Efficient Algorithms for Elliptic Curve Cryptosystems

by
Jorge Guajardo

A Thesis
submitted to the Faculty
of the
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical Engineering
by

____

May, 1997

Approved:

____

Dr. Christof Paar
Thesis Advisor
ECE Department

____

Dr. David Cyganski
Thesis Committee
ECE Department

____

Dr. Robert Dulude
Thesis Committee
GTE Government Systems

____

Dr. Gabor Sarkozy
Thesis Committee
CS Department

____

Dr. John Orr
Department Head
ECE Department

Abstract

Elliptic curves are the basis for a relative new class of public-key schemes. It is predicted that elliptic curves will replace many existing schemes in the near future. It is thus of great interest to develop algorithms which allow efficient implementations of elliptic curve crypto systems. This thesis deals with such algorithms.

Efficient algorithms for elliptic curves can be classified into low-level algorithms, which deal with arithmetic in the underlying finite field and high-level algorithms, which operate with the group operation. This thesis describes three new algorithms for efficient implementations of elliptic curve cryptosystems. The first algorithm describes the application of the Karatsuba-Ofman Algorithm to multiplication in composite fields $GF((2^n)^m)$. The second algorithm deals with efficient inversion in composite Galois fields of the form $GF((2^n)^m)$. The third algorithm is an entirely new approach which accelerates the multiplication of points which is the core operation in elliptic curve public-key systems. The algorithm explores computational advantages by computing repeated point doublings directly through closed formulae rather than from individual point doublings. Finally we apply all three algorithms to an implementation of an elliptic curve system over $GF((2^{16})^{11})$. We provide absolute performance measures for the field operations and for an entire point multiplication. We also show the improvements gained by the new point multiplication algorithm in conjunction with the k-ary and improved k-ary methods for exponentiation.

Preface

This thesis describes the research that I conducted while completing my Master's degree at Worcester Polytechnic Institute. I have attempted to compile as much information related to elliptic curves as it is adequate and relevant to this thesis. I have also tried to be as explicit as possible in the derivation of formulae and theorems. Finally, I hope this work to be of use in both university and industry settings in which elliptic curve cryptosystems are being developed.

I would like to thank the following people for their support throughout the months that I worked on this thesis. First, I would like to thank Prof. Paar for all the help, advise, and support that he gave me throughout my graduate work at WPI. I would also like to thank Prof. Paar for the atmosphere of friendship and camaraderie that he developed while working with him.

I am grateful to Prof. David Cyganski and Prof. Gabor Sarkozy from Worcester Polytechnic Institute and Dr. Robert Dulude from GTE Government Systems for the valuable suggestions, comments and advices that they gave me as members of my Thesis committee.

Finally, I would like to give special thanks to several of my friends and family: Rafael Alaña, Moises Gomez Morante, and Enrique Martinez for always being there and provide me with support and encouragement to complete my thesis; Kathy Kreitner, my lovely girlfriend, for all the time that she spent helping me to prepare my thesis defense and the manuscript that you are about to read; my family for always being there to support me and encourage me; and finally, the guys from the lab Gre-

gory Haskins, Martin Rosner and Frank Schaefers for always being in a good mood and providing a great atmosphere to work. To all of you thank you very much.

Jorge Guajardo

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Thesis Outline . . . 3

2 Background **5**

2.1 Galois Fields $GF((2^n)^m)$ . . . 5

2.1.1 Polynomial Rings and Fields . . . 5

2.1.2 Composite Fields . . . 6

2.1.3 Choosing the Field Polynomial $P(x)$ . . . 7

2.2 Non-Supersingular Elliptic Curves over Fields of Characteristic Two . 8

2.2.1 Background on Elliptic Curves . . . 8

2.2.2 Elliptic Curves over Fields of Characteristic Two in Affine Coordinates . . . 10

2.2.3 Multiples of Points . . . 12

3 Computer Arithmetic in Composite Fields and Elliptic Curves **13**

3.1 Computer Representation of Galois Field Elements . . . 13

3.1.1 Composite Field Libraries . . . 14

3.1.2 Application to $GF((2^{16})^{11})$ . . . 15

3.2 Computer Arithmetic in Composite Fields . . . 15

3.2.1 Multiplication in $GF(2^n)$ . . . 16

3.2.2 Addition in $GF((2^n)^m)$ 17

3.2.3 Multiplication in $GF((2^n)^m)$ 17

3.2.4 Squaring in $GF((2^n)^m)$ 19

3.2.5 Inversion in $GF((2^n)^m)$ 19

3.2.6 Arithmetic in $GF((2^{16})^{11})$ 20

3.3 Computer Representation of Elliptic Curve Elements 21

4 Previous Work **22**

4.1 Elliptic Curve Implementations over $GF(2^k)$ 22

4.1.1 Early work 22

4.1.2 Hardware Implementation over $GF(2^{155})$ Using Normal Basis Representation 25

4.1.3 Software Implementation over $GF(2^{155})$ Using Standard Basis Representation 26

4.2 Elliptic Curve Implementations over Composite Fields 29

4.2.1 Implementation over $GF((2^8)^{13})$ 30

4.2.2 Implementations over $GF((2^{16})^{11})$ 31

4.3 Efficient Exponentiation 34

5 Fast Multiplication in Composite Galois Fields $GF((2^n)^m)$ **38**

5.1 The KOA for Polynomials of Degree $2^t - 1$ 39

5.2 Complexity of the KOA for Polynomials of Degree $2^t l - 1$ 40

5.3 Complexity of the KOA for Polynomials of Degree $2^t l - 2$ 42

5.4 Complexity Analysis for Software Implementations 45

5.5 Multiplication in $GF((2^{16})^{11})$ 48

6 Efficient Inversion in Composite Galois Fields $GF((2^n)^m)$ **50**

6.1 Exponentiation in $GF((2^n)^m)$ 52

6.2 Multiplication in $GF((2^n)^m)$, where the Product is an Element of $GF(2^n)$ 53

6.3 Inversion in $GF(2^n)$ and Multiplication of an Element from $GF(2^n)$ with an Element from $GF((2^n)^m)$ 54
6.4 Inversion in $GF((2^{16})^{11})$ 55

7 A New Approach to Point Doubling for Elliptic Curves 56
7.1 Principle Idea 57
7.1.1 The Break-Even Point 60
7.1.2 Theoretical and Practical Timings 62
7.2 Complexity Analysis of the k-ary Method 63
7.2.1 k-ary Method Complexity 64
7.2.2 Complexity of the k-ary Method with Formulae for $k = 4$ . . . 65
7.2.3 Relative Improvement 65
7.3 Complexity Analysis of the Improved k-ary Method with Formulae for $k = 5$ 67
7.4 Application to Point Multiplication 70

8 Elliptic Curve Key Exchange Protocols 73
8.1 Elliptic Curve Analog to Diffie-Hellman Key Exchange 73
8.2 Proposed IEEE Standard 75
8.3 Performance Analysis 77

9 Discussion 79
9.1 Conclusions 79
9.2 Recommendations for Further Research 80
9.2.1 Generalization of the Improved Point Multiplication 81
9.2.2 Implementation of a Variant of the Improved k-ary Method . . 81
9.2.3 Implementation of the Sliding Window Method for Exponentiation 81
9.2.4 Implementation of a Different Inversion Algorithm 82

A Proofs for Doubling Formulas **83**

A.1 Proof for Theorem 13 83

A.2 Proof for Theorem 14 84

A.3 Proof for Theorem 15 85

List of Tables

4.1 Timings for various field and elliptic curve operations [SOOS95]. . . . 29

4.2 Comparison of timings for elliptic curve and field operations using normal and polynomial basis representations (time in seconds)[HMV92]. 31

4.3 Comparison of timings for elliptic curve and field operations between elliptic curve implementations over $GF((2^{16})^{11})$. NOTE: unmarked times are in microseconds. 34

5.1 Comparison of timings for 176-bit multiplication in $GF((2^{16})^{11})$ using the Karatsuba-Ofman algorithm and the straight forward multiplication method. 49

7.1 Complexity comparison: Individual doublings vs. direct computation of several doublings. 61

7.2 Timings for various field operations in $GF((2^{16})^{11})$. 62

7.3 Timing comparison: Individual doublings vs. direct computation of several doublings in $GF((2^{16})^{11})$. 63

7.4 Comparison of complexities required to perform the multiplication eP using the regular k-ary method, $k = 4$, and the k-ary method with four direct doublings. 66

7.5 Frequency of occurrence for possible h_i and $5 - h_i$ values. 67

7.6 Complexity of Doubling Approach. 68

7.7 Number of Operation for Bit Patterns of the Most Significant Word of the Multiplier. 69

7.8 Comparison of average time required to perform the nP calculation in $GF((2^{16})^{11})$ using the regular k-ary method and the k-ary method with four direct doublings using a DEC Alpha with 175 MHz clock frequency. 71

7.9 Timings for various field and elliptic curve operations. 71

8.1 Time comparison for key-exchange algorithms: Modulo arithmetic vs. elliptic curves over the field $GF((2^{16})^{11})$. 77

Chapter 1

Introduction

1.1 Motivation

In 1976, Diffie and Hellman revolutionized the field of cryptography with the invention of public-key cryptography [DH76]. Soon after, RSA, the first usable public key cryptosystem, was introduced [RSA78]. This particular cryptosystem is based on the difficulty of factoring very large numbers and today, it is still the most widely used public-key cryptosystem in the world. Since then, in the field of computational number theory, major work has been done towards efficient integer factorization. As a consequence, new types of public-key algorithms have arisen. The most important competitors to RSA are schemes based on the Discrete Logarithm (DL) problem. Originally, the DL problem was considered in the multiplicative group of a finite field, especially a prime field or a field of characteristic 2, since these fields seemed most appropriate for implementations. Then in 1985, a variant of the DL problem was proposed by Miller [Mil86] and Koblitz [Kob87], based on the group of points of an elliptic curve (EC) over a finite field.

A main feature that makes elliptic curves attractive is the relatively short operand length. Cryptosystems which explore the DL problem over elliptic curves can be built with an operand length of 140–200 bits [Men93b] as compared to RSA and systems based on the DL in finite fields both of which require operands of 512–1024 bits. Other advantages are the large numbers of curves available to provide the groups and the absence of sub-exponential time algorithms (such us the index calculus method) to attack cryptosystems in these groups. The latter property provides a very good long-term security against current attacks. In addition, IEEE [KMQV96] and other standard bodies such as ANSI and ISO are in the process of standardizing elliptic curve cryptosystems. It is important to point out that elliptic curves can provide various security services such as key exchange, privacy through encryption,and sender authentication and message integrity through digital signatures. For these reasons it is expected that elliptic curves will become very popular for many information security applications in the near future. It is thus very attractive to provide algorithms which allow for efficient implementations of elliptic curve cryptosystems. This thesis will deal with such algorithms.

Efficient algorithms for elliptic curves can be classified into high-level algorithms, which operate with the group operation, and into low-level algorithms, which deal with arithmetic in the underlying finite field. For efficient implementations it is obviously the best to optimize both types of algorithms. The main part of the thesis will introduce three algorithms, one high-level algorithm for point multiplication and two low-level for finite field inversion and multiplication, respectively.

1.2 Thesis Outline

Chapter 2 gives an elementary introduction to composite Galois Fields and elliptic curves. It covers some of the mathematical theory behind the construction of composite fields and the basic equations that will govern the addition and doubling of elliptic curve points. Finally, it introduces the basic method used to compute the multiple of a point and the notation that will be used throughout this thesis to refer to composite fields and elliptic curves.

Chapter 3 presents the approach that we used to implement in software addition, multiplication, and inversion over Galois Fields. We also introduce the concept of the Table Look-Up (TLU) which will be fundamental to the discussions about algorithm complexity in Chapter 5 and the basic data structure used to represent elliptic curve points.

Chapter 4 summarizes previous works on elliptic curve cryptosystems presented in the research community in the past. We will introduce some of the previous elliptic curve implementations found in the literature, both over $GF(2^k)$ and $GF((2^n)^m)$. We will also summarize the work in each of following areas: efficient inversion and efficient multiplication algorithms for arithmetic in $GF((2^n)^m)$ and elliptic curve point addition algorithms. Finally, we will present some of the improved algorithms used to compute the product of an elliptic curve point by a large integer.

Chapter 5 provides a detailed treatment of the Karatsuba-Ofman algorithm (KOA) applied to field multiplication in $GF((2^n)^m)$. We provide a complexity analysis of the KOA for software implementations where arithmetic in the subfield $GF(2^n)$ is based on table look-up.

Chapter 6 shows an algorithm for efficiently computing the inverse of an element in the composite Galois field $GF((2^n)^m) \cong GF(2^k)$. The algorithm is based

on an idea by Itoh and Tsujii [IT88], but is optimized for a standard base representation and for binary field polynomials. The algorithm reduces inversion in the composite field to inversion in the subfield $GF(2^n)$. Unlike the inversion algorithms in [SOOS95, WBV$^+$96], the inversion algorithm is based on Fermat's little theorem.

Chapter 7 introduces an entirely new approach for accelerating the multiplication of points on an elliptic curve. The approach works in conjunction with the k-ary and the sliding window methods. The method is applicable to elliptic curves over any field, but we provide worked-out formulae for elliptic curves over fields of characteristic two. In addition, we show the actual performance of the newly introduced algorithm and the ones treated in Chapters 5 and 6 in an implementation of an elliptic curve cryptosystem over $GF(2^{176}) \cong GF((2^{16})^{11})$. We provide absolute timing measurements for an entire elliptic curve multiplication as well as timings for individual operations.

Chapter 8 describes an elliptic curve algorithm analog to the Diffie-Hellman key exchange protocol [DH76], as well a systems analog to the ElGamal cryptosystem [ElG85]. In addition, a draft of the proposed IEEE Standard for elliptic curve cryptosystems will be introduced. Timing estimates for several of these systems will be provided.

Finally, Chapter 9 will discuss the results of this research. It will also provide the reader with recommendations for further research in the general area of algorithm optimization for elliptic curves.

Chapter 2

Background

This chapter introduces elliptic curves and composite Galois fields over $GF((2^n)^m)$ and the notation that will be used throughout this thesis to refer to them. We also present the formulae used to add elliptic curve points for curves over fields of characteristic two. Finally, we introduce an algorithm used to efficiently compute a multiple of an elliptic curve point.

2.1 Galois Fields $GF((2^n)^m)$

2.1.1 Polynomial Rings and Fields

It is known that the set of integers modulo q, where q is a prime, forms a field, where a field is as defined in [LC83]. This field is denoted as Z_q. One can also define $Z_q[x]$ to be the set of all polynomials with coefficients from Z_q in the indeterminate x. Then, one can construct the ring of polynomials modulo q by combining the set $Z_q[x]$ with the operations of addition and multiplication of polynomials (as defined in the usual way) and reducing the coefficients modulo q [LN83].

A second ring can be constructed in a similar manner. This time, we will construct the ring of polynomials modulo $f(x)$, where $f(x) \in Z_q[x]$ and $\deg(f(x)) = m \geq 1$. This ring is denoted by $Z_q[x]/f(x)$. The elements of the ring are all those polynomials in $Z_q[x]$ with degree less or equal to $m-1$. Addition and multiplication are defined as in the case of $Z_q[x]$ followed by a reduction modulo $f(x)$. It turns out that if $f(x)$ is irreducible, $GF(q^m) \cong Z_q[x]/f(x)$, is a finite field [LC83]. In addition, it has been shown that an irreducible polynomial of degree m over $GF(q)$ exist for any finite field $GF(q)$. In the rest of this thesis the notation $GF(q^m)$ will be used when referring to finite fields.

Notice that $GF(q^m)$ is also referred to as an "extension field of $GF(q)$" and it has order q^m. The field $GF(q)$ is then referred to as the "ground field" or "subfield" of $GF(q^m)$ [McE87]. Finally, every element in $GF(q^m)$ can be represented as a polynomial $A(x) = a_{m-1}x^{m-1} + \cdots + a_0$ with coefficients $a_i \in GF(q),\ i = 0, 1, \cdots, m-1$. $A(x)$ is said to be "a polynomial over $GF(q)$." These q^m polynomials in $GF(q^m)$ form the residue classes modulo $f(x)$ of all polynomials over $GF(q)$.

2.1.2 Composite Fields

In this section we will introduce a special type of finite fields, called composite fields, which will prove to be an essential concept in the development of this thesis.

In Section 2.1.1, it was assumed that q was a prime when constructing a finite field. In fact, q does not need to be a prime. The order of a field only needs to be a power of a prime or $q = p^m$ where p is prime. In particular, we can build further extensions on extension fields. For instance, the field $GF(q^n)$ can be extended to $GF((q^n)^m)$. Notice that in practical applications (hardware or software) where finite field arithmetic needs to be implemented, the choice of $q = 2$ is very beneficial because of the way in which information is represented inside computers. Thus, we define a

special type of finite fields.

Definition 1 *We call two pairs* $\{GF(2^n), Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i\}$ *and* $\{GF((2^n)^m), P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i\}$ *a* composite field *if*

- $GF(2^n)$ *is constructed as an extension field of* $GF(2)$ *by* $Q(y)$*,*
- $GF((2^n)^m)$ *is constructed as extension field of* $GF(2^n)$ *by* $P(x)$*.*

where $Q(y)$ *is an irreducible polynomial over* $GF(2)$ *and* $P(x)$ *is also irreducible over* $GF(2)$*. In the rest of this thesis composite fields will be denoted by* $GF((2^n)^m)$*.*

It is important to point out that from a mathematical point of view $GF((2^n)^m)$ is isomorphic to $GF(2^k)$ for $nm = k$ [LN83]. However, although a field of order 2^{nm} and one of order 2^k are isomorphic, their algorithmic complexity is different with respect to the field operations addition and multiplication and, in general, it will depend on the choice of m and n and more specifically on the polynomials $Q(y)$ and $P(x)$ [Paa94].

2.1.3 Choosing the Field Polynomial $P(x)$

As stated above, the polynomials $Q(y)$ and $P(x)$ are of great importance in determining the algorithmic complexity of arithmetic in the field $GF((2^n)^m)$. In this section we will explore a way to choose field polynomials that we will provide us with certain algorithmic advantages.

