SWA) can use these integer representations to generate point multiplication methods. We also modified the Katti representation to make it more suitable for use with the SWA. We then introduced new simple methods to compute single and multiple point multiplications. The proposed methods are computation and memory efficient, and so are suited to memory and power constrained devices.
- In Chapter 4, we presented timing results for the algorithms required to implement Elliptic Curve Digital Signature Algorithm (ECDSA). We investigated the trade-off between speed and memory using some optimization techniques as well as compiler optimization. We also compared the timing results of the proposed random point

multiplication method with that of binary method and showed that we can get 24-27% improvement by storing four points (including the base point) at the precomputation stage (Further improvements can be obtained for the finite field $GF(p_{384})$ by storing up to 8 points at the precomputation stage). It was also shown that word multiplication efficiency has a direct effect on point multiplication performance.

5.2 Suggestions for Future Work

There are several areas of future work that can be considered.

1. Software or hardware implementation of the proposed point multiplication methods on a microprocessor or DSP.
2. Employing the proposed method for implementing Hyperelliptic Curve Cryptography (HECC).
3. An extensive study of NIST-recommended curves implementation on constrained devices such as smart cards.

Bibliography

[1] W. Diffie and M.E. Hellman, "New directions in cryptography", *IEEE Transactions on Information Theory*, vol. 22, pp. 644-654, 1976.

[2] N. Koblitz, "Elliptic curve cryptosystems", *Mathematics of Computation*, vol. 48, pp. 203-209, 1987.

[3] V. Miller, "Uses of elliptic curves in cryptography", *Springer-Verlag Lecture Notes in Computer Science*, vol. 218, pp. 417-426, 1987.

[4] A. Menezes, Y. Wu, R. Zuccherato, "An Elementary Introduction to Hyperelliptic Curves", appendix in *algebraic aspects of cryptography* by N. Koblitz, *Springer-Verlag,* pp. 155-178, 1998.

[5] D. V. Chudnovsky and G.V. Chudnovsky, "Sequences of numbers generated by addition in formal groups and new primality and factorization tests", *Advances in Applied Mathematics*, vol. 7, pp. 385-434, 1987.

[6] H. Cohen, A. Miyaji, and T. Ono., "Efficient elliptic curve exponentiation using mixed coordinates", Advances in Cryptology, *Springer-Verlag Lecture Notes in Computer Science,* vol. 1514, pp. 51-65, 1998.

[7] I.F. Blake, G. Seroussi, N.P. Smart, "Elliptic curves in cryptography", *Cambridge University Press*, 1999.

[8] Certicom ECC Challenge, http://www.certicom.com.

[9] Certicom Announces Elliptic Curve Cryptosystem (ECC) Challenge Winner, http://www.certicom.com/index.php?action=company,press_archive&view=121

[10] P. van Oorschot and M.Wiener, "Parallel collision search with cryptanalytic applications", *Journal of Cryptology*, vol. 12, pp. 1-28, 1999.

[11] R. Silverman and J. Stapleton, Contribution to ANSI X9F1 Working Group, 1997.

[12] A. Menezes, T. Okamoto and S. Vanstone, "Reducing elliptic curve logarithms to logarithms in a finite field", *IEEE Transactions on Information Theory*, vol. 39, pp. 1639-1646, 1993.

[13] R. Balasubramanian and N. Koblitz, "The improbability that an elliptic curve has subexponential discrete log problem under the Menezes-Okamoto-Vanstone algorithm", *Journal of Cryptology*, vol. 11, pp. 114-145, 1998.

[14] I. Semaev, "Evaluation of discrete logarithms in a group of p-torsion points of an elliptic curve in characteristic p", *Mathematics of Computation*, vol. 67, pp. 353-356, 1998.

[15] N. Smart, "The discrete logarithm problem on elliptic curves of trace one", *Journal of Cryptology*, vol. 12, pp. 193-196, 1999.

[16] T. Satoh and K. Araki, "Fermat quotients and the polynomial time discrete log algorithm for anomalous elliptic curves", *Commentarii Mathematici Universitatis Sancti Pauli,* vol. 47, pp. 81-92, 1998.

[17] R. Gallant, R. Lambert and S. Vanstone, "Improving the parallelized Pollard lambda search on anomalous binary curves", *Mathematics of Computation*, vol. 69, pp. 1699-1705, 2000.