The polynomial $P(x)$ generates the field $GF((2^n)^m)$ and it is very important in reducing the complexity of the basic operations in the Galois field $GF((2^n)^m)$. [Jun93] showed that if $gcd(n, m) = 1$ then every $P(x)$ of degree m which is irreducible over

$GF(2)$, is also irreducible over $GF(2^n)$. Therefore, one can choose n and m to be relatively prime and then choose $P(x)$ in such a way that it only has binary coefficients as opposed to coefficients from $GF(2^n)$. Furthermore, one should try to choose an irreducible polynomial $P(x)$ with the least number of coefficients, thus minimizing the complexity of modular arithmetic in the Galois field.

2.2 Non-Supersingular Elliptic Curves over Fields of Characteristic Two

In this section, we define elliptic curves and give general equations that describe them. The section is meant to be a condensed summary that introduces the reader to the concept of the elliptic curve. For a more extensive treatment referred to [Men93b]. We then specialize to the case of elliptic curves over finite fields of characteristic 2. Finally, we define the group law for this curves and analyze the complexity of these equations.

2.2.1 Background on Elliptic Curves

Let K be a field with characteristic char(K). An elliptic curve over K, denoted by E, is the set of points (X, Y, Z), which satisfy a generalized smooth Weierstrass equation:

$$E: \; Y^2Z + a_1XYZ + a_3YZ^2 \; = \; X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{2.1}$$

where $a_1, ..., a_6 \in \overline{K}$ ($\overline{K}$ is a fixed algebraic closure of K) and smooth refers to the fact that for all points (X,Y,Z)$\in P^2(\overline{K})$ satisfying,

$$F(X,Y,Z) = Y^2Z + a_1XYZ + a_3YZ^2 - X^3 - a_2X^2Z - a_4XZ^2 - a_6Z^3 = 0$$

at least one of the three partial derivatives $\frac{\partial F}{\partial X}$, $\frac{\partial F}{\partial Y}$, and $\frac{\partial F}{\partial Z}$ is non-zero at P. There is exactly one point in E with Z-coordinate equal to 0, that is (0,1,0). We call this point the *point at infinity* and denote it by $\mathcal{O}$.

To obtain an equation in non-homogeneous coordinates, we can let $x = X/Z$ and $y = Y/Z$, and together with the special point $\mathcal{O}$, we get [Kaz92] :

$$E: \; y^2 + a_1xy + a_3y \;=\; x^3 + a_2x^2 + a_4x + a_6 \tag{2.2}$$

Thus, an elliptic curve E defined over a finite field K is the set of solutions $(x, y) \in K^2$ which satisfies (2.2) together with the point of infinity $\mathcal{O}$.

Elliptic curves can be simplified over fields of different characteristics by means of coordinate transformation. However, in the rest of this thesis, we will be only concerned with curves of characteristic 2. One can distinguish between two types of curves in this case:

- Singular elliptic curves with char(K)=2, where E is defined as

  $$y^2 + ay = x^3 + bx + c$$

  and,
- Non-supersingular elliptic curves with char(K)=2, where E is defined as

  $$y^2 + xy = x^3 + ax^2 + c.$$

It is important to point out that an elliptic curve is said to be supersingular if and only if the $j-invariant$ of E is not equal to zero, otherwise it is non-supersingular [Men93a].

As a result of the MOV reduction attack [MOV93], which only applies to supersingular elliptic curves, the security of systems based on the two classes of curves

differ radically. The MOV attack leads to sub-exponential attacks (index-calculus method) against supersingular elliptic curves, whereas the best known attacks against non-supersingular elliptic curves have an exponential complexity. It is important to point out that supersingular elliptic curves should be avoided when implementing cryptosystems based on elliptic curves. Moreover, the use of non-supersingular curves in public-key cryptography can provide far more security per bit ratio than existing systems such as RSA [AMV93], therefore they will be the only ones studied in this thesis.

2.2.2 Elliptic Curves over Fields of Characteristic Two in Affine Coordinates

In the remainder of this report we will study optimum ways to implement non-supersingular elliptic curves over $GF(2^k)$. Thus, we define an elliptic curve in this context as:

$$y^2 + xy = x^3 + ax^2 + c \tag{2.3}$$

where $a, c \in GF(2^k), \quad c \neq 0$, together with the point at infinity $\mathcal{O}$ [AMV93].

The addition operation for an elliptic curve E (2.3) is defined as follows. Let $P = (x_1, y_1) \in E$; then $-P = (x_1, y_1 + x_1)$. $P + \mathcal{O} = \mathcal{O} + P = P$ for all $P \in E$. If $Q = (x_2, y_2) \in E$ and $Q \neq -P$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \begin{cases} \lambda^2 + \lambda + x_1 + x_2 + a & P \neq Q \\ \lambda^2 + \lambda + a & P = Q \end{cases} \tag{2.4}$$

and

$$y_3 = \begin{cases} \lambda(x_1 + x_3) + x_3 + y_1 & P \neq Q \\ \\ x_1^2 + \lambda x_3 + x_3 & P = Q \end{cases} \tag{2.5}$$

where

$$\lambda = \begin{cases} \frac{y_1 + y_2}{x_1 + x_2} & P \neq Q \\ \\ x_1 + \frac{y_1}{x_1} & P = Q \end{cases} \tag{2.6}$$

From (2.4) and (2.5) the addition of two different elliptic curve points, i.e., $P \neq Q$, requires the following field operations:

- 8 Additions
- 1 Squaring
- 2 Multiplications
- 1 Inverse

On the other hand, the doubling of an elliptic curve point, i.e., $P = Q$, using (2.4) and (2.5) requires the following field operations:

- 5 Additions
- 2 Squarings
- 2 Multiplications
- 1 Inverse

From the discussion in Section 3.1, we know that both multiplication and inversion are the time critical operations and therefore the need to minimize their number when doing arithmetic with elliptic curves. Chapter 7 will introduce a new

method of doubling elliptic curve points that minimizes the number of inversions at the expense of more multiplications.

2.2.3 Multiples of Points

When implementing an elliptic curve cryptosystem, such as those described in Chapter 8, one is required to compute

$$eP = \underbrace{P + P + \cdots + P}_{e \text{ times}},$$

where e is a positive integer and $P \in E$. For very large values of e, a straightforward summation becomes impractical and so we use a method which is analogous to the square and multiply algorithm for exponentiation [Knu81]. This method is known as "repeated double and add" and it is described in Theorem 1 [MQV95].

Theorem 1 *Let $P \in E$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_2$ be the binary representation of the multiplier e where the most significant bit e_t of e is 1. Then, $Q = eP$ can be computed using the following algorithm.*

Algorithm (Input: $P = (x, y), e$; Output: $Q = eP$)
1. $Q \leftarrow P$
2. For $i = t - 1$ to 0
   - 2.1 $Q \leftarrow Q + Q$
   - 2.2 If $e_i = 1$, then $Q \leftarrow Q + P$
4. Return(Q)

Theorem 1 implies that for a randomly selected integer e with $t + 1$ bits, one needs t doubling steps and an average $t/2$ adding steps. Notice, however, that improved methods for exponentiation have been suggested. Some of them will be explored in Chapters 4 and 7.

Chapter 3

Computer Arithmetic in Composite Fields and Elliptic Curves

This chapter introduces the way in which we represent Galois field elements in software implementations. We also present the algorithms used to implement in software addition, multiplication, and inversion over Galois fields. Finally, we introduce the concept of the Table Look-Up (TLU) which will be fundamental to the discussions about algorithmic complexity in Chapter 5.

3.1 Computer Representation of Galois Field Elements

When optimizing Galois field arithmetic, it is necessary to have a good understanding of the internal representation of the Galois field elements in the computer. The

software implementation that was used in realizing this work used three different libraries to deal with Galois fields $GF((2^n)^m)$ [HPR].

3.1.1 Composite Field Libraries

The first library implemented, called `gfopsn.c`, defines the type `gfelt` which is used to deal with elements of the ground field $GF(2^n)$, where $n \leq 32$. Notice that these field elements are represented by integer variables. The field polynomial $Q(y)$ is determined by the user. The second library is called `polyopsn.c`. This library defines the type `gfpoly` to perform different arithmetic operations with polynomials over $GF(2^n)$. The `gfpoly` type was defined as an array of `gfelt` elements. The elements of the array `gfpoly` signify the coefficients of the polynomial. The coefficient of x^i, is stored at position i+2. Position 0 of the array is reserved for an integer indicating the degree of the polynomials, whereas position 1 of the array is reserved for an integer indicating the number of memory cells allocated. It is important to point out that the polynomials themselves are not elements of $GF(2^n)$, but rather the coefficients of the polynomials.

The last library corresponds to the file `cgfops.c`. In this library the type `comp` was defined to deal with composite field elements or elements of the Galois field $GF((2^n)^m)$. Notice that this new type was defined the same as the `gfpoly` type but the elements of this field are constructed from the polynomial $P(x)$. Thus, if $A(x) \in GF((2^n)^m)$, we can represent it as follows:

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} \cdots + a_1x + a_0, \quad a_i \in GF(2^n) \tag{3.1}$$

Furthermore, the polynomial $A(x)$ is represented as an array of `gfelts` $(a_{m-1} \cdots a_1 a_0)$ of length m+2 (m entries for m coefficients plus two entries for the degree and the total

size of the array, as described above). Notice that `cgfops.c` allows field arithmetic in $GF((2^n)^m)$, by performing polynomial arithmetic modulo $P(x)$.

3.1.2 Application to $GF((2^{16})^{11})$

In this section, we consider the special case of the Galois Field $GF((2^{16})^{11})$. Notice that $\gcd(16, 11) = 1$ thus, if we choose a polynomial of degree 11 which is irreducible over $GF(2)$, it will also be irreducible over $GF(2^{16})$.

In order to construct the field $GF((2^{16})^{11})$, we need to choose irreducible polynomials for both the ground field $GF(2^{16})$ and the composite field $GF((2^{16})^{11})$. From [LN83], we chose $Q(y) = y^{16} + y^{11} + y^6 + y^5 + 1$ and $P(x) = x^{11} + x^2 + 1$. It is important to point out that because $P(x)$ is a trinomial, the complexity of the modular reduction will be minimal as opposed to choosing a $P(x)$ with more coefficients.

From Section 3.1, we know that an element $A(x) = a_{10}x^{10} + a_9x^9 \cdots + a_1x + a_0$, $a_i \in GF(2^{16})$, of the composite field $GF((2^{16})^{11})$ will be represented by an integer array with 11 entries, each entry being an element of type `gfelt` representing a coefficient a_i of the polynomial $A(x)$. Each coefficient a_i in turn is represented as a 16-bit integer variable.

3.2 Computer Arithmetic in Composite Fields

This section introduces the concept of the table look-up as it applies to arithmetic in composite fields. In addition, it describes in a more detailed manner the type of arithmetic operations that are defined in some of the libraries that were introduced in Section 3.1 and the way these libraries are implemented in software.

3.2.1 Multiplication in $GF(2^n)$

The major advantage of using composite Galois fields is that most of the time consuming operations in the ground field, such as inversion and multiplication, can be accelerated using table look-up (TLU) [Bea96, WBV+96]. Table look-ups are based on the idea of precomputing "log" and "antilog" tables in the ground field $GF(2^n)$. By log, one means the analog of the logarithm function in a discrete sense. In other words, given that all non-zero elements of $GF(2^n)$ can be represented as the power of a primitive element ω, we define the log function as follows:

$$k = \log(\omega^k), \quad \omega^k \in GF(2^n), k \in \{1, 2, \cdots, 2^n - 1\} \tag{3.2}$$

where $GF(2^n) = \{0, \omega, \omega^2, \cdots, \omega^{2^n-1} = 1\}$.

By antilog we mean the inverse of the log function. In other words, if the log function is as defined in (3.2) then (3.3) defines the antilog function as follows:

$$\omega^k = \text{antilog}(k) = \text{antilog}(\log(\omega^k)), \quad \omega^k \in GF(2^n), k \in \{1, 2, \cdots, 2^n - 1\} \tag{3.3}$$

(3.2) and (3.3) will prove to be fundamental for efficient software implementations as described in Chapter 6.

Finally, the product of two elements $\omega_i, \omega_j \in GF(2^n)$ can be reduced to an integer addition and three table look-ups as follows:

$$\omega_i \omega_j = \text{antilog}(\log(\omega_i) + \log(\omega_j)) \pmod{2^n - 1} \tag{3.4}$$

3.2.2 Addition in $GF((2^n)^m)$

Addition in $GF((2^n)^m)$ is performed by using a routine defined in the library `polyopsn.c`. The function definition is `polysum(A, B, *X)` where `A`, `B`, and `*X` are all of type `gfpoly` or, in other words, polynomials with coefficients in $GF(2^n)$. The same routine is also called by `cgfopsn.c` to perform addition of composite field elements. Composite field addition, $C(x) = A(x) + B(x),\ A(x), B(x), C(x) \in GF((2^n)^m)$, amounts to adding the coefficients of two composite field elements in standard base representation as follows

$$\begin{aligned} C(x) &= (c_{m-1}x^{m-1} + \cdots + c_1x + c_0) = \\ &= (a_{m-1}x^{m-1} + \cdots + a_1x + a_0) + (b_{m-1}x^{m-1} + \cdots + b_1x + b_0) \end{aligned} \tag{3.5}$$

where $a_i + b_i = c_i \in GF(2^n)$ and $C(x) \in GF((2^n)^m)$. This operation can be performed by bitwise XORing of the coefficients of the polynomials. Notice that one only needs a loop with m iterations which is fast.

3.2.3 Multiplication in $GF((2^n)^m)$

The multiplication of two composite field elements $A(x), B(x) \in GF((2^n)^m)$ can be performed in standard base representation as follows:

$$C(x) = A(x) \times B(x) \bmod P(x), \tag{3.6}$$

where $P(x)$ is the irreducible polynomial of the field $GF((2^n)^m)$. The field multiplication can be performed in two steps:

1. Ordinary polynomial multiplication ($\times$)

2. Reduction modulo the field polynomial (*mod*).

Step 1 will be treated in detail in Chapter 5 where the Karatsuba-Ofman algorithm is studied. The following will explain how the reduction modulo the field polynomial is performed. Notice that in both steps, the basic arithmetic operations, addition and multiplication of polynomials, are performed in the ground field $GF(2^n)$. As stated in previous sections, the additions are XOR operations and the multiplications are table look-ups as shown in Section 3.2.1.

In order to perform modular reduction in $GF((2^n)^m)$ we will consider the intermediate product $C'(x) = A(x) \times B(x)$. $C'(x)$ is a polynomial over $GF(2^n)$ with $deg(C'(x)) \leq 2m-2$ with coefficients $c'_i \in GF(2^n)$. Then, we can represent $C(x)$ where $deg(C(x)) \leq m-1$ as follows:

$$\begin{aligned} C(x) &= C'(x) \bmod P(x) \\ &= c'_{2m-2}x^{2m-2} + \cdots + c'_0 \bmod P(x) \\ &= c_{m-1}x^{m-1} + \cdots + c_0 \end{aligned} \tag{3.7}$$

The reduction modulo $P(x)$ can be viewed as a linear mapping of the $2m-1$ coefficients of $C'(x)$ into the m coefficients of $C(x)$. This mapping can be represented in a matrix notation as follows:

$$\begin{pmatrix} d_0 \\ d_1 \\ \vdots \\ d_{m-1} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 & r_{0,0} & \cdots & r_{0,m-2} \\ 0 & 1 & \cdots & 0 & r_{1,0} & \cdots & r_{1,m-2} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & r_{m-1,0} & \cdots & r_{m-1,m-2} \end{pmatrix} \begin{pmatrix} d'_0 \\ \vdots \\ d'_{m-1} \\ d'_m \\ \vdots \\ d'_{2m-2} \end{pmatrix}. \tag{3.8}$$

The matrix on the right hand side of (3.8) consists of a m by m identity matrix and a m by $m-1$ reduction matrix, which is a function of the chosen monic field polynomial $P(x)$ [Paa94]. If $P(x)$ has only coefficients from $GF(2)$, the entries $r_{i,j}$ of the reduction matrix are binary, allowing one to compute $C(x)$ with only additions in the ground field.

3.2.4 Squaring in $GF((2^n)^m)$

The squaring operation is a linear operation for composite fields of characteristic 2 as it is the case for $GF((2^n)^m)$. Thus, given $A(x) \in GF((2^n)^m)$ as defined in (3.1), $A^2(x)$ can be computed as follows [McE87]:

$$A^2(x) = a_{m-1}^2 x^{2(m-1)} + a_{m-2}^2 x^{2(m-2)} \cdots + a_1^2 x^2 + a_0^2 \bmod P(x) \tag{3.9}$$

where the reduction modulo $P(x)$ is performed as indicated in (3.8). Therefore, it is easy to see that in performing the squaring operation one does not need to perform most of the products that are computed in the multiplication of two general composite field elements, thus reducing the complexity of the squaring operation to m ground field multiplications. As in the case of general multiplication, only additions and multiplications (done through table look-ups) of ground field elements are required in the squaring operation.

3.2.5 Inversion in $GF((2^n)^m)$

The most costly operation in a software implementation that deals with Galois field arithmetic is inversion. Several algorithms have been proposed to accomplish this. In Chapter 4 we introduce the so called "Almost Inverse Algorithm" proposed in [SOOS95]. However, the method used in this implementation is based on a modified

version of Itoh and Tsujii's Algorithm for inversion in composite fields [Paa95]. This algorithm is studied in detail in Chapter 6. For now, it suffices to say that the idea behind the algorithm is to reduce inversion in $GF((2^n)^m)$ to inversion in the ground field $GF(2^n)$ through the use of table look-ups.

3.2.6 Arithmetic in $GF((2^{16})^{11})$

Again, we consider the special case of the Galois field $GF((2^{16})^{11})$. Chapter 5 will deal extensively with the multiplication of polynomials in this field, and Chapter 6 will discuss an inversion algorithm. We will now describe how modulo reduction and squarings are performed in this field. From Section 3.2.3, we know that the modulo reduction can be based on a matrix description. We chose $P(x)$ to be the irreducible polynomial $x^{11} + x^2 + 1$. For the Galois field $GF((2^{16})^{11})$, the matrix in (3.8) is entirely binary and is shown in Figure 3.1. Notice that computation of the matrix vector product requires only 21 coefficient additions in the ground field $GF(2^{16})$.

$$\begin{pmatrix} c_o \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \\ c_8 \\ c_9 \\ c_{10} \end{pmatrix} = \left(\begin{array}{ccccccccccc|cccccccccc}
1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1\\
0&1&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0\\
0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0&1\\
0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0&0\\
0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0&0\\
0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0&0\\
0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0&0\\
0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0&0\\
0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1&0\\
0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0&1\\
0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&1&0
\end{array}\right) \begin{pmatrix} c'_o \\ c'_1 \\ c'_2 \\ c'_3 \\ \vdots \\ c'_{10} \\ c'_{11} \\ \vdots \\ c'_{18} \\ c'_{19} \\ c'_{20} \end{pmatrix}$$

Figure 3.1: Matrix for fast modulo reduction with $P(x) = x^{11} + x^2 + 1$

We consider now the squaring operation in the field $GF((2^{16})^{11})$. In our implementation, given a composite field element $A(x)$, where: $A(x) = a_{10}x^{10} + \ldots + a_1x + a_0$, $a_i \in GF(2^n)$, the squared element is: $A^2(x) = a_{10}^2x^{20} + a_9^2x^{18} + \cdots + a_1^2x^2 + a_0^2$. We

now can use a modified form of the reduction matrix given in Figure 3.1 to find the resulting coefficients for $A^2(x)$. Thus, we are required to perform 22 table look-ups and 6 additions in the subfield $GF(2^{16})$.

3.3 Computer Representation of Elliptic Curve Elements

As it was stated in Section 2.2.1, every point on an elliptic curve is described by the coordinates (x, y). In our software implementation, a new data type was defined to represent elliptic curve points. This new data type was called `ellc_elt`, where:

```
typedef struct ellc_elt
{
   comp pointx;              /* Composite field element      */
   comp pointy;              /* Composite field element      */
   int    infinity;          /* point of infinity = 1, else 0 */
} ellc_elt;
```

The coordinates x and y are represented by a composite field element of type `comp` (as defined in Section 3.1). In addition, the variable `infinity` will be used to indicate whether the point at infinity $\mathcal{O}$ has occurred.

Chapter 4

Previous Work

This chapter summarizes previous works on elliptic curve cryptosystems presented in the research community in the past. We will introduce some of the previous elliptic curve implementations found in the literature, both over $GF(2^k)$ and $GF((2^n)^m)$. We will also summarize the work in each of following areas: efficient inversion and efficient multiplication algorithms for arithmetic in $GF((2^n)^m)$ and elliptic curve point addition algorithms. Finally, we will present some of the improved algorithms used to compute the product of an elliptic curve point by a large integer.

4.1 Elliptic Curve Implementations over $GF(2^k)$

4.1.1 Early work

The earliest references to the way in which elliptic curves are implemented are [MV90] and [MV93]. In both references, the authors briefly discuss how arithmetic in $GF(2^k)$ can be be efficiently implemented in hardware. The selection of a curve and a finite field that minimizes the number of field operations is treated as well as alternate

ways to addition of points in an elliptic curve. Finally, the elliptic curve analog of the ElGamal cryptosystem is analyzed and estimates for its throughput are given.

The underlying idea in [MV90] and [MV93] is to view the field $GF(2^k)$ as a k-dimensional vector space over $GF(2)$ and define a basis for it. Once the basis has been defined, the elements of the finite field $GF(2^k)$ can be conveniently represented as 0–1 vectors of length k. Addition can be implemented by bitwise XORing the vector representations which takes only a few clock cycles (Notice that it the addition operation will only take one cycle if the processor has word length equal to k). [MV90] and [MV93] also introduce the idea of optimal normal basis for efficient implementation of both squaring (one clock cycle to perform by shifting the vector components)and multiplication (takes k clock cycles if optimal normal bases exist for the field, otherwise low-complexity normal basis should be considered). The Itoh and Tsujii algorithm [IT88] is introduced as the most efficient technique to compute an inverse based on Fermat's Little Theorem. However, the algorithm is costly for hardware implementations because it requires the storage of several intermediate steps and therefore an alternate algorithm to perform inversion in a Galois field is also mentioned (see [ABMV93]).

In [MV90] and [MV93] supersingular elliptic curves were chosen to implement the ElGamal cryptosystem because it is possible to avoid the inversion operation in doubling a point if the coefficients of the elliptic curve are carefully chosen. Nevertheless, the addition of two different points still requires the computation of inverses in the underlying finite field and these, despite the existence of efficient algorithms to compute them, are by far the most costly operations to perform. Thus, [MV90] and [MV93] introduce projective coordinates as a method of adding two different elliptic curve points without resorting to the inverse operation. Using the repeated double and add method described in Theorem 1, the product of a point $P \in E$ by an integer e with $H_w(e) = t + 1$ will take $2t$ multiplications and t inversions using affine

coordinates ($H_w()$ denotes the Hamming weight of the binary representation of the operand). On the other hand, if using projective coordinates and the same exponentiation method the total operation count is reduced to $9t+2$ multiplications and one inversion. This reduction in inversions occurs at the expense of extra multiplications and of space since one needs extra registers to store the intermediate results in the algorithm.

[MV93]also provides throughput estimates for the encryption rate using the elliptic curve analog of the ElGamal cryptosystem. A supersingular elliptic curve $E: \; y^2+y = x^3+x+1$ over $GF(2^{239})$ was chosen. It was assumed that multiplication took 239 cycles and inversion using the Itoh and Tsujii algorithm $\lfloor \log_2(239-1) \rfloor + H_w(239-1)-1$ multiplications. Elements of the field $GF(2^{239})$ were represented using optimal normal basis. Since the field size is small, it was assumed that the number of registers was not important and thus projective coordinates were chosen to represent the points on the curve. Finally, the Hamming weight of the multiplier was limited to 30 or less, thus increasing the speed of the system and putting an upper bound on the time required for the encryption. With these assumptions and assuming a 40 MHz clock rate, 145 kbits/s is achieved. Notice that for this specific curve, the computation of logarithms is believed to be as hard as for logarithms computed in $GF(2^{4k})$ using the MOV attack thus, the index calculus method is infeasible with current computer resources, but might be within reach in the near future..

Non-supersingular curves are also introduced. They are very attractive for security reasons because the MOV attack can not be applied to them thus the size of the underlying Galois field can be smaller than for supersingular curves. As in the case of the supersingular curves, the underlying field should also have an optimal normal basis in order to achieve efficient field arithmetic and the need for the inverse computation may be eliminated by resorting to projective coordinates.

Finally, [MV93] mentions an implementation of a cryptosystem in hardware. The implementation explored a chip explicitly designed to perform efficient elliptic curve point addition over the field $GF(2^{155})$ and it will be the subject of Section 4.1.2.

4.1.2 Hardware Implementation over $GF(2^{155})$ Using Normal Basis Representation

[AMV93] discusses a VLSI implementation of an arithmetic processor in $GF(2^{155})$ which allows an efficient implementation of a non-supersingular elliptic curve cryptosystem. Following the same approach as in [MV93] and [MV90], optimal normal basis are chosen to perform arithmetic in the underlying field $GF(2^{155})$. Addition was implemented by simply XORing the vector representations of the finite field elements in $GF(2^{155})$. The normal base multiplier required three registers A, B, and C, where C was the product register. The C register contained logic to interconnect the A and B registers. In addition, the registers were directly interconnected to allow fast register transfers. Since computations on elliptic curves were very I/O intensive, a 32-bit wide I/O structure was incorporated into the device which also allowed for 16- and 8-bit transfers.

This architecture implied that each register was treated as five individually addressable 32-bit locations (5 padding bits were used in the high order bits) which allowed to load or unload an entire register in 10 clock cycles or 250 nanoseconds. The last optimization in the design was to use a simple instruction set, flexible enough to allow the implementation of a variety of functions. The instruction set was divided into two types: instructions for elementary register operations and instructions for more involved field operations.

It is important to point that [AMV93] were able to implement the coprocessor

in a relatively small custom gate array by restricting its functionality to the minimum necessary to implement the elliptic curve cryptosystem. The final device required the equivalent of about 11000 gates and ran at a 40 MHz clock rate, requiring less than 15 percent of the area of currently available smart card chips.

As in [MV93] and [MV90], projective coordinates were used to implement point addition thus avoiding the inverse operation. Addition of two different points in a non-supersingular elliptic curve thus required 13 field multiplications and a doubling required 7 multiplications. Assuming a Hamming weight multiplier of weight 20 (notice that this does not compromise the security of the system) and a clock rate of 40 MHz, the approximate throughput rate on any non-supersingular elliptic curve over $GF(2^{155})$ is 60 kbits per second (notice that for the unrestricted multiplier the throughput was estimated to be 40 kbits per second).

4.1.3 Software Implementation over $GF(2^{155})$ Using Standard Basis Representation

The contribution by [SOOS95] describes an implementation of the Diffie-Hellman (DH) key exchange protocol using a non-supersingular elliptic curve over the field $GF(2^{155})$. In this implementation, the constant a in (2.3) is chosen to be equal to zero. With this choice, one effectively eliminates one addition from the calculation of the x-coordinate of a point P for both addition and doubling of elliptic curve points.

The computation of eP, where e is a large integer and P is a point on the elliptic curve, is crucial to the speed of the key exchange or digital signature generation. In particular, the number of point additions and point doublings should be minimized. For a randomly chosen 155-bit multiplier, the computation of eP will require on average 154 point doublings and 77 point additions using the standard

double and add algorithm. Although, the number of doublings is essentially fixed, it is possible to reduce the number of additions. [SOOS95] implemented speed–ups to this end.

In the initial phase of the DH protocol both parties choose a primitive element P_0. Then, they generate their secret keys $a_A, a_B \in Z$ and compute their public keys $K_{pubA} = a_A P_0$ and $K_{pubB} = a_B P_0$, both points on the elliptic curve. Since P_0 is a system parameter, the preparation of tables of multiples of P_0 are useful. [SOOS95] used a precomputed table which consisted of $(16)^q \cdot P_0$ for $q = 0, \cdots, 38$, to compute eP, where e is a 155-bit integer, with 42 point additions and no doublings. In Section 4.3, we discuss this method further which was introduced by [BGMW92].

In the second phase of the DH protocol, one is required to compute a multiple of point P (the public key of the other party) not known ahead of time. For this situation [SOOS95] implemented a blend of the k-ary method [Knu81] and Booth's algorithm. In this case, a table of the odd multiples of P is computed from P to $15P$. Then, using the table, several doubling steps can be performed before an addition is necessary. In the case that an even multiple of P is necessary, an odd multiple of P introduced a couple of steps earlier in the doubling process is used instead. In addition, since subtraction of points is not more costly than addition one has the option of subtracting a point when necessary. On the average, for a random 155-bit multiplier, one is required to perform 152 doubling steps and 32 additions/subtractions, including the precomputation.

The operations of addition, squaring, and multiplication in the underlying Galois Field that are needed to compute the addition or doubling of two elliptic curve points are implemented using the same principles that were described in Section 3.2, using the irreducible polynomial $P(x) = x^{155} + x^{62} + 1$ (Notice that [SOOS95] do not use the Karatsuba-Ofman algorithm or other advanced multiplication algorithms for

multiplication but rather ordinary polynomial multiplication). However, they develop a new algorithm for inversion based on the Euclidean division algorithm.

The "Almost Inverse Algorithm" computes $B(x) \in GF(2^{155})$ and r such that:

$$A(x)B(x) = x^r \bmod P(x) \tag{4.1}$$

where $\deg(B(x)) < \deg(P(x))$ and $r < 2\deg(P(x))$. After executing the algorithm one only needs to divide $B(x)$ by x^r to find the inverse $A(x)$. The algorithm is described in Theorem 2.

Theorem 2 *Let $A(x), B(x), x^r \in GF(2^k)$ and let $P(x)$ be the field polynomial of $GF(2^k)$. Then $A^{-1}x^r \bmod P(x)$ can be computed using the following "Almost Inverse Algorithm" as follows.*

Algorithm (Input: $A(x), P(x)$; Output: $A^{-1}x^r$)
1. Initialization.
   - 1.1 $r \leftarrow 0$
   - 1.2 $B(x) \leftarrow 1$
   - 1.3 $C(x) \leftarrow 0$
   - 1.4 $F(x) \leftarrow A(x)$
   - 1.5 $G(x) \leftarrow P(x)$
2. loop: While $F(x)$ is even, do
   - 2.1 $F(x) \leftarrow F(x)x^{-1}$
   - 2.2 $C(x) \leftarrow C(x)x$
   - 2.3 $k \leftarrow k + 1$
3. If $F(x) = 1$, then return $B(x)$ and k
4. If $\deg(F(x)) < \deg(G(x))$, then
   - 4.5.1 $F(x) \longleftrightarrow G(x)$ (exchange $F(x), G(x)$)
   - 4.5.2 $B(x) \longleftrightarrow C(x)$ (exchange $B(x), C(x)$)
5. $F(x) \leftarrow F(x) + G(x)$
6. $B(x) \leftarrow B(x) + C(x)$
7. Goto loop.

In order to obtain the inverse of $A(x)$ one needs to divide $B(x)$ out by x^r working modulo $P(x)$. The strategy proposed in [SOOS95] is to successively divide

Operations	Type of Operation	Average Timing
Basic Field Operations	155 bit square	1.65 μsec
	155 x 155 bit multiply	7.10 μsec
	155 bit inverse	25.21 μsec
Elliptic Curve Operations	Multiply new elliptic curve point	7.8 msec
	Multiply known elliptic curve point	1.8 msec

Table 4.1: Timings for various field and elliptic curve operations [SOOS95].

$B(x)$ by u^w where w is the word size of the computer and finish up with a division by a smaller power of x. Two parts can be distinguish in this procedure. First, a suitably chosen multiple of $P(x)$ (P(x) is multiplied by the least significant w bits in $B(x)$) is added to $B(x)$ so as to zero out the w low order bits of $B(x)$. Second, the new $B(x)$ is right shifted by w bits, effectively dividing it by x^r.

[SOOS95] used a DEC Alpha 3000 with a 175 MHz RISC Architecture and a 64 bit word size. Their timings are summarized in Table 4.1. These timings will be highly relevant for our work in later chapters.

4.2 Elliptic Curve Implementations over Composite Fields

Software implementations of EC over composite Galois field $GF((2^n)^m)$ were first described in [HMV92] for the field $GF((2^8)^{13})$. More recently, EC systems for the field $GF((2^{16})^{11})$ were described independently in [WBV$^+$96] and [Bea96]. The purpose of this section is to summarize their findings and contributions. In addition, timings of the different implementations will be provided.

4.2.1 Implementation over $GF((2^8)^{13})$

In this paper, [HMV92] compared the basic operations of squaring, multiplication, and inversion for the fields $GF(2^{105})$ and $GF((2^8)^{13})$. Arithmetic in $GF(2^{105})$ was implemented by using normal basis representation. Addition is accomplished by XORing the two vectors representing the elements to be added. Inversion, on the other hand, is computed by first converting to a polynomial basis representation of $GF(2^{105})$ using a precomputed change of basis matrix, compute the inverse in polynomial base representation using the Euclidean algorithm, and finally, convert back to normal base representation. Notice, that implementations will more efficient when using optimal normal basis as opposed to a randomly chosen normal basis.

Arithmetic in the field $GF((2^8)^{13})$ was implemented using a composite field polynomial basis representation or, in other words, by looking at the field $GF(2^{104})$ as a vector space over $GF(2^8)$ rather than over $GF(2)$. The field polynomial of $GF(2^8)$ was chosen to be the irreducible polynomial $Q(y) = y^8 + y^7 + y^3 + y^2 + 1$. The paper proposes for the first time the idea of table look-ups. Two tables "log" and "antilog" were defined as in (3.2) and (3.3). Then, multiplication and inversion in the ground field are accomplished by table look-up (see Section 3.2.1).

The polynomial $P(x) = x^{13}+x^7+x^6+x+1$ is chosen to be the field polynomial of the composite field $GF((2^8)^{13})$. Consequently, elements of this field are represented as polynomials over $GF(2^8)$ with maximum degree equal to 12 and all arithmetic done modulo $P(x)$. Squaring and multiplication in the composite field are implemented as described in Sections 3.2.3 and 3.2.4 with multiplication of coefficients being done through table look-up. Inversion, on the other hand, is implemented by using the extended Euclidean algorithm. Table 4.2 shows a comparison of the timings of the basic field operations on a SUN SPARCstation-2.

Non-supersingular elliptic curves were chosen to implement the cryptosystem.

Operations	Method/Type of Operation	$GF(2^{105})$	$GF((2^8)^{13})$
Field operations	10000 squarings	0.02	0.15
	10000 multiplications	7.6	1.95
	10000 inversions	15.3	9.46
Elliptic Curve Operations	10000 curve additions	33.9	14.4
	10000 curve doublings	38.6	16.2
	eP using double and add	—-	0.24
	eP using Brickell's method	—-	0.052

Table 4.2: Comparison of timings for elliptic curve and field operations using normal and polynomial basis representations (time in seconds)[HMV92].

They conclude that although hardware implementations can take advantage of projective coordinates (thus avoiding inversions), it seems that in software the affine representation is superior. Finally, they suggest an alternate method to the double and add algorithm for computing the multiple of a point. This method is developed in [BGMW92] and it will be treated in Section 4.3. Table 4.2 shows the timings for the basic curve operations (doubling and addition of points) as well as for the computation of the multiple of a point.

4.2.2 Implementations over $GF((2^{16})^{11})$

In both [WBV+96] and [Bea96] elliptic curve cryptosystems were implemented using non-supersingular elliptic curves. Both implementations explore arithmetic over the composite fields $GF((2^{16})^{11})$ using polynomial basis representations over $GF(2^{16})$. In both contributions, addition, squaring, and multiplication are implemented in the same form, as explained in Sections 3.2.2, 3.2.4, and 3.2.3. Notice that multiplication of the coefficients of two elements of the field $GF((2^{16})^{11})$ is accomplished through table look-up as proposed in [HMV92]. Inversion, however, is very different in both implementations.

[WBV+96] explores two approaches. The first approach consists of computing

the inverse of an element $A(x) \in GF((2^n)^m)$ using the extended Euclidean algorithm which is described in Theorem 3. In this theorem the following notation is used: Given a polynomial $A(x)$ of degree $m-1$, A_0 represents the number of coefficients in $A(x)$ (or m) and A_i represents the $i-1$ coefficient if $A(x)$ is represented as in (3.1). [WBV+96] concludes that this algorithm is faster in a polynomial base for composite fields than in fields $GF(2^k)$.