[18] M. Wiener and R. Zuccherato, "Faster attacks on elliptic curve cryptosystems", Selected Areas in Cryptography, *Springer-Verlag Lecture Notes in Computer Science,* vol. 1556, pp. 190-200, 1999.

[19] D.V. Bailey and C. Paar, "Efficient arithmetic in finite field extensions with application in elliptic curve cryptography", *Journal of Cryptology*, vol. 14, pp. 153-176, 2001.

[20] A. Solinas, "Generalized Mersenne numbers", Technical Report, *Center of Applied Cryptographic Research*, CORR. 99-39, University of Waterloo, 1999. Available from http://www.cacr.math.uwaterloo.ca

[21] D.V. Bailey and C. Paar, "Optimal extension fields for fast arithmetic in public-key algorithms", *Springer-Verlag Lecture Notes in Computer Science*, vol. 1462, pp. 472-485, 1998.

[22] N. Smart, "A comparison of different finite fields for use in elliptic curve cryptosystems", Research Report CSTR-00-007, *University of Bristol*, 2000.

[23] P. Ning and Y.L Yin, "Efficient software implementation for finite field multiplication in normal basis", *Springer-Verlag Lecture Notes in Computer Science*, vol. 2229, pp. 177-188, 2001.

[24] N. Koblitz, "CM-curves with good cryptographic properties", *Springer-Verlag Lecture Notes in Computer Science*, vol. 576, pp. 279-287, 1992.

[25] R. Gallant, R. Lambert, and S. Vanstone, "Faster point multiplication on elliptic curves with efficient endomorphisms", *Springer-Verlag Lecture Notes in Computer Science*, vol. 2139, pp. 190-200, 2001.

[26] K. Okeya, H. Kurumatani, and K. Sakurai, "Elliptic curves with the Montgomery-form and their cryptographic applications", Public Key Cryptography, *Springer-Verlag Lecture Notes in Computer Science,* vol. 1751, pp. 238-257, 2000.

[27] M. Brown, D. Hankerson, J. Hernandez, and A. Menezes, "Software implementation of the NIST elliptic curves over prime fields", Topics in Cryptology, *Springer-Verlag Lecture Notes in Computer Science,* vol. 2020, pp. 250-265, 2001.

[28] D. Knuth, "The art of computer programming-seminumerical algorithms", *Addison-Wesley*, 1998.

[29] A. Schönhage and V. Strassen, "Schnelle multiplication *großer*", *Computing*, vol. 7, pp. 281-292, 1971.

[30] P.Y. Liardet , and N. Smart, "Preventing SPA/DPA in ECC systems using the Jacobi form", Cryptographic Hardware and Embedded Systems, *Springer-Verlag Lecture Notes in Computer Science,* vol. 2162, pp. 391-401, 2001.

[31] M. Joye and J. Quisquater, "Hessian elliptic curves and side-channel attacks", Cryptographic Hardware and Embedded Systems, *Springer-Verlag Lecture Notes in Computer Science*, vol. 2162, pp. 402-410, 2001.

[32] T. Hasegawa, J. Nakajima and M. Matsui, "A small and fast software implementation of elliptic curve cryptosystems over $GF(p)$ on a 16-bit microcomputer", *IEICE Trans. Fundamentals*, vol. E82-A, no.1, 1999.

[33] B. Möller, "Algorithms for multi-exponentiation", *Springer-Verlag Lecture Notes in Computer Science*, vol. 2259, pp. 165-180, 2001.

[34] K. Koyama and Y. Tsuruoka, "Speeding up elliptic cryptosystems by using a signed binary window method", *Springer-Verlag Lecture Notes in Computer Science*, vol. 740, 1993.

[35] R.S. Katti, "Speeding up elliptic cryptosystems using a new signed binary representation for integers", Proc. Euro-Micro Conf. Digital Design, 2002.

[36] R.M. Avanzi, "On multi-exponentiation in cryptography", Technical Report 2002/154, *Cryptology Print Archive*, available from http://eprint.iacr.org/2002/154, 2002.

[37] E. Brickell, D. Gordon, K. McCurley and D. Wilson, "Fast exponentiation with precomputation", *Springer-Verlag Lecture Notes in Computer Science*, vol. 658, pp. 200-207, 1993.

[38] C. Lim and P. Lee, "More flexible exponentiation with precomputation", *Springer-Verlag Lecture Notes in Computer Science*, vol. 839, pp. 95-107, 1994.

[39] FIPS 186-2, "Digital signature standard (DSS)", *National Institute of Standards and Technology*, 2000.