The second approach is the Almost Inverse algorithm explained in Theorem 2. [WBV+96] compares the two algorithms and observes that their behavior is very similar. The main difference is that the Almost Inverse algorithm cancels powers of x from lower degree to higher degree whereas the extended Euclidean algorithm moves from higher degree to lower degree. This translates into two important benefits of the Almost Inverse algorithm when working in standard base representation. However, if working in polynomial base representation this advantages are irrelevant and thus, [WBV+96] conclude that the Euclidean division algorithm is slightly more efficient since it returns the inverse at the end of its execution as opposed to the Almost Inverse algorithm where one still has to divide by x^r.

Theorem 3 *Let $A(x), B(x), \in GF((2^n)^m)$ and let $P(x)$ be the field polynomial of $GF((2^n)^m)$. Then $B(x) = A^{-1} \bmod P(x)$ can be computed using the extended Euclidean algorithm as follows.*

Algorithm (Input: $A(x), P(x)$; Output: A^{-1})
1. Initialization.
   - 1.1 $B(x) \leftarrow 1$
   - 1.2 $C(x) \leftarrow 0$
   - 1.3 $F(x) \leftarrow A(x)$
   - 1.4 $G(x) \leftarrow P(x)$
2. Do
   - 2.1 If $\deg(F(x)) = 0$ then return $B(x)/F_1$
   - 2.2 If $\deg(F(x)) < \deg(G(x))$ then
     - 2.2.1 $F(x) \longleftrightarrow G(x)$ (exchange $F(x), G(x)$)
     - 2.2.2 $B(x) \longleftrightarrow C(x)$ (exchange $B(x), C(x)$)

2.3 $j = \deg(F(x)) - \deg(G(x))$
2.4 $\alpha = F_{F_0}/G_{G_0}$
2.5 $F(x) \leftarrow F(x) + \alpha x^j G(x)$
2.6 $B(x) \leftarrow B(x) + \alpha x^j C(x)$
Goto 2

Finally, the computation of the multiple of a point is achieved by implementing the double and add/subtract algorithm. [WBV+96] use a 177-bit multiplier which means that 176 doublings and an average of 59 additions/subtractions are required for one "exponentiation."

[Bea96] explores an implementation of the inversion operation originally described in [IT88]. In this paper, an efficient method for inversion in $GF(2^k)$ based on Fermat's little theorem is discussed. In Section 6 of the reference the method is extended to composite Galois fields $GF((2^n)^m)$ using normal base representation. On the other hand, this inversion algorithm was optimized for composite fields in polynomial base representation in [Paa94] and [Paa95] and it will be treated extensively in Chapter 6 of this thesis. In addition, the analog of an exponentiation operation for elliptic curves was implemented using the double and add algorithm described in Theorem 1. Thus, a 176-bit multiplier e meant that eP would take 175 doublings and an average of 86 addition steps. Finally, [Bea96] implemented the elliptic curve analog of the Diffie-Hellman Key exchange. Table 4.3 compares the timing results that both [WBV+96] and [Bea96] presented in their contributions. Notice that [WBV+96] measurements were performed in a Pentium/133 based PC whereas [Bea96] used a DEC Alpha 3000, 175 MHz RISC architecture with a 64-bit word size.

Operations	Method/Type of Operation	[WBV+96]	[Bea96]
Field operations	176 bit add	1.2	1.19
	176 bit squaring	5.9	4.23
	176 bit multiply	62.7	43
	176 bit inversion	160	285
Elliptic Curve Operations	Addition	306 (estimate)	452
	Doubling	309 (estimate)	441
	Point multiplication	72 ms (estimate)	123 ms
	Elliptic curve key exchange	—-	246 ms

Table 4.3: Comparison of timings for elliptic curve and field operations between elliptic curve implementations over $GF((2^{16})^{11})$. NOTE: unmarked times are in microseconds.

4.3 Efficient Exponentiation

The problem of multiplying a point P of an EC by a (large) integer k is analogous to exponentiation of an element in a multiplicative group to the kth power. The standard algorithm for this problem is the binary exponentiation method (or square-and-multiply algorithm) which is studied in detail in [Knu81] and versions of which have been adapted to the elliptic curve case in Theorem 1. A generalization of the binary method is the k-ary method [Coh93, Koc95, MvOV97] which processes k exponent bits in one iteration. Theorem 4 was adapted from [MvOV97] and it describes the algorithm as it is applied to elliptic curves.

Theorem 4 *Let $P \in E$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$ be the radix representation of the multiplier e in base b where $b = 2^k$ for $k \geq 1$. Then, $Q = eP$ can be computed using the following algorithm.*

Algorithm (Input: $P = (x, y), e$; Output: $Q = eP$)
1. Precomputation
    1.1 $P_0 \leftarrow \mathcal{O}$ (Point at infinity)
    1.2 For $i = 1$ to $2^k - 1$
        $P_i = P_{i-1} + P$ (i.e., $P_i = i * P$)
2. $Q \leftarrow \mathcal{O}$

3. For $i = t$ to 0
   3.1 $Q \leftarrow 2^k Q$
   3.2 $Q \leftarrow Q + P_{e_i}$
4. Return(Q)

Notice that Step 3.1 in the algorithm involves the doubling of point Q, k times, and Step 3.2 requires one point addition. Thus, the complexity of the k-ary method with t iterations is kt point doublings, t point additions from the loop in Step 3, and $2^k - 2$ point additions from the precomputation in Step 1.2 (One should not count the first addition in Step 1.2 since P is added to the point at infinity).

Theorem 5 *Let $P \in E$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_b$ be the radix representation of the multiplier e in base b where $b = 2^k$ for $k \geq 1$. Also, for each i such that $0 \leq i \leq t$, if $e_i \neq 0$, then write $e_i = 2^{h_i} u_i$ where u_i is odd; if $e_i = 0$ then let $h_i = 0, u_i = 0$. Then, $Q = eP$ can be computed using the following algorithm.*

Algorithm (Input: $P = (x, y), e$; Output: $Q = eP$)
1. Precomputation
   1.1 $P_0 \leftarrow \mathcal{O}$ (Point at infinity)
   1.2 $P_1 \leftarrow P$
   1.3 $P_2 \leftarrow 2P$
   1.4 For $i = 1$ to $2^{k-1} - 1$
      $P_{2i+1} = P_{2i-1} + P_2$
2. $Q \leftarrow \mathcal{O}$
3. For $i = t$ to 0
   $Q \leftarrow 2^{h_i}(2^{k-h_i} Q + P_{u_i})$
4. Return(Q)

Further improvements of the k-ary method include the improved k-ary method and the sliding window method both treated in [MvOV97] and [Koc95]. Theorems 5 and 6 have been adapted from [MvOV97] and describe these algorithms.

Theorem 6 *Let $P \in E$ and $e = (e_t e_{t-1} \cdots e_1 e_0)_2$ be the binary representation of the multiplier e together with an integer $k \geq 1$ (window size). Then, $Q = eP$ can be computed using the following algorithm.*

Algorithm (Input: $P = (x, y), e$; Output: $Q = eP$)
1. Precomputation
    1.1 $P_1 \leftarrow P$
    1.2 $P_2 \leftarrow 2P$
    1.3 For $i = 1$ to $2^{k-1} - 1$
        $P_{2i+1} = P_{2i-1} + P_2$
2. $Q \leftarrow \mathcal{O}$
3. $i \leftarrow t$
4. While $i \geq 0$ do
    4.1 If $e_i = 0$ then
        4.1.1 $Q \leftarrow 2Q$
        4.1.2 $i \leftarrow i - 1$
    4.2 Else, find the longest bit-string $e_i e_{i-1} \cdots e_l$
        such that $i - l + 1 \leq k$ and $e_l = 1$, and do the following
        4.2.1 $A \leftarrow 2^{i-l+1} A + P_{(e_i e_{i-1} \cdots e_l)_2}$
        4.2.2 $i \leftarrow l - 1$
5. Return(Q)

[Koc95] performs a detailed analysis of all window techniques for exponentiation. In addition, Chapter 7 will explore the application of several of this techniques to elliptic curve cryptosystems over composite fields $GF((2^{16})^{11})$ and it will provide a complexity analysis of this algorithms for special cases.

Finally, in some protocols, like in the calculation of the public key in the Diffie-Hellman key exchange, the point P, input to these exponentiation algorithms, is known ahead of time. In these cases, it is possible to apply an algorithm introduced in [BGMW92]. This algorithm is described in Theorem 7 which has been adapted from the original paper to the case of elliptic curves.

Theorem 7 *Let $P \in E$ and let h and $e = \sum_{i=0}^{t} e_i s_i$ be positive integers with $0 \leq e_i \leq h$ and $0 \leq i \leq t$. Now, suppose the products $P_i = s_i P$ for $0 \leq i \leq t$ are available ahead of time by precomputation (Storage required is for $t + 1$ curve points). Then, if $t + h \geq 2$, $Q = eP$ can be computed with $t + h - 2$ additions with the following algorithm shows.*

Algorithm (Input: $\{s_0 P, s_1 P, \cdots, s_t P\}$, $e = \sum_{i=0}^{t} e_i s_i$,and h; Output: $Q = eP$)

1. $B \leftarrow \mathcal{O}$ (Point at infinity)
2. $Q \leftarrow \mathcal{O}$ (Point at infinity)
3. For $j = h$ to 1 by -1
   3.1 For each i for which $e_i = j$ do the following
   $$B \leftarrow B + s_i P$$
   3.2 $Q \leftarrow B + Q$
4. Return(Q)

The most obvious application of Theorem 7 is the case in which the multiplier e is represented in radix b. Then, if $e = \sum_{i=0}^{t} e_i b^i$, $P_i = b^i P$ for $0 \leq i \leq t$ are precomputed. In addition, if we assume that e will be uniformly distributed on the range $\{0, \cdots, N\}$, then $t+1 \leq \lceil \log_b N \rceil$, we will expect on average that a digit (in the b-radix representation of e) will be zero about $1/b$ of the time and thus, the average number of additions will be $\frac{b-1}{b} \lceil \log_b N \rceil + b - 3$ (Notice $h = b - 1$).

Chapter 5

Fast Multiplication in Composite Galois Fields $GF((2^n)^m)$

With respect to complexity, field multiplication is the second most costly operation in EC systems only after inversion. Since the new point multiplication algorithm from Section 7.1 trades field inversions for field multiplications, it is especially attractive to provide efficient multiplication algorithms. In this section we apply the Karatsuba-Ofman Algorithm (KOA) [Knu81, KO63] to polynomials over Galois fields $GF(2^n)$ of degree $m-1$ which represent a field element in $GF((2^n)^m)$. First, we consider the general KOA as it is applied to two polynomials $A(x)$ and $B(x)$ with maximum degree $m-1$ over the field $\mathcal{F}$. We present the results for the case $m = 2^t$ as described in [Paa96]. In addition, we derive new formulas for the multiplicative and additive complexity for the cases $m = 2^t l$ and $m = 2^t l - 1$. Finally, we define two new operations, table look-up (TLU) and exponent addition (EXPA), and derive their complexities for the three cases. These two operations are of central importance for an exact complexity analysis of a software implementation of the KOA in composite fields.

5.1 The KOA for Polynomials of Degree $2^t - 1$

Recall from Section 3.2.3 that field multiplication in $GF((2^n)^m)$ consists of polynomial multiplication and modulo reduction, where polynomial multiplication is by far the most costly operation. We are interested in finding the product of two polynomials $A(x)$ and $B(x)$ with maximum degree of $2^t - 1$ over a field $\mathcal{F}$. Note that straightforward polynomial multiplication requires m^2 coefficient multiplications. Thus, each polynomial possesses at most $2^t = m$, t integer, coefficients and we want to find $C(x) = A(x)B(x)$ such that $\deg(C(x)) \leq 2m - 2$. Then by splitting $A(x)$ and $B(x)$ into an upper and lower half, we can apply the KOA as follows:

$$\begin{aligned} A(x) &= x^{\frac{m}{2}}(a_{m-1}x^{\frac{m}{2}-1} + \cdots + a_{\frac{m}{2}}) + (a_{\frac{m}{2}-1}x^{\frac{m}{2}-1} + \cdots + a_0) = A_h x^{\frac{m}{2}} + A_l \\ B(x) &= x^{\frac{m}{2}}(b_{m-1}x^{\frac{m}{2}-1} + \cdots + b_{\frac{m}{2}}) + (b_{\frac{m}{2}-1}x^{\frac{m}{2}-1} + \cdots + b_0) = B_h x^{\frac{m}{2}} + B_l \end{aligned} \tag{5.1}$$

Using (5.1), a set of auxiliary polynomials $D_j^{(i)}(x)$ (for $i = 1 \ldots t$ and $j = 0 \ldots (3^i - 1)$) can be defined as follows:

$$\begin{aligned} D_0^{(1)}(x) &= A_l(x)B_l(x) \\ D_1^{(1)}(x) &= (A_h(x) + A_l(x))(B_h(x) + B_l(x)) \\ D_2^{(1)}(x) &= A_h(x)B_h(x) \end{aligned} \tag{5.2}$$

Then, $C(x) = A(x)B(x)$ is obtained by:

$$C(x) = D_0^{(1)}(x) + (D_1^{(1)}(x) - D_0^{(1)}(x) - D_2^{(1)}(x))x^{\frac{m}{2}} + D_2^{(1)}(x)x^m \tag{5.3}$$

Since (5.2) requires three multiplications of polynomials with $m/2$ coefficients, the number of multiplications has been reduced to $3/4m^2$. However, the algorithm can be applied recursively to the three polynomial multiplications in (5.2). The polynomials

A_h, A_l, and $(A_h + A_l)$ as well as their B counterparts are again split in half yielding polynomials $D_j^{(2)}(x)$ of degree $m/2^2 - 1$. In the ith iteration $\deg(D_j^{(i)}(x)) = m/2^i - 1$ and after $t = \log_2 m$ steps the algorithm ends with polynomials $D_j^{(t)}(x)$ of degree 0. The following theorem derived in [Paa96] provides expressions for the computational complexity of the KOA for polynomials over fields of characteristic 2. Notice that MUL denotes the number of coefficient multiplications and ADD denotes the number of coefficient additions in $GF(2^n)$.

Theorem 8 *Consider two arbitrary polynomials in one variable of degree less than or equal to $m-1$ where $m = 2^t$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\#MUL = m^{\log_2 3} \quad (5.4)$$

$$\#ADD \leq 6m^{\log_2 3} - 8m + 2 \quad (5.5)$$

It should be noted that since we assume a field $\mathcal{F}$ with characteristic 2, the subtractions in (5.3) become additions.

5.2 Complexity of the KOA for Polynomials of Degree $2^t l - 1$

In this section we generalize the KOA from above. In the following, $\gcd(l, 2) = 1$. We consider the product of two polynomials $A(x)$ and $B(x)$ with maximum degree of $2^t l - 1$ over a field $\mathcal{F}$. In particular, we want to find $C(x) = A(x)B(x)$ such that $\deg(C(x)) \leq 2m - 2$ with $m = 2^t l$, t an integer. Notice that since the polynomial has an even number of coefficients m, the general KOA can still be applied and (5.1) and (5.2) still hold. However, since the algorithm can only run for t iterations (as many

powers of 2 as there are in m), we get in the final step that $\deg(D_j^{(t)}(x)) = l-1$. Polynomials $D_j^{(t)}$ are then multiplied using the school book method.

Theorem 9 *Consider two arbitrary polynomials in one variable of degree less than or equal to $m-1$ where $m = 2^t l$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\#MUL = l^2 \left(\frac{m}{l}\right)^{\log_2 3} = l^{2-\log_2 3} m^{\log_2 3} \tag{5.6}$$

$$\#ADD = (l-1)^2 \left(\frac{m}{l}\right)^{\log_2 3} + (8l-2)\left(\frac{m}{l}\right)^{\log_2 3} - 8m + 2 \tag{5.7}$$

Proof. We will consider three different stages in the algorithm. First, we consider the number of additions due to the splitting in (5.2). Taking into account that the number of polynomials triples after each iteration step, and noticing that the KAO runs t times, we find:

$$\#ADD_1 = \sum_{i=1}^{t} 3^{i-1} \frac{2m}{2^i} = \frac{2m}{3}\left(2\left(\frac{3}{2}\right)^{t+1} - 3\right) = 2l\left(\frac{m}{l}\right)^{\log_2 3} - 2m$$

Second, we consider the product of the $D_j^{(t)}(x)$ polynomials (all of which are of degree $l-1$). Since we can not split them anymore, the school book method is applied to each of the $D_j^{(t)}(x)$ polynomials yielding l^2 multiplications and $(l-1)^2$ additions per product. Noticing again that the number of polynomials triples with each iteration step and that KAO has run through t iterations, it is easy to see that multiplying $3^t = (m/l)^{\log_2 3}$ polynomials requires:

$$\#MUL_1 = l^2 \left(\frac{m}{l}\right)^{\log_2 3}$$

$$\#ADD_2 = (l-1)^2 \left(\frac{m}{l}\right)^{\log_2 3}$$

The third and final stage corresponds to merging the polynomials according to (5.3). It is important to point out that there are two kinds of additions (or subtractions) involved. First, the additions due to subtracting three polynomials with $2^i l - 1$ coefficients and second, the additions due to the overlapping of terms. In the first case, one should take into account that there are 2 subtractions per coefficient and that the number of polynomials triples with each

iteration, one readily obtains:

$$\#ADD_{3a} = \sum_{i=1}^{t} 3^{t-i} 2(2^i l - 1)$$

In the second case, we notice that $\deg(D_j^{(t-i+1)}(x)) = 2^i l - 2$ (again for $i = 1 \dots t$ and $j = 0 \dots (3^i - 1)$). Since the degree of the polynomials corresponding to a given iteration is always the same, we can pick any j. Thus, we consider the case for j=0. Then:

$$D_0^{(t-i)} = D_0^{(t-i+1)}(x) + (D_1^{(t-i+1)}(x) - D_0^{(t-i+1)}(x) - D_2^{(t-i+1)}(x))x^{2^{i-1}l} + D_2^{(t-i+1)}(x)x^{2^i l}$$

From this equation is easy to get the overlap by:

$$\#ADD_{overlap3terms} = (2^i l - 2 - 2^{i-1} l + 1) + (2^{i-1} l + 2^i l - 2 - 2^i l + 1) = 2^i l - 2$$

Since the number of polynomials triples with each iteration, we find:

$$\#ADD_{3b} = \sum_{i=1}^{t} 3^{t-i} (2^i l - 2)$$

Combining the additions of this last step one gets that the total number of additions due to the merging of the polynomials according to (5.3) is

$$\#ADD_3 = \sum_{i=1}^{t} 3^{t-i} (2(2^i l - 1) + (2^i l - 2)) = (6l - 2) \left(\frac{m}{l}\right)^{\log_2 3} - 6m + 2$$

Then, the overall complexities in Theorem 2 are obtained by summation of the partial complexities. This ends the proof. □

Notice that (5.6) and (5.7) reduce to (5.4) and (5.5) for $l = 1$.

5.3 Complexity of the KOA for Polynomials of Degree $2^t l - 2$

In the previous sections, we covered the cases for $m = 2^t$ and $m = 2^t l$. However, in applications of elliptic curve systems we are often interested in composite fields over the Galois field $GF((2^n)^m)$ where n and m are relatively prime as described in

Section 4.2. Since it is often desired to have n even there is a need to consider the case for odd m, i.e., $m = 2^t l - 1$. In particular, we want to find $C(x) = A(x)B(x)$ such that $\deg(C(x)) \leq 2m-2$ with $m = 2^t l - 1$, t an integer. Then, we can represent $A(x)$ and $B(x)$ in a way similar to (5.1) by adding an extra term with coefficient $a_m = 0$. For convenience, we will introduce the parameter $k = m + 1$ and express $A(x)$ and $B(x)$ as follows:

$$\begin{aligned} A(x) &= x^{\frac{k}{2}}(0x^{\frac{k}{2}-1} + a_{k-2}x^{\frac{k}{2}-2} + \cdots + a_{\frac{k}{2}}) + (a_{\frac{k}{2}-1}x^{\frac{k}{2}-1} + \cdots + a_0) = A_h x^{\frac{k}{2}} + A_l \\ B(x) &= x^{\frac{k}{2}}(0x^{\frac{k}{2}-1} + b_{k-2}x^{\frac{k}{2}-2} + \cdots + b_{\frac{k}{2}}) + (b_{\frac{k}{2}-1}x^{\frac{k}{2}-1} + \cdots + b_0) = B_h x^{\frac{k}{2}} + B_l \end{aligned} \tag{5.8}$$

Notice that now the polynomials $A(x)$ and $B(x)$ have an even number of coefficients ($k = m+1 = 2^t l$), allowing us to apply the general KOA to (5.8) t times. This reduces this problem to the case for $m = 2^t l$, permitting us to apply the same equations. However, since we have one less coefficient the final multiplicative and additive complexities are reduced. Theorem 3 summarizes the results.

Theorem 10 *Consider two arbitrary polynomials in one variable of degree less than or equal to $m-1$ where $m = 2^t l - 1$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\#MUL = l^2 \left(\frac{m+1}{l}\right)^{\log_2 3} - 2l + 1 \tag{5.9}$$

$$\begin{aligned} \#ADD &= (l-1)^2 \left(\frac{m+1}{l}\right)^{\log_2 3} \\ &\quad + (8l-2)\left(\frac{m+1}{l}\right)^{\log_2 3} - 8(m+1) + (5 - 2l - 4t) \end{aligned} \tag{5.10}$$

Proof. In order to prove Theorem 3, we will again consider three different

stages of the KOA. In the first stage, we consider the number of additions due to the splitting in (5.2). In this case, the number of polynomials triples after each iteration, thus the total number of polynomials is the same as in the case $m = 2^t l$. However, in each iteration the total number of additions is decreased by two since the total number of "real coefficients" in $A(x)$ or $B(x)$ is at most $2^t l - 1$ as opposed to $2^t l$. Therefore, we find:

$$\#ADD_1 = \sum_{i=1}^{t} \left(3^{i-1} \frac{2k}{2^i} \right) - 2 = 2l \left(\frac{m+1}{l} \right)^{\log_2 3} - 2(m+1) - 2t$$

Second, we consider the product of the $D_j^{(t)}(x)$ polynomials (notice that as in the previous cases, we will refer to the polynomials in each iteration as $D_j^{(i)}(x)$ with $j = 0 \ldots 3^i - 1$) all of which have l coefficients except for the one corresponding to $j = 3^t - 1$ which only has $l - 1$ coefficients. As before, we apply the school book method to find the product yielding $(l-1)^2$ multiplications and $(l-2)^2$ additions for $D_{3^t-1}^{(t)}(x)$ and l^2 multiplications and $(l-1)^2$ additions for the rest. Thus:

$$\begin{aligned}
\#MUL_1 &= 2\, l^2\, 3^{t-1} + (3^{t-1} - 1)l^2 + (l-1)^2 = 3^t l^2 - 2l + 1 \\
\#ADD_2 &= 2\, (l-1)^2\, 3^{t-1} + (3^{t-1} - 1)(l-1)^2 + (l-2)^2 = 3^t (l-1)^2 - 2l + 3
\end{aligned}$$

Third, we consider the merging of the polynomials according to (5.3). It is important to point out that, as before, there are two kinds of additions (or subtractions) involved (notice that since we restrict our discussion to fields of characteristic 2 subtractions and additions are equivalent). First, subtracting three polynomials with either $k/2^{i-1} - 1$ or $k/2^{i-1} - 3$ coefficients and second, the additions due to the overlapping of terms. In the first case, one should take into account that there are two subtractions per coefficient, that the number of polynomials triples with each iteration, and that in each iteration there is only one polynomial that has two coefficients less than the rest. Combining these observations, one obtains:

$$\#ADD_{3a} = \sum_{i=1}^{t} 2\, 3^{t-i} (2^i l - 1) - 2$$

Second, $2^i l - 2$ additions due to the overlapping of the three terms. It turns out that the number of overlaps is the same as in the case $m = 2^t l$. However, one should be careful of the values of l for which this assertion holds. As we know, the only polynomial that is shorter than those for the case $m = 2^t l$ are the polynomials $D_{3^i-1}^{(i)}(x)$, so these are the ones we should consider to see if there are any changes with respect to the previous cases. So we have that in

general:

$$D^{(t-i)}_{3^{t-i}-1} = D^{(t-i+1)}_{3^{t-i+1}-3} + \left(D^{(t-i+1)}_{3^{t-i+1}-2} - D^{(t-i+1)}_{3^{t-i+1}-3} - D^{(t-i+1)}_{3^{t-i+1}-1}\right) x^{\frac{k}{2^{t-i+1}}} + D^{(t-i+1)}_{3^{t-i+1}-1} x^{\frac{k}{2^{t-i}}}$$

Noticing that

$$\begin{aligned} \deg(D^{(t-i+1)}_{3^{t-i+1}-3}) &= \deg(D^{(t-i+1)}_{3^{t-i+1}-2}) = \frac{k}{2^{t-i}} - 2 \\ \deg(D^{(t-i+1)}_{3^{t-i+1}-1}) &= \frac{k}{2^{t-i}} - 4 \end{aligned}$$

We readily get that the following condition should hold:

$$\frac{k}{2^{t-i}} - 2 + \frac{k}{2^{t-i+1}} \leq \frac{k}{2^{t-i}} - 4 + \frac{k}{2^{t-i}}$$

Using the fact that $k = 2^t l$ and rearranging terms, we find:

$$2^{i-1}(2l-1) \geq 2$$

Since i goes from 1 to t, it is sufficient to find the values of l for which the following inequality is true:

$$(2l-1) \geq 2$$

Solving for l, we find that $l \geq 1.5$. But l has to be an odd number greater than 1 (from the definition of m),thus, the number of additions due to overlapping is

$$\#ADD_{3b} = \sum_{i=1}^{t} 2\, 3^{t-i}(2^i l - 2)$$

for any value of l which is consistent with our definition of m. Combining the partial results we obtain (5.9) and (5.10). This completes our proof. □

5.4 Complexity Analysis for Software Implementations

It was explained in Section 3.2.1 that one of the advantages of using composite fields was that both multiplication and inversion in $GF((2^n)^m)$ could be reduced to inversion and multiplication in the ground field which are realized through table look-ups. In this section we will apply this concept to obtain exact counts of the number of

operations needed to perform a multiplication in the field $GF((2^n)^m)$. Notice that here we use the fact that all non-zero elements $a_i \in GF(2^n)$ form a cyclic group. As stated in Section 3.2.1, these elements can be expressed as multiples of a primitive element ω: $a_i = \omega^i$.

Then, we store all the pairs (a_i, i) in two tables: log and antilog, sorted by the first component (a_i) and second component (i), respectively. Thus, the product of two elements $a_j, a_k \in GF(2^n)$ can be obtained as shown in (3.4). Notice that (3.4) implies that two elements of the ground field $GF(2^n)$ can be multiplied using three table look-up operations and one addition modulo the order of the multiplicative group (exponent addition). It is important to point out that depending on the hardware platform (e.g., microprocessor, RISC, etc.) the relative speed for the two types of operations can differ dramatically. For instance, in our implementation where $n = 16$ it was found that access to the large look-up tables took about 6 clock cycles on a DEC Alpha workstation, whereas element addition and exponent addition took about 2 clock cycles on average. Thus, in order to obtain valid performance predictions one needs exact counts of the number of operations.

Based on these two new operations (table look up (TLU) and exponent addition (EXPA)) we have derived new formulas and re-written Theorems 1, 2, and 3 as follows.

Corollary 1 *Consider two arbitrary polynomials $A(x), B(x)$ in one variable of degree less than or equal to $m-1$ where $m = 2^t$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\#ADD = 6m^{\log_2 3} - 8m + 2 \quad (5.11)$$

$$\#TLU = 3m^{\log_2 3} \quad (5.12)$$

$$\#EXPA \quad = \quad m^{\log_2 3} \tag{5.13}$$

Proof. The number of additions follows directly from Theorem 1. The number of exponent additions is the same as the number of multiplications since for each multiplication one has to perform one exponent addition, so this also follows from Theorem 1. The TLU's can be divided into three kinds. First, the TLU's needed to convert each element α_i to its corresponding exponent representation i. These are just twice the number of coefficients in one of the polynomials being multiplied or $2m = 2^{t+1}$. Second, we consider the number of TLU's required to convert back to vector representation once two elements have been multiplied. These are just equal to the number of multiplications or $m^{\log_2 3}$. Finally, we consider the number of TLU's due to splitting of polynomials in (5.2). From [Paa96] proof of the complexity of the KOA for the polynomials of degree $2^t - 1$, we find that the number of additions due to the splitting is:

$$\#ADD_{splitting} = 2m^{\log_2 3} - 2m = 2(3^t - m)$$

But each time you perform an addition due to the splitting, you create a new coefficient that needs to be transform to exponent representation, one concludes that the number of additions due to the splitting and the number of TLU's due to the splitting are the same. Thus, adding up the partial results one obtains (5.12). This ends the proof. □

Corollary 2 *Consider two arbitrary polynomials in one variable of degree less than or equal to $m-1$ where $m = 2^t l$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\#ADD \quad = \quad (l-1)^2 \left(\frac{m}{l}\right)^{\log_2 3} + (8l-2)\left(\frac{m}{l}\right)^{\log_2 3} - 8m + 2 \tag{5.14}$$

$$\#TLU \quad = \quad l\left(\frac{m}{l}\right)^{\log_2 3}(l+2) \tag{5.15}$$

$$\#EXPA \quad = \quad l^2\left(\frac{m}{l}\right)^{\log_2 3} \tag{5.16}$$

Proof. As in Corollary 1, both the number of additions and exponent additions follow from Theorem 2. The number of TLU's can be found by adding the TLU's needed to convert each element α_i to its corresponding exponent

representation i or $2m = 2^{t+1}l$, the number of TLU's required to convert back to vector representation once two elements have been multiplied which is the same as the number of multiplications, and the number of TLU's due to the splitting of polynomials in (5.2) which as in the case of Corollary 1 is equal to the number of additions due to the splitting or $2l\left(\frac{m}{l}\right)^{\log_2 3} - 2m$. Adding up the partial results we get (5.15). □

Corollary 3 *Consider two arbitrary polynomials in one variable of degree less than or equal to $m-1$ where $m = 2^t l - 1$, with coefficients in a field $\mathcal{F}$ of characteristic 2. Then, by using the Karatsuba-Ofman algorithm the polynomials can be multiplied with:*

$$\begin{aligned}
\#ADD &= (l-1)^2\left(\frac{m+1}{l}\right)^{\log_2 3} \\
&\quad +(8l-2)\left(\frac{m+1}{l}\right)^{\log_2 3} - 8(m+1) + (5-2l-4t) &(5.17)\\
\#TLU &= \left(\frac{m+1}{l}\right)^{\log_2 3} l(l+2) - 2(t+l) &(5.18)\\
\#EXPA &= l^2\left(\frac{m+1}{l}\right)^{\log_2 3} - 2l + 1 &(5.19)
\end{aligned}$$

Proof. (5.17) and (5.19) follow directly from Theorem 3. The number of TLU's is obtained from the number of TLU's needed to convert each element α_i to its corresponding exponent representation i or $2m = 2^{t+1}l - 2$, the the number of TLU's required to convert back to vector representation once two elements have been multiplied which is the same as the number of multiplications, and the number of TLU's due to the splitting of polynomials in (5.2) which is equal to the number of additions due to the splitting or $2l\left(\frac{m+1}{l}\right)^{\log_2 3} - 2(m+1) - 2t$. Adding up the partial results one finds (5.18). □

5.5 Multiplication in $GF((2^{16})^{11})$

We summarize this section by considering the complexity of a multiplication in $GF((2^n)^m)$ for $n = 16$ and $m = 11$. We can apply Corollary 3 and let $m = 11$,

$l = 3$, and $t = 2$. From there we obtain that one needs 129 additions and 76 multiplications in $GF(2^{16})$ or equivalently, 129 coefficient additions, 125 table look-ups, and 76 exponent additions. Notice, when using the regular method for multiplication one would require 121 multiplications and 100 coefficient additions or, in terms of table look-ups, 100 coefficient additions, $121 + 22 = 143$ table look-ups, and 121 exponent additions. If one compares both complexities and ignores exponent and coefficient additions, one can readily see that the theoretical improvement in the timing for the multiplication operation when using the Karatsuba-Ofman algorithm would be about 12.5 percent.

Method	Average Timing (μsec)
Straight forward method	43.0
Karatsuba-Ofman algorithm	38.6

Table 5.1: Comparison of timings for 176-bit multiplication in $GF((2^{16})^{11})$ using the Karatsuba-Ofman algorithm and the straight forward multiplication method.

Table 5.1 compares the timings obtained from the implementation of both multiplication algorithms (KOA and regular multiplication). It can be seen from the table that the improvement in timing for one multiplication is about 10.5 percent which agrees with the theoretical predictions.

Chapter 6

Efficient Inversion in Composite Galois Fields $GF((2^n)^m)$

As stated in the previous sections, inversion is the most costly arithmetic operation in EC systems. In the following an inversion method based on Fermat's little theorem will be developed which is entirely different from the approach in [WBV+96, SOOS95]. The basic property of the algorithm developed in this section is that inversion in $GF((2^n)^m)$ is reduced to inversion in the subfield $GF(2^n)$. It is important to point out that subfield inversion can be done extremely fast through table look-up provided n is moderate, say $n \leq 16$. Notice also that the Itoh and Tsujii's Algorithm introduced in [IT88] was originally applied to composite fields $GF((2^n)^m)$ represented in normal bases. However, we applied and optimized this algorithm to composite fields in standard base representation, as suggested in [Paa95]. Finally, we show a major computational advantage for the case that the field polynomial has only coefficients from $GF(2)$.

We want to determine the inverse of $A \in GF((2^n)^m)$, $A \neq 0$. A is given as

$$A(x) = a_{m-1}x^{m-1} + \cdots + a_1 x + a_0, \quad a_i \in GF(2^n). \tag{6.1}$$

By applying Fermat's Theorem, we can readily obtain that

$$A^{2^{nm}-1} = AA^{2^{nm}-2} = 1, \quad \forall A \in GF((2^n)^m) \setminus \{0\}, \tag{6.2}$$

from which it follows that

$$A^{-1} = A^{2^{nm}-2}. \tag{6.3}$$

(6.3) shows that the inverse of an element $A \in GF((2^n)^m)$ can be computed by raising it to the power of $2^{nm} - 2 = 2 + 2^2 + 2^3 + \cdots + 2^{nm-1}$ using the standard "binary method" described in [Knu81]. However, in the following we derive a method which reduces inversion in the composite field $GF((2^n)^m)$ can be reduced to inversion in the ground field $GF(2^n)$, one obtains a better method to calculate the inverse of an element A. The following theorem describes the algorithm.

Theorem 11 *[Paa94] The multiplicative inverse of an element A of the composite field $GF((2^n)^m)$ can be computed by*

$$A^{-1} = (A^r)^{-1} A^{r-1}, \tag{6.4}$$

where $A^r \in GF(2^n)$ and $r = (2^{nm} - 1)/(2^n - 1)$.

Computing the inverse through Theorem (11) requires four steps: exponentiation in $GF((2^n)^m)$ (A^{r-1}), multiplication of A and A^{r-1} to get $A^r \in GF(2^n)$, inversion in $GF(2^n)$, and multiplication of $(A^r)^{-1}A^{r-1}$. Each of these steps will be analyzed in the following.

6.1 Exponentiation in $GF((2^n)^m)$

The first step in the algorithm above is the computation of A^{r-1} where $A \in GF((2^n)^m)$. Notice that r can be expressed as a sum of powers as follows:

$$r - 1 = \frac{2^{nm} - 1}{2^n - 1} - 1 = 2^n + 2^{2n} + 2^{3n} + \cdots + 2^{(m-1)n} \tag{6.5}$$

This representation is similar to the binary representation of the number $2^{nm} - 2 = 2 + 2^2 + 2^3 + \cdots + 2^{nm-1}$ and hence, the optimized method from [IT88] can be applied. The method requires $\lfloor \log_2(m-1) \rfloor + H_w(m-1) - 1$ general multiplications and $m-1$ exponentiations to the power of 2^n [IT88], with both types of operations performed in $GF((2^n)^m)$($H_w()$ denotes the Hamming weight of the binary representation of the operand). Multiplications can be realized using the Karatsuba-Ofman Algorithm described in Chapter 5 and exponentiation is realized as explained below. Let B and C be elements of $GF((2^n)^m)$. We want to find $C(x) = B^{2^n}$, where $B(x) = \sum_{i=0}^{m-1} b_i x^i$. This can be performed as follows (the proof is based on [McE87, Lemma 5.12]):

$$C(x) = \sum_{i=0}^{m-1} c_i x^i = \left(\sum_{i=0}^{m-1} b_i x^i \right)^{2^n} = \sum_{i=0}^{m-1} b_i^{2^n} x^{i2^n} = \sum_{i=0}^{m-1} b_i x^{i2^n}, \quad b_i \in GF(2^n). \tag{6.6}$$

Assuming $2^n > m - 1$, there are $m - 1$ powers of x which must be reduced modulo the field polynomial $P(x)$, namely the powers x^{i2^n}, $i = 1, 2, \ldots, m-1$. We use the following notation for the representation of these powers in the residue classes modulo $P(x)$:

$$x^{i2^n} = s_{0,i} + s_{1,i} x + \cdots + s_{m-1,i} x^{m-1} \bmod P(x), \quad i = 1, 2, \ldots, m-1. \tag{6.7}$$

Using the coefficients $s_{j,i}$, the exponentiations in (6.6) can be expressed in matrix

form as

$$\begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{m-1} \end{pmatrix} = \begin{pmatrix} 1 & s_{0,1} & s_{0,2} & \cdots & s_{0,m-1} \\ 0 & s_{1,1} & s_{1,2} & \cdots & s_{1,m-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & s_{m-1,1} & s_{m-1,2} & \cdots & s_{m-1,m-1} \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{m-1} \end{pmatrix}. \tag{6.8}$$

A main computational advantage occurs if $P(x)$ is chosen to have only binary coefficients, as suggested in Section 3.2.3. In this case, all powers of $x^a \bmod P(x)$ belong to a subfield whose elements are represented by binary polynomials. In particular, all coefficients $s_{n,i}$ in (6.8) are binary, i.e., elements from $GF(2)$. In addition, since both n and $P(x)$ are known ahead of time one exponentiation is reduced from $(m-1)m$ constant multiplications and $m(m-2)+1$ additions to only $(m-1)^2/4$ additions on average.

6.2 Multiplication in $GF((2^n)^m)$, where the Product is an Element of $GF(2^n)$

The second step performs the operation

$$A^r = A^{r-1}A, \tag{6.9}$$

where $A^r \in GF(2^n)$, and the two operands are elements in $GF((2^n)^m)$. We consider the multiplication $D(x) = B(x)C(x) \bmod P(x)$ where $B, C \in GF((2^n)^m)$ and $D \in GF(2^n)$. First, we consider the pure polynomial multiplication of B and C:

$$D'(x) = B(x)C(x) = \left(\sum_{i=0}^{m-1} b_i x^i\right)\left(\sum_{i=0}^{m-1} c_i x^i\right) = \left(\sum_{i=0}^{m-2} d'_i x^i\right). \tag{6.10}$$

We know that $D'(x) \equiv D(x) = d_0 \bmod P(x)$, i.e., that all but the zero coefficient of $D'(x)$ vanish after reduction modulo $P(x)$. The reduction modulo $P(x)$ is done as explained in Section 3.2.3 with a reduction matrix of the form of (3.8). Using this matrix representation and the fact that $\deg(D(x)) = 0$, one finds that $D(x)$ can be expressed as:

$$D(x) = d_0 = d'_0 + \sum_{i=0}^{m-2} r_{0,i}\, d'_{m+i} \pmod{P(x)} \tag{6.11}$$

which eliminates all constant multiplications.

6.3 Inversion in $GF(2^n)$ and Multiplication of an Element from $GF(2^n)$ with an Element from $GF((2^n)^m)$

The third and fourth steps carry small complexities since both involve operations with elements of the subfield. First, we calculate the inverse of A^r with two table look-ups [WBV+96] since A^r is an element of the ground field. This inversion operation is accomplished by first finding the exponent i to which a primitive element $\omega \in GF(2^n)$ would be raised, in order to obtain A^r. This first step is done by making use of the log function described in Section 3.2.1 (first table look-up). Second, one takes the negative of this exponent modulo $2^n - 1$, and finally one converts the previous result back to the element representation with the antilog function (second table look-up). This process is shown in analytical form in (6.12).

$$A^{-r} = \text{antilog}[-\log(A^r) \pmod{2^n - 1}] \tag{6.12}$$

In the fourth step, we compute $A^{-1} = (A^r)^{-1}A^{r-1}$ by multiplying $(A^r)^{-1}$, which is also an element of $GF(2^n)$, with A^{r-1}, an element of $GF((2^n)^m)$. This last operation requires m multiplication in the ground field $GF(2^n)$. Notice that there is no reduction modulo $P(x)$ since all arithmetic is done in $GF(2^n)$.

6.4 Inversion in $GF((2^{16})^{11})$

We consider the special case where $n = 16$ and $m = 11$. In this case, we chose as field polynomial the trinomial $P(x) = x^{11} + x^2 + 1$ because it would minimize the number of non-zero entries in the matrices of (6.8) and (3.8). Using this polynomial, one can find A^{r-1} with 4 multiplications in $GF((2^{16})^{11}$ and 390 additions in $GF(2^{16})$. Similarly, $A^r = A^{r-1}A$ can be computed using 12 multiplications and 10 additions in $GF(2^{16})$, $(A^r)^{-1}$ requires one inversion in the subfield $GF(2^{16})$, and $(A^r)^{-1}A^{r-1}$ involves 11 subfield multiplications in $GF(2^{16})$. Thus, the total complexity of an inversion in $GF((2^{16})^{11})$ is 4 multiplications in the composite field and one inversion, 23 multiplications, and 400 additions in the subfield. This translates into an inversion time of 158.7 μsec. This time is essentially determined by the 4 multiplications required to perform the inversion operation.

Chapter 7

A New Approach to Point Doubling for Elliptic Curves

This chapter introduces an entirely new approach for accelerating the multiplication of points on an elliptic curve. The approach works in conjunction with the k-ary and the sliding window methods. The method is applicable to elliptic curves over any field, but we provide worked-out formulae for elliptic curves over fields of characteristic two. In addition, we show the actual performance of the newly introduced algorithm and the ones treated in Chapters 5 and 6 in an implementation of an elliptic curve cryptosystem over $GF(2^{176}) \cong GF((2^{16})^{11})$. We provide absolute timing measurements for an entire elliptic curve multiplication as well as timings for individual operations.

7.1 Principle Idea

The basic operation for the DL problem for elliptic curves is "multiplication" of a point $P \in E$ with an integer e, which is of the order of $\#E$. One way of performing this operation is analogous to the square and multiply algorithm for exponentiation [Knu81] and it is known as "repeated double and add" as described in Theorem 1. A generalization of this method is the k-ary method which is described in Theorem 4. This algorithm reduces the number of additions needed in the regular double and add algorithm.

Notice that in Theorem 4, Step 3.1 in the algorithm involves the doubling of point Q, k times, and Step 3.2 requires one point addition. Since point doubling is the most costly operation, it is extremely attractive to find ways of accelerating the doubling operation. Recall now from Section 2.2.2 that the doubling of an elliptic curve point requires one inversion and that in most practical applications, inversion is by far the most expensive operation to perform. In the following we will introduce an entirely new approach to compute repeated point doublings over an elliptic curve which reduces the number of inversion at the cost of extra multiplications and thus the complexity of the overall computation of the multiple of an elliptic curve point.

Our new approach is based on the following principle. First, observe that the k-ary method relies on k *repeated* doublings. The new approach allows computation of $2^k P = (x_k, y_k)$ directly from $P = (x, y)$ without computing the intermediate points $2P$, 2^2P, $\cdots$,$2^{k-1}P$. Such direct formulae are obtained by inserting (2.4) and (2.5) into one another. Theorem 12 describes a formula for computing two point doublings directly and its proof gives the derivation.

Theorem 12 *Given a point $P = (x, y)$ on the elliptic curve E one can compute the point $Q = 2^2P = (x_2, y_2)$ with 1 inverse, 9 multiplications, 6 squarings, and 10*

additions as shown in (7.1) and (7.2).

$$x_2 = \frac{\zeta^2 + (\delta\gamma)\zeta}{(\delta\gamma)^2} + a, \tag{7.1}$$

$$y_2 = \frac{\zeta(\delta\gamma)x_2 + (\delta^2)^2}{(\delta\gamma)^2} + x_2, \tag{7.2}$$

where $\gamma = x^2$,$\eta = \gamma + y$, $\delta = \eta^2 + \eta x + a\gamma$, $\xi = \eta x + \gamma$, *and* $\zeta = \delta(\delta + \xi) + \gamma^2\gamma$.

Proof. We begin by rewriting (2.4) and (2.5) in terms of x, y, x_1, and y_1 where $P = (x, y)$ and $2P = (x_1, y_1)$,

$$x_1 = \frac{(x^2 + y)^2 + (x^2 + y)x + ax^2}{x^2}$$

$$y_1 = \frac{x^2 x + (x^2 + y)x_1}{x} + x_1$$

Letting $\gamma = x^2$, $\eta = \gamma + y$, and $\delta = \eta^2 + \eta x + a\gamma$, we get:

$$x_1 = \frac{\delta}{\gamma}$$

$$y_1 = \frac{\gamma x + \eta x_1}{x} + x_1$$

Substituting x by x_1 and x_1 by x_2, we readily find an equation for $2^2 P = (x_2, y_2)$,

$$x_2 = \frac{(x_1^2 + y_1)^2 + (x_1^2 + y_1)x_1 + ax_1^2}{x_1^2}$$

$$y_2 = \frac{x_1^2 x_1 + (x_1^2 + y_1)x_2}{x_1} + x_2$$

Next, we consider the term $x_1^2 + y_1$. By using the expressions for x_1 and y_1 in terms of γ, η, δ, and the new variables $\xi = \eta x + \gamma$ and $\zeta = \delta^2 + \gamma^2\gamma + \delta\xi$, we get:

$$x_1^2 + y_1 = \frac{\zeta}{\gamma^2}$$

Substituting back into the expression for x_2,

$$x_2 = \frac{\left(\frac{\zeta}{\gamma^2}\right)^2 + \left(\frac{\zeta}{\gamma^2}\right)x_1}{\left(\frac{\delta}{\gamma}\right)^2} + a$$

Doing the algebra and simplifying, one readily obtains (7.1). A similar proce-

dure will yield for y_2 the following intermediate result:

$$y_2 = \frac{\frac{\zeta}{\gamma^2}x_2 + \left(\frac{\delta}{\gamma}\right)^2 \frac{\delta}{\gamma}}{\left(\frac{\delta}{\gamma}\right)} + x_2$$

Simplifying and multiplying numerator and denominator by δ one gets,

$$y_2 = \frac{\left(\frac{\zeta\gamma x_2 + \delta^2\delta}{\gamma^2}\right)\delta}{\delta^2} + x_2$$

which after simplification turns into (7.2). This ends the proof.□

We continued in a similar manner and found expressions for $2^3P = (x_3, y_3)$, $2^4P = (x_4, y_4)$, and $2^5P = (x_5, y_5)$. Again, these expressions, described in Theorems 13, 14, and 15, only require one inversion as opposed to the three, four, or five inversions that the regular double and add algorithm would require in each one of these cases. It is important to point out that the point P has to be an element of prime order belonging to the cyclic subgroup corresponding to the largest prime factor in the order of E. This last requirement ensures that $4P$, $8P$, $16P$, or $32P$ will never equal $\mathcal{O}$. Notice that this is compliant with [KMQV96].

Theorem 13 *Given a point $P = (x, y)$ on the elliptic curve E one can compute the point $Q = 2^3P = (x_3, y_3)$ with 1 inverse, 14 multiplications, 7 squarings, and 17 additions as shown in (7.3) and (7.4).*

$$x_3 = \frac{\omega^2 + \omega\rho}{\rho^2} + a \tag{7.3}$$

$$y_3 = \frac{(v^2)^2 + \omega\rho x_3}{\rho^2} + x_3 \tag{7.4}$$

Theorem 14 *Given a point $P = (x, y)$ on the elliptic curve E one can compute the point $Q = 2^4P = (x_4, y_4)$ with 1 inverse, 19 multiplications, 15 squarings, and 20 additions as shown in (7.5) and (7.6).*

$$x_4 = \frac{\theta^2 + \theta\mu\rho^2}{(\mu\rho^2)^2} + a \tag{7.5}$$

$$y_4 = \frac{(\mu^2)^2 + (\theta\mu\rho^2)x_4}{(\mu\rho^2)^2} + x_4 \quad (7.6)$$

Theorem 15 *Given a point $P = (x, y)$ on the elliptic curve E one can compute the point $Q = 2^5P = (x_5, y_5)$ with 1 inverse, 24 multiplications, 19 squarings, and 22 additions as shown in (7.7) and (7.8).*

$$x_5 = \frac{\sigma^2 + \sigma\beta\kappa^2}{(\beta\kappa^2)^2} + a \quad (7.7)$$

$$y_5 = \frac{(\beta^2)^2 + x_5(\sigma\beta\kappa^2)}{(\beta\kappa^2)^2} + x_5 \quad (7.8)$$

where γ,η, δ, ξ, and ζ are as defined in Theorem 12, and $\tau = \delta\gamma$, $\upsilon = \zeta^2 + \tau\zeta + \tau^2 a$, $\rho = \upsilon\tau^2$, $\omega = \upsilon(\upsilon + \zeta\tau) + (\tau\delta^2)^2 + \rho$, $\mu = \omega^2 + \omega\rho + a\rho^2$, and $\theta = \mu^2 + \mu(\omega\rho) + \mu\rho^2 + (\upsilon^2\rho)^2$, $\kappa = \mu\rho^2$, $\beta = \theta^2 + \theta\kappa + a\kappa^2$, and $\sigma = \beta(\beta + \theta\kappa + \kappa^2) + \kappa^2(\mu^2)^2$.

The advantage of Equations (7.1) and (7.2) is that they only require one inversion as opposed to the two inversions that two separate double operations would require for computing $4P$. The "price" that must be paid is $9 - 4 = 5$ extra multiplications if squarings and additions are ignored. For $k = 2$, the direct formulae (7.1) and (7.2) trade thus one inversion at the cost of 5 multiplications. It is easy to see that the formulae are an advantage in situations where inversion is at least five times as costly as multiplication. However, this "break even point" decreases if the method is extended to the computation of 2^kP for $k > 2$ as described below.

7.1.1 The Break-Even Point

For application in practice it is highly relevant to compare the complexity of our newly derived formulae with that of the double and add algorithm. If we note that our method reduces inversions at the cost of multiplications, the performance of the new method depend on the cost factor of one inversion relatively to the cost of one multiplication. For this purpose we introduce the notion of a "break even point."

Since it is possible to express the time that it takes to perform one inversion in terms of the equivalent number of multiplication times, we define the break even point as the number of multiplication times needed per inversion so that our formulae outperform the regular double and add algorithm. The results are summarized in Table 7.1.

Calculation	Method	Complexity				Break Even Point
		Sq.	Add.	Mult.	Inv.	
$4P$	Direct Doublings	6	10	9	1	1 inv. > 5 mult.
	Individual Doublings	4	10	4	2	
$8P$	Direct Doublings	7	17	14	1	1 inv. > 4 mult.
	Individual Doublings	6	15	6	3	
$16P$	Direct Doublings	15	20	19	1	1 inv. > 3.7 mult.
	Individual Doublings	8	20	8	4	
$32P$	Direct Doublings	19	22	24	1	1 inv. > 3.5 mult.
	Individual Doublings	10	25	10	5	

Table 7.1: Complexity comparison: Individual doublings vs. direct computation of several doublings.

Next, the break-even point is derived for the case of 2 doublings. In general we have that for our formulae to be advantageous we need the following relation to hold:

$$Cost(\text{2 doublings}) > Cost(\text{Formula for 2 doublings})$$

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

$$2(\text{2 multiplications } +1 \text{ inverse}) > (\text{9 multiplications } +1 \text{ inverse})$$

Defining $r = M/I$, where M stands for the time that it takes for one multiplication to be performed and I is the time for one inversion, one can re-write the above expressions as:

$$2(2M + rM) > (9M + rM)$$

Solving for r in terms of M one obtains:

$$r > (9-4)M \Rightarrow r > 5M$$

7.1.2 Theoretical and Practical Timings

We performed timing measurements on the individual doubling operation and the corresponding formulae presented in Theorems 12, 13, 14,and 15. In addition, we developed timing estimates based on the observed timings of field operations in the Galois field $GF((2^{16})^{11})$ as presented in Table 7.2 (Notice multiplication and inversion were implemented as described in Chapters 5 and 6).

Type of Operation	Average Timing (μsec)
176 bit addition	1.19
176 bit squaring	4.23
176 x 176 bit multiplication	38.56
176 bit inverse	158.73

Table 7.2: Timings for various field operations in $GF((2^{16})^{11})$.

In analyzing this results, one has to take into account the fact that in our implementation all operations were run on a DEC Alpha 3000, a 175 MHz RISC processor with a 64-bit word size and that the coordinates of the elliptic curve points that were doubled are in the field $GF((2^{16})^{11})$. Notice that one inversion time correspond to 4.12 multiplication times. Using Table 7.1, we can readily predict that the timings for the formulae presented in Theorems 13, 14 and 15 should outperform the timings for the individual doublings. In addition, using the complexity shown in Table 7.1 and the timings shown in Table 7.2 we can make estimates as to how long a doubling operation will take using both formulae and individual doublings. Table 7.3 summarizes the estimated times and compares them to the actual times

that doublings take.

Calculation	Method	Predicted Timing	Measured Timing	% Improvement	
				Predicted	Measured
$8P$	Direct Doublings	748.41 μsec	904.812 μsec	0.30	12.5
	Individual Doublings	750.78 μsec	1.035 msec		
$16P$	Direct Doublings	978.62 μsec	1.141 msec	2.24	17.85
	Individual Doublings	1.001 msec	1.389 msec		
$32P$	Direct Doublings	1.191 msec	1.380 msec	4.80	22.08
	Individual Doublings	1.251 msec	1.771 msec		

Table 7.3: Timing comparison: Individual doublings vs. direct computation of several doublings in $GF((2^{16})^{11})$.

Two important observations are worth noticing:

- First, the average timings are always greater than the estimated timings, which we attributed to the presence of overhead in the routines that implement the algorithms (e.g., initialization of variables).
- Second, and more important than the first observation, the average timings imply that the formulae outperform the regular approach of doubling elliptic curve points by much more than predicted.

7.2 Complexity Analysis of the k-ary Method

In this section, we perform an analysis of the k-ary method when it is used in conjunction with the formulae presented in Theorems 12, 13, 14, and 15. In addition, we compare the complexity of both approaches to the k-ary method, with and without formulae. Finally, we derive an expression that predicts the theoretical improvement of the k-ary method when applied with the formulae, in terms of the ratio between inversion and multiplication times.

7.2.1 k-ary Method Complexity

In Section 4.3, we noticed that the complexity of the k-ary method was:

$$\#\text{Additions}_{k-ary} = 2^k - 2 + t \tag{7.9}$$

$$\#\text{Doublings}_{k-ary} = kt \tag{7.10}$$

where $t+1$ is the number of k-bit words in the multiplier e. If we consider a $(l+1)$-bit multiplier, we can rewrite (7.9) and (7.10) as: (Notice $t = \lceil \frac{l+1}{k} \rceil - 1$)

$$\#\text{Additions}_{k-ary} = 2^k + \left\lceil \frac{l+1}{k} \right\rceil - 3 \tag{7.11}$$

$$\#\text{Doublings}_{k-ary} = k\left(\left\lceil \frac{l+1}{k} \right\rceil - 1\right) \tag{7.12}$$

Recall from Section 2.2.2 that both, point addition and doubling, require two field multiplications and one field inversion (squarings and additions will not be taken into account since they are almost for "free" when compared to multiplication and inversion). Then, the number of multiplications and inversions in the k-ary method are:

$$\#\text{Multiplications}_{k-ary} = 2(k+1)\left\lceil \frac{l+1}{k} \right\rceil + 2^{k+1} - (6+2k) \tag{7.13}$$

$$\#\text{Inversions}_{k-ary} = (k+1)\left\lceil \frac{l+1}{k} \right\rceil + 2^k - (3+k) \tag{7.14}$$

For the specific case $k = 4$,which is the optimum choice for most elliptic curve systems, (7.13) and (7.14) reduce to:

$$\#\text{Multiplications}_{k-ary}(k=4) = 10\left\lceil \frac{l+1}{4} \right\rceil + 18 \tag{7.15}$$

$$\#\text{Inversions}_{k-ary}(k=4) \quad = \quad 5\left\lceil \frac{l+1}{4} \right\rceil + 9 \tag{7.16}$$

7.2.2 Complexity of the k-ary Method with Formulae for $k = 4$

In order to apply the formulae of Theorem 14, we substitute the double and add algorithm in Step 3.1 by (7.5) and (7.6). Then, (7.11) and (7.12) become:

$$\#\text{Additions}_{k-ary} \quad = \quad 2^k + \left\lceil \frac{l+1}{k} \right\rceil - 3 \tag{7.17}$$

$$\#\text{Doubling Formulas}_{k-ary} \quad = \quad t \tag{7.18}$$

Recall from Theorem 14 that one needs 19 field multiplications and one inverse to perform four consecutive doublings with the formulae in (7.5) and (7.6). Then, noticing that the complexity of a point addition operation has not changed, we find:

$$\#\text{Multiplications}_{k-ary\ with\ formulae}(k=4) \quad = \quad 21\left\lceil \frac{l+1}{4} \right\rceil + 7 \tag{7.19}$$

$$\#\text{Inversions}_{k-ary\ with\ formulae}(k=4) \quad = \quad 2\left\lceil \frac{l+1}{4} \right\rceil + 12 \tag{7.20}$$

7.2.3 Relative Improvement

In this section, we consider the special case in which $l + 1 = 176$, or in other words, the case of a 176-bit multiplier. Notice that, $k = 4$ in all computations because it was found to be the optimum value of k for this number of bits. The number of multiplications and inversions needed for the k-ary method with and without formulas are summarized in Table 7.4.

From Table 7.4 we derived expressions for the time it would take to perform

Field Operation	k-ary method	k-ary method with formulae
#multiplications	458	931
#inversions	229	100

Table 7.4: Comparison of complexities required to perform the multiplication eP using the regular k-ary method, $k = 4$, and the k-ary method with four direct doublings.

a whole point multiplication as:

$$\mathrm{T}_{Regular\ method} = 458t_{MULT} + 229t_{INV} \tag{7.21}$$

$$\mathrm{T}_{Formula\ method} = 931t_{MULT} + 100t_{INV} \tag{7.22}$$

where t_{MULT} means the time required for one field multiplication and t_{INV} signifies the time required for one field inversion. Notice that from (7.21) and (7.22) one can readily derive the relative improvement by defining $r = t_{INV}/t_{MULT}$ as:

$$\text{Relative Improvement} = \frac{\mathrm{T}_{Regular\ method} - \mathrm{T}_{Formula\ method}}{\mathrm{T}_{Regular\ method}} \tag{7.23}$$

or using (7.21) and (7.22)

$$\text{Relative Improvement} = \frac{129r - 473}{229r + 458} < 56.3\% \tag{7.24}$$

Notice that the relative improvement for the ratio of the inversion time to the multiplication time in our implementation is:

$$\text{Relative Improvement}(r = 4.19) = \frac{129(4.19) - 473}{229(4.19) + 458} \times 100 = 13.5\%$$

As will be shown in Section 7.4 the actual improvement is 17 percent which is in accordance with the observation in Table 7.3.

7.3 Complexity Analysis of the Improved k-ary Method with Formulae for $k = 5$

In this section, we consider the improved k-ary method of exponentiation described in Theorem 5. We develop a complexity analysis of this algorithm for the case $k = 5$ and find specific values for a 176-bit multiplier.

By choosing $k = 5$, we limit the size of each word e_i in the b-radix representation of the multiplier to 5 bits (see Theorem 5). This implies that there are 32 possible different values that each 5-bit word e_i can attain. Furthermore, if one was to divide the e_i words into subgroups based on the value of h_i and $5 - h_i$ (as defined in Theorem 5) one would find the distribution found in Table 7.5. Notice that the values of h_i will determine the number of doublings performed in a given iteration of the improved k-ary method for exponentiation.

h_i	$5 - h_i$	Frequency
0	5	17
1	4	8
2	3	4
3	2	2
4	1	1

Table 7.5: Frequency of occurrence for possible h_i and $5 - h_i$ values.

As in Section 7.2, we will limit our complexity analysis to multiplications and inversions since additions and squarings are "cheap" when compared to the other field operations. Notice, then that depending on the value of h_i, one will have to perform five doublings (D_5), four doublings (D_4), three doublings (D_3), two doublings (D_2), or one doubling (D_1). Table 7.6 shows a summary of the complexity of these operations. It is important to point out that since the formulae presented in Theorems 12, 13, 14, and 15 are only more efficient than the double and add algorithm for three, four, or

five doublings, these are the only formulae that will be used. The standard doubling formula for elliptic curves from (2.4) and (2.5) will be used in the remainder of the cases.

Method of Doubling	# of Doublings	# Mult.	# Inv.
Regular Method	D_1	2	1
	D_2	4	2
Formulae Method	D_3	14	1
	D_4	19	1
	D_5	24	1

Table 7.6: Complexity of Doubling Approach.

From Theorem 5, we know that there $2^4 - 1$ additions and one doubling due to precomputation, and t additions, $t(h_i + (5 - h_i))$ doubling steps (why we write the number of doublings in this way rather than $5t$ will become apparent later) for iterations $t - 1$ to 0, and a variable number of operations in the first iteration of the algorithm, depending on the number of significant bits in the most significant word of the multiplier.

Now, we specialize our analysis to the case in which we have a $l + 1 = 176$ bit multiplier. In this case, the algorithm will go through $t + 1 = \lceil (l + 1)/5 \rceil = 36$ iterations. Then, we can find the complexity of the precomputations as follows: (Notice that as before one point addition takes two multiplications and one inversion)

$$\begin{aligned}\#\text{Multiplications}_{precom} &= 2 + 15(2) = 32 \\ \#\text{Inversions}_{precom} &= 1 + 15 = 16\end{aligned}$$

The complexity of the precomputation is found by noticing that the most significant word of the base-32 representation of the multiplier can have at most two bits (in general there will be at most $2^{\lceil (l+1)/k \rceil k - (l+1)}$ bits). Therefore, we have the complexity shown in Table 7.7.

Bit Pattern	Doublings	Table Look-ups
00	0	0 (go to next word)
01	0	1
10	1	1
11	0	1

Table 7.7: Number of Operation for Bit Patterns of the Most Significant Word of the Multiplier.

Since table look-ups are essentially for free and assuming that every bit pattern has the same probability, we will have that an average of $0.25 \cdot 2 = 0.5$ multiplications and $0.25 \cdot 1 = 0.25$ inversions from the first iteration of the algorithm. For the remaining t iterations we will have the following average number of operations: (Notice that D_1, D_2, D_2, D_3, D_4, and D_5 are as defined in Table 7.6)

$$\begin{aligned}\text{Avg. \# operations} &= \Big(\frac{17}{32}D_5 + \frac{1}{32}(D_4 + D_1) + \frac{2}{32}(D_3 + D_4) + \big) \\ &\quad (\frac{4}{32}(D_2 + D_3) + \frac{8}{32}(D_1 + D_4)\Big)\, t\end{aligned}$$

Substituting the corresponding number of multiplications and inversions in for D_1, D_2, D_2, D_3, D_4, and D_5, we get:

$$\begin{aligned}\#\text{Multiplications}_{doubling} &= \frac{24675}{32} = 771.09 \\ \#\text{Inversions}_{doubling} &= \frac{1855}{32} = 57.97\end{aligned}$$

The number of additions is just equal to t or 35 additions. Notice that there is a 1/32 chance that $e_i = 0$, in which case there would not be an addition, thus in terms of the field operations we have that 35 additions contribute:

$$\#\text{Multiplications}_{additions} = 2\frac{31}{32}35 = 67.81$$

$$\#\text{Inversions}_{additions} = \frac{31}{32}35 = 33.91$$

Thus, adding up the partial complexities, we find that the improved k-ary method for exponentiation with a 176-bit multiplier and $k = 5$, will require on average:

$$\#\text{Multiplications}_{improved\ k-ary} = 871.41 \quad (7.25)$$

$$\#\text{Inversions}_{improved\ k-ary} = 108.13 \quad (7.26)$$

Comparing (7.25) and (7.26) to the results presented in Table 7.4, it can be seen that the number of multiplications decreases by about 60 at the cost of about 8 extra inversions. Assuming that one inversion time is 4.19 multiplication times, one finds that the improvement from the k-ary method with formulae to the improved k-ary method with formulae is on average about 25.55 multiplication times or on a DEC Alpha with a 175 MHz clock frequency, 985 μsec.

7.4 Application to Point Multiplication

This section describes the application of the various algorithms to an actual EC system over the field $GF(2^{16})^{11}) \cong GF(2^{176})$. First we investigate multiplication of a point which is the core operation in a Diffie-Hellman key exchange or a digital signature generation. We compare the timings obtained for different parameters k in the k-ary method and the improved k-ary method with and without the new formulae of Section 7.1. We also present the timings for several arithmetic operations in the composite field $GF((2^{16})^{11})$ and the timings for several algorithms used to compute nP where n is a long integer (176 bits) and P is a point on the elliptic curve as described in (2.3). The DEC Alpha 3000, a 175 MHz RISC architecture with a 64 bit word size was used to perform all measurements. In some cases timings on a

DEC Alpha with a clock frequency of 233 MHz will also be provided as a measure of comparison.

Method	Window Size k	Average Timing (in msec)
k-ary	3	87
	4	84
	5	88
k-ary with formulae	4	68

Table 7.8: Comparison of average time required to perform the nP calculation in $GF((2^{16})^{11})$ using the regular k-ary method and the k-ary method with four direct doublings using a DEC Alpha with 175 MHz clock frequency.

After several measurements, it was found that the optimum value for the window size in the k-ary method was $k = 4$. Table 7.8 presents these results. It is easy to see that by implementing the k-ary method with the formulas of Section 7.1, one can achieve speed-ups of up to 17 percent. Notice that the optimum value of k is still influenced by the number of additions that are needed to pre-compute the table used in the k-ary method as described in Theorem 4.

Operations	Method/ Type of Operation	Avg. Timing (175 MHz)	Avg. Timing (233 MHz)
Basic Field Operations	176 x 176 bit multiplication	38.6 μsec	29.9 μsec
	176 bit inverse	158.7 μsec	115.7 μsec
Multiply new elliptic curve point ($n \rightarrow 176$ bits)	Double and Add Algorithm	92.7 msec	64.6 msec
	k-ary method ($k = 4$)	82.7 msec	61.7 msec
	k-ary method with formulae ($k = 4$)	**68.3** msec	**49.2** msec
	Improved k-ary ($k = 5$)	75.8 msec	58.3 msec
	Improved k-ary with formulae ($k = 5$)	69.3 msec	50.4 msec
Multiply known elliptic curve point ($n \rightarrow 176$ bits)	Brickell's Algorithm ($base = 2^4 = 16$)	19.7 msec	13.4 msec

Table 7.9: Timings for various field and elliptic curve operations.

Table 7.9 presents the timings for various arithmetic operations and for several algorithms used to compute nP. Notice also that the last entry of Table 7.9

corresponds to the nP calculation when the point P is known ahead of time, thus it is possible to pre-compute a table of multiples of P. This implies that the formulae derived in Section 7.1 were not used in this algorithm. Finally, that the timings of Table 7.9 do not show any timing improvement due to the use of the improved k-ary method contrary to the predictions made in Section 7.3. In fact the timings indicate that the improved k-method with formulae is slower than the k-ary method with formulae. This can be explained by the fact that in the k-ary method one only realizes doublings with the formulae while in the improved k-ary method you need to use the double and add algorithm in some cases since the formulae are not effective for small numbers of doublings (1 or 2). This inclusion of the double and add algorithm will create overhead which eventually will make the improved k-ary method slower than the k-ary method.

Chapter 8

Elliptic Curve Key Exchange Protocols

This chapter is based on the discussion in [Bea96, Chapter 5]. This chapter introduces several elliptic curve based cryptosystems. An analog of the Diffie-Hellman key exchange protocol will be presented as well as other systems whose one-way functions have been replaced by elliptic curves. Finally, we provide timing estimates for a software implementation of the elliptic curve analog of the Diffie-Hellman key exchange protocol.

8.1 Elliptic Curve Analog to Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange algorithm can easily be implemented using elliptic curves. Let's suppose that Alice and Bob want to agree upon a key which will later be used in conjunction with a private-key cryptosystem. They first publicly choose

an elliptic curve E over a finite field $GF(2^k)$. Each of their keys will be constructed using a point $\alpha \in E(GF(2^k))$ which they have randomly chosen and made public. Point α is the generator of a cyclic subgroup of E, and the group order should be of the same magnitude as $GF(2^k)$.

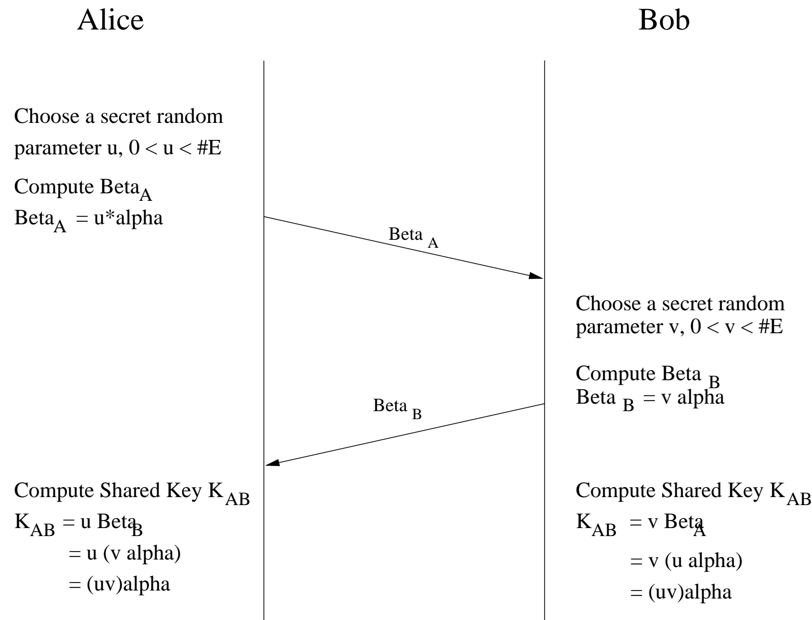

Figure 8.1: Elliptic curve key exchange protocol

Alice and Bob now have the same key $K_{uv} = u\,v\,\alpha \in E$. Security is gained by the intractability of finding point K_{uv}. Notice that without solving the elliptic curve discrete logarithm problem, which is defined as follows:

$$\text{Given: } \alpha,\ \beta \in E \text{ such that: } \beta = n\,\alpha, \text{where } n \text{ is an integer.}$$

$$\text{Determine: } n.$$

there seems to be no efficient way to compute $u\,v\,\alpha$ knowing only $u\,\alpha$ and $v\,\alpha$.

8.2 Proposed IEEE Standard

The next cryptosystem that will be described is a draft of a proposed IEEE Standard [MQV95]. Changes may occur in the design of such a system, since it is still in the working stages.

This system is designed to be an analog to the ElGamal public-key cryptosystem. However, unlike the previous system discussed, this system does not contain the same drawbacks that were previously mentioned. The idea behind this proposal, is to standardize a method, which is secure, and at the same time able to be implemented in software and hardware at reasonable speeds.

This method, like the others we discussed, has the same system setup. An elliptic curve E defined over a finite field $GF(q)$ is chosen. A point P with order n is then selected. These values are all public information.

Again, generating the public and private keys is similar to the methods already discussed. For the following discussion: *Bob* is sending a message M to *Alice*.

Key Generation

1. Select a random integer d in the range $\{1 \rightarrow n-1\}$.
2. Compute the point $Q := dP$.
3. The entity's public key is the point Q.
4. The entity's private key is the integer d.

Encryption Process

1. *Bob* gets *Alice's* public key: Q.

2. The message M is represented as a pair of field elements (m_1, m_2), $m_1 \in GF(q)$, $m_2 \in GF(q)$.

3. Select a random integer k in the range $\{1 \rightarrow n-1\}$.

4. Compute the point $(x_1, y_1) := kP$.

5. Compute the point $(x_2, y_2) := kQ$.

6. The field elements $m_1,\ m_2$, and x_2 are combined in a predetermined manner to obtain the two field elements c_1 and c_2. (Discussed below)

7. Transmit the data $c := (x_1, y_1, c_1, c_2)$ to *Alice*.

Decryption Process

1. Compute the point $(x_2, y_2) := d(x_1, y_1)$, using its private key d.

2. Recover the message m_1 and m_2 from c_1, c_2, and x_2.

What makes this method more secure than the one discussed earlier, is the way that the field elements $m_1,\ m_2$, and x_2 are combined. They are combined in a manner in which an intruder who knows c_1, c_2 and half the message, say m_1, cannot recover the second half of the message m_2, nor will he be able to substitute m_1 by another message m_1' of his choice. The following method is used to encrypt the message and form the field elements $c_1,\ c_2$:

1. Form the field element x_3 by setting to 0 the most significant bit of x_2.

2. Compute the field element x_2^3.

3. Form the field element y_3 by setting to 0 the most significant bit of x_2^3.

4. Form the field element x_4 by concatenating the most significant bits of x_3 followed by the least significant bits of y_3.

5. Form the field element y_4 by concatenating the most significant bits of y_3 followed by the least significant bits of x_3.

6. Compute $z_1 := m_1 \oplus y_3$ and $z_2 := m_2 \oplus x_3$, where $\oplus$ is bitwise XOR.

7. Perform field multiplications to get $c_1 := x_4 * z_1$ and $c_2 := y_4 * z_2$.

Message expansion can be reduced to about 3/2 if we represent point P by its x-coordinate and one bit of y_1. We can do this using the method that is outlined in the appendix.

8.3 Performance Analysis

In this section, we provide a comparison between the speeds that elliptic curve cryptosystems can attain and the speeds of other algorithms such as the Diffie-Hellman key exchange over prime fields. It is assume that both algorithms are run on a DEC Alpha 3000 with a 175 MHz clock frequency. Finally it is important to point out that the Diffie-Hellman key exchange times over prime fields are not estimates but rather the timings reported in [Bea96]. The elliptic curve key-exchange algorithm

	DH 512	DH 1024	EC 176
Key exchange times	1.16sec	8.62sec	88 msec

Table 8.1: Time comparison for key-exchange algorithms: Modulo arithmetic vs. elliptic curves over the field $GF((2^{16})^{11})$.

outperforms the Diffie-Hellman algorithm for both the 512 and 1024 bit modulus. It is important to point out that the times reported in Table 8.1 are average times, since

each key exchange will depend on the specific exponent used. Finally, notice that the elliptic curve that was used provide the same level of security as a Diffie-Hellman key exchange with 1200 bits, thus, the time improvement that is obtained with elliptic curves is even more dramatic.

Chapter 9

Discussion

This section will summarize the results that were obtained throughout the research work that culminated with this thesis. A summary of the chapters and their main results will be provided as well as some recommendations for future research.

9.1 Conclusions

This research has demonstrated that elliptic curve cryptosystems are well suited for cryptographic applications and can be used in practical applications. Furthermore, elliptic curve cryptosystems are a logical alternative to other systems based in DL problem over finite fields because of their security and their efficiency derived from their short key lengths. Three algorithms were developed which can be used to construct cryptosystems that outperform existing systems today. The main achievements of this research included:

- Demonstrated an efficient implementation of the multiplication operation in the composite field $GF((2^{16})^{11})$ that outperforms by ten percent a straightforward

approach. This was achieved by optimizing and analyzing the Karatsuba-Ofman algorithm for multiplication in composite fields for software implementations.

- Demonstrated an efficient implementation of the Itoh and Tsujii's algorithm for computing inverses in composite fields in standard base representation which had been initially tailored for Galois fields in normal base representation.

- Developed a totally new approach for point multiplication, the core operation in the Diffie-Hellman key exchange protocol and the generation of digital signatures. This approached proved to provide our implementation with a 17 percent improvement over the standard k-ary method for exponentiation.

- The software implementation achieved a time of 49.21 msec in a DEC Alpha with a 233 MHz clock frequency for performing a whole elliptic curve point multiplication, which is essentially the time needed for a Diffie-Hellman key exchange or signature generation.

- Provided time estimates for an elliptic curve key-exchange protocol that outperforms the Diffie-Hellman key exchange algorithm based on modulo-p arithmetic with 1024 bits by a factor of almost 100.

9.2 Recommendations for Further Research

This section will provide the reader with an overview of the possible areas in which further work could be pursued. Many of the ideas have come up as a result of this research and provide opportunities to investigate further the possibilities of the algorithms that were developed in this thesis.

9.2.1 Generalization of the Improved Point Multiplication

Chapter 7 explored a new approach for point doubling in non-supersingular elliptic curves over fields of characteristic 2. It would be interesting to explore the benefits of this idea as applied to curves over fields of characteristic 3, prime fields, and for supersingular and hyperelliptic curves. Also, from a theoretical point of view, it would be nice if general formulas for the computation of a multiple of a point could be derived so as to provided a formula for any point $Q_k = 2^k P$ in terms of the previous formula for $Q_{k-1} = 2^{k-1} P$.

9.2.2 Implementation of a Variant of the Improved k-ary Method

It was found in Chapter 7 that the improved k-ary method did not outperform the k-ary method. On possible solution would be to derive further formulas for 6,7,8,9,and 10 doublings in a row and allow for more doublings in each iteration of the algorithm without altering the value of k that was found to be optimum.

9.2.3 Implementation of the Sliding Window Method for Exponentiation

The sliding window method for exponentiation, it is a generalization of the k-ary and improved k-ary methods which uses variable values for k, known as the window size. A possible project would be to explore this algorithm and the feasibility of applying the doubling formulas to it.

9.2.4 Implementation of a Different Inversion Algorithm

It was found in Chapter 6 that the exponentiation algorithm that was used in our implementation was not the most efficient one. Thus, it would be necessary to implement the algorithms that were suggested in [WBV+96] for inversion in composite fields.

Appendix A

Proofs for Doubling Formulas

A.1 Proof for Theorem 13

In the following, we will derive Theorem 13. We assume that we have obtained (7.1) and (7.2) to calculate $2^2P = (x_2, y_2)$ and that we are trying to find similar expressions for $2^3P = (x_3, y_3)$. In addition, we keep the notation of Theorem 12 in which $\gamma = x^2$,$\eta = \gamma + y$, $\delta = \eta^2 + \eta x + a\gamma$, $\xi = \eta x + \gamma$, and $\zeta = \delta(\delta + \xi) + \gamma^2\gamma$, and $P = (x, y)$.

Then, by writing (2.4) and (2.5) in terms of x_2, y_2, x_3, and y_3, one finds:

$$\begin{aligned} x_3 &= \frac{(x_2^2 + y_2)^2 + (x_2^2 + y_2)x_2 + ax_2^2}{x_2^2} \\ y_3 &= \frac{x_2^2 x_2 + (x_2^2 + y_2)x_3}{x_2} + x_3 \end{aligned}$$

Next we consider the term $x_2^2 + y_2$ by plugging in (7.1) and (7.2) for x_2 and y_2,

respectively. Then, if we let $\tau = \delta\gamma$ and $\upsilon = \zeta^2 + \tau\zeta + \tau^2 a$, we get:

$$x_2^2 + y_2 = \frac{\upsilon^2 + \zeta\tau\upsilon + \tau^2\delta^4 + \tau^2\upsilon}{\tau^4}$$

Plugging this expression into the expression for x_3, we get:

$$x_3 = \frac{\left[\frac{\upsilon^2+\zeta\tau\upsilon+\tau^2\delta^4+\tau^2\upsilon}{\tau^4}\right]^2 + \left[\frac{\upsilon^2+\zeta\tau\upsilon+\tau^2\delta^4+\tau^2\upsilon}{\tau^4}\right]\left(\frac{\upsilon}{\tau^2}\right)}{\frac{\upsilon^2}{\tau^4}} + a$$

Letting $\rho = \upsilon\tau^2$ and $\omega = \upsilon(\upsilon+\zeta\upsilon)+(\tau\delta^2)^2+\rho$ and simplifying we immediately obtain (7.3). Notice that (7.4) is also easily obtained by first substituting $x_2 = \frac{\upsilon}{\tau^2}$ and plugging this expression into the equation for y_3 to obtain:

$$y_3 = \frac{\left(\frac{\upsilon}{\tau^2}\right)\frac{\upsilon}{\tau^2} + \frac{\omega}{\tau^4}x_3}{\frac{\upsilon}{\tau^2}} + x_3$$

which after simplification yields (7.4).□

A.2 Proof for Theorem 14

In the following, we will derive Theorem 14. We assume that we have obtained (7.3) and (7.4) to calculate $2^3P = (x_3, y_3)$ and that we are trying to find similar expressions for $2^4P = (x_4, y_4)$. In addition, we keep the notation of Theorem 13 in which $\gamma = x^2$,$\eta = \gamma + y$, $\delta = \eta^2 + \eta x + a\gamma$, $\xi = \eta x + \gamma$, $\zeta = \delta(\delta + \xi) + \gamma^2\gamma$,$\tau = \delta\gamma$, $\upsilon = \zeta^2 + \tau\zeta + \tau^2 a$, $\rho = \upsilon\tau^2$, $\omega = \upsilon(\upsilon + \zeta\tau) + (\tau\delta^2)^2 + \rho$, and $P = (x, y)$.

Then, by writing (2.4) and (2.5) in terms of x_3, y_3, x_4, and y_4, one finds:

$$x_4 = \frac{(x_3^2 + y_3)^2 + (x_3^2 + y_3)x_3 + ax_3^2}{x_3^2}$$

$$y_4 \quad = \quad \frac{x_3^2 x_3 + (x_3^2 + y_3)x_4}{x_3} + x_4$$

Next we use (7.3) and (7.4) and consider the term $x_3^2 + y_3$. Thus, we get:

$$x_3^2 + y_3 = \frac{(\omega^2 + \omega\rho + \rho^2 a)^2}{\rho^4} + \frac{\upsilon^4\rho^2 + \omega\rho(\omega^2 + \omega\rho + \rho^2 a) + \rho^2(\omega^2 + \omega\rho + \rho^2 a)}{\rho^4}$$

Letting $\mu = \omega^2 + \omega\rho + \rho^2 a$ and $\theta = \mu(\mu + \rho(\omega + \rho)) + (\upsilon^2\rho)^2$, we get:

$$x_3 = \frac{\mu}{\rho^2}$$

and

$$x_3^2 + y_3 = \frac{\theta}{\rho^4}$$

Plugging this expression into the equation for x_4, we get

$$x_4 = \frac{\left(\frac{\theta}{\rho^4}\right)^2 + \frac{\theta}{\rho^4}\frac{\mu}{\rho^2}}{\left(\frac{\mu}{\rho^2}\right)^2} + a$$

which after simplifying turns into (7.5). Similarly, we write y_4 in terms of x_3 and y_3 as

$$y_5 = \frac{\left(\frac{\mu}{\rho^2}\right)^2\left(\frac{\mu}{\rho^2}\right) + \frac{\theta}{\rho^4}x_4}{\frac{\mu}{\rho^2}} + x_4$$

Simplifying, we get (7.6).□

A.3 Proof for Theorem 15

In the following, we will derive Theorem 15. We assume that we have obtained (7.5) and (7.6) to calculate $2^4P = (x_4, y_4)$ and that we are trying to find similar expressions for $2^5P = (x_5, y_5)$. In addition, we keep the notation of Theorem 15 in

which $\gamma = x^2$,$\eta = \gamma + y$, $\delta = \eta^2 + \eta x + a\gamma$, $\xi = \eta x + \gamma$, $\zeta = \delta(\delta + \xi) + \gamma^2\gamma$,$\tau = \delta\gamma$, $\upsilon = \zeta^2 + \tau\zeta + \tau^2 a$, $\rho = \upsilon\tau^2$, $\omega = \upsilon(\upsilon + \zeta\tau) + (\tau\delta^2)^2 + \rho$, $\mu = \omega^2 + \omega\rho + a\rho^2$, $\theta = \mu^2 + \mu(\omega\rho) + \mu\rho^2 + (\upsilon^2\rho)^2$, and $P = (x, y)$.

We then re-write (2.4) and (2.5) in terms of x_4, y_4, x_5, and y_5 as:

$$\begin{aligned} x_5 &= \frac{(x_4^2 + y_4)^2 + (x_4^2 + y_4)x_4 + ax_4^2}{x_4^2} \\ y_5 &= \frac{x_4^2 x_4 + (x_4^2 + y_4)x_5}{x_4} + x_5 \end{aligned}$$

Next, consider the term $x_4^2 + y_4$ and use (7.5). Using(7.6) for x_4 and y_4, we find:

$$x_4^2 + y_4 = \frac{(\theta^2 + \theta\mu\rho^2 + a(\mu\rho^2)^2)^2}{(\mu\rho^2)^4} + \frac{\left[(\mu^2)^2 + (\theta\mu\rho^2)\frac{\left(\theta^2 + \theta\mu\rho^2 + a(\mu\rho^2)^2\right)}{(\mu\rho^2)^2}\right]}{(\mu\rho^2)^2} + \frac{(\theta^2 + \theta\mu\rho^2 + a(\mu\rho^2)^2)}{(\mu\rho^2)^2}$$

Letting $\kappa = \mu\rho^2$, $\beta = \theta^2 + \theta\kappa + a\kappa^2$, and $\sigma = \beta(\beta + \theta\kappa + \kappa^2) + \kappa^2(\mu^2)^2$, one obtains:

$$x_4^2 + y_4 = \frac{\sigma}{(\kappa^2)^2}$$

Then:

$$x_5 = \frac{\left(\frac{\sigma}{(\kappa^2)^2}\right)^2 + \frac{\sigma}{(\kappa^2)^2}\frac{\beta}{\kappa^2}}{\frac{\beta^2}{\kappa^4}} + a$$

which readily simplifies into (7.7). In a similar manner, we obtain:

$$y_5 = \frac{\left(\frac{\beta}{\kappa^2}\right)^2 \frac{\beta}{\kappa^2} + \frac{\sigma}{(\kappa^2)^2} x_5}{\frac{\beta}{\kappa^2}} + x_5$$

which turns into (7.8) after simplification.$\square$

Bibliography

[ABMV93] G. Agnew, T. Beth, R. Mullin, and S. Vanstone. Arithmetic operations in $GF(2^m)$. *Journal of Cryptology*, 6:3–13, 1993.

[AMV93] G.B. Agnew, R.C. Mullin, and S.A. Vanstone. An implementation of elliptic curve cryptosystems over $F_{2^{155}}$. *IEEE Journal on Selected areas in Communications*, 11(5):804–813, June 1993.

[Bea96] D. Beauregard. Efficient algorithms for implementing elliptic curve public-key schemes. Master's thesis, ECE Dept., Worcester Polytechnic Institute, Worcester, MA, May 1996.

[BGMW92] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation (extended abstract). In *Advances in Cryptology — EUROCRYPT '92*, pages 200–207. Springer-Verlag, 1992.

[Coh93] H. Cohen. *A Course in Computational Algebraic Number Theory.* Springer-Verlag, Berlin, 1993.

[DH76] W. Diffie and M.E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, 1976.

[ElG85] T. ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[HMV92] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In Rainer A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT '92*, pages 163–173, Berlin, 1992. Springer-Verlag. Lecture Notes in Computer Science Volume 658.

[HPR] O. Hooijen, C. Paar, and S. Rees. Composite Galois field library. Institute for Experimental Mathematics, University of Esssen, Germany.

[IT88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[Jun93] D. Jungnickel. *Finite Fields.* B.I.-Wissenschaftsverlag, Mannheim, Leipzig, Wien, Zürich, 1993.

[Kaz92] O. Kazarin. Use of properties of elliptic curves. *Automatic control and computer sciences*, 26(5):19–26, 1992.

[KMQV96] J. Koeller, A. Menezes, M. Qu, and S. Vanstone. Elliptic Curve Systems. Draft 8, IEEE P1363 Standard for RSA, Diffie-Hellman and Related Public-Key Cryptography, May 1996. working document.

[Knu81] D.E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys.-Dokl. (Engl. transl.)*, 7(7):595–596, 1963.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[Koc95] C. K. Koc. Analysis of sliding window techniques for exponentiation. *Computers and Mathematics with Applications*, 30(10):17–24, November 1995.

[LC83] S. Lin and D.J. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice-Hall, Englewood Cliffs, NJ, 1983.

[LN83] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, Massachusetts, 1983.

[McE87] R.J. McEliece. *Finite Fields for Computer Scientists and Engineers*. Kluwer Academic Publishers, 1987.

[Men93a] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, Boston, 1993.

[Men93b] A.J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[Mil86] V. Miller. Use of Elliptic Curves in Cryptography. In Hugh C. Williams, editor, *Advances in Cryptology — CRYPTO '85*, pages 417–428, Berlin, 1986. Springer-Verlag. Lecture Notes in Computer Science Volume 218.

[MOV93] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Transactions on Information Theory*, 39(5):1639–1646, September 1993.

[MQV95] A. Menezes, M. Qu, and S. Vanstone. Elliptic curve systems. *Proposed IEEE P1363 Standard*, pages 1–42, April 1995.

[MV90] A. Menezes and S. Vanstone. The implementation of elliptic curve cryptosystems. In *Advances in Cryptology — AUSCRYPT '90*, pages 2–13, January 1990.

[MV93] A. Menezes and S. Vanstone. Elliptic curve cryptosystems and their implementation. *Journal of Cryptography*, 6:209–224, 1993.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, Florida, 1997.

[Paa94] C. Paar. *Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields*. PhD thesis, (Engl. transl.), Institute for Experimental Mathematics, University of Essen, Essen, Germany, June 1994.

[Paa95] C. Paar. Some remarks on efficient inversion in finite fields. In *1995 IEEE International Symposium on Information Theory*, page 58, Whistler, B.C. Canada, September 17–22 1995.

[Paa96] C. Paar. A new architecture for a parallel finite field multiplier with low complexity based on composite fields. *IEEE Transactions on Computers*, 45(7):856–861, July 1996.

[RSA78] R.L. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast Key Exchange with Elliptic Curve Systems. In Don Coppersmith, editor, *Advances in Cryptology — CRYPTO '95*, pages 43–56, Berlin, 1995. Springer-Verlag. Lecture Notes in Computer Science Volume 963.

[WBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT '96*, pages 65–76, Berlin, 1996. Springer-Verlag. Lecture Notes in Computer Science Volume 1233